Function pointers and address of a function

So I figured when making function pointers, you do not need the `operator &` to get the address of the initial function:

```
#include <stdio.h>

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

int main () {

    double (*fun1)(double) = &foo;
    double (*fun2)(double) =  foo;

    printf("%f\n",fun1(10));
    printf("%f\n",fun2(10));

    printf("fun1 = %p \t &foo = %p\n",fun1, &foo);
    printf("fun2 = %p \t  foo = %p\n",fun2,  foo);

    int a[10];

    printf("  a = %p \n &a = %p  \n",a,&a);

    return 0;
}
```

output:

```
>./a.out
100.000000
100.000000
fun1 = 0x4004f4      &foo = 0x4004f4
fun2 = 0x4004f4       foo = 0x4004f4
  a = 0x7fff26804470
 &a = 0x7fff26804470
```

Then I realized this is also true for arrays, meaning that if you have `int a[10]` both `a` and `&a` point to the same location. Why is that with arrays and functions? Is the address saved in a memory location that has the same address as the value(address) being saved in it?

c function pointers function-pointers

edited Nov 1 '16 at 22:35
skrtbhtngr
1,739 ● 14 ● 26

asked Mar 4 '12 at 5:44
GradGuy
2,370 ● 3 ● 25 ● 39

I never quite understood why you didn't have to use the address-of operator to take the address of a function. I have always assumed that is just a bit of syntactic sugar. – Bill Mar 4 '12 at 5:48

@Bill: Exactly! That's what I always use to think but it seems you don't need to! – GradGuy Mar 4 '12 at 5:51

If the address was saved in the same memory location as itself, there couldn't be the actual function code/array data! You'll practically treat arrays as pointers and functions passed as arguments also as pointers, so you won't need the addressof operator. Basically, the 'value' of a function or an array is nonsense. Only the address makes sense, so you get that when you query the 'value', and also when you query the address. – user529758 Mar 4 '12 at 6:22

5 Answers

Given `int a[10]`, both `a` and `&a` yield the same address, yes, but their types are different.

`a` is of type `int[10]`. When it is implicitly converted to a pointer type, the pointer is of type `int*` and points to the initial element of the array. `&a` is of type `int (*)[10]` (that is, a pointer to an array of ten integers). Because there can be no padding in an array, they both yield pointers with the same *value*, but the pointers have different *types*.

Functions are similar to arrays, but not entirely the same. Your function `foo` is of type `double(double)`. Whenever `foo` is used in an expression and is not the operand of the unary `&` operator, it is implicitly converted to a pointer to itself, which is of type `double(*)(double)`.

So, for all practical purposes, the name of a function and a pointer to the same function are interchangeable. There are some subtleties, all of which I discuss in an answer to "Why do all these crazy function pointer definitions all work? What is really going on?" (That question was asked about C++, but the rules for nonmember functions in C++ are the same as for functions in C.)

edited May 23 at 12:25
Community ♦
1 ● 1

answered Mar 4 '12 at 5:51
James McNellis
265k ● 50 ● 756 ● 877

No, there's no extra storage dedicated to pointing to the function/array.

With most variables `variable_name` has a meaning other than getting the address of that variable, so you need to use `&variable` to get the address.

With a function or array, `function_name` (by itself, not followed by parentheses) doesn't have any other meaning, so there was no problem with interpreting it as taking the address of the function.

Likewise in reverse: a normal pointer needs to be dereferenced explicitly, but a pointer to a function doesn't (again, because there's no other reasonable interpretation), so given a pointer to a function like:

```
int (*func)(param_list);
```

The following are equivalent to each other -- both call whatever function `func` points at:

```
(*func)(params);

func(params);
```

answered Mar 4 '12 at 5:52

Jerry Coffin
350k ● 35 ● 408 ● 829

2 ▲ There seem's to be a slight different though, for int a[10], (*a)[5] is not a valid call – GradGuy Mar 4 '12 at 6:02

Basically, since the function name is "known" to be a function, the & is not strictly necessary. This behavior is the same for arrays. Recall that a function itself is not a variable, so it behaves a little differently than you might expect sometimes. If you have the 2nd edition of K&R, you can check out section 5.11 on pointers to functions, or the reference manual at the end,

> Section A7.1 Pointer generation: If the type of an expression or subexpression is "array of T" for some type T, then the value of the expression is a pointer to the first object in the array, and the type of the expression is altered to "pointer to T." This conversion does not take place of the expression is the operand of the unary & operator, ... Similarly, an expression of type "function returning T," except when used as the operand of the & operator, is converted to "pointer to function returning T."
>
> Section A7.4.2 Address Operator: The unary & operator takes the address of its operand.... The result is a pointer to the object or function referred to by the lvalue. If the type of the operand is T, the type of the result is "pointer to T."

As far as I know, this is the same for C99.

edited Nov 1 '16 at 23:41

skrtbhtngr
1,739 ● 14 ● 26

answered Mar 4 '12 at 6:17

Cameron
1,395 ● 7 ● 12

`fun` and `&fun` are exactly the same (except that sizeof(f) is illegal). `a` and `&a` are the same up to pointer arithmetic: `a + 10 == &a + 1`, because `10*sizeof(*a) == sizeof(a)` (where `sizeof(*a) == sizeof(int)`).

edited Mar 31 '15 at 22:25

answered Mar 31 '15 at 22:14

Loic
541 ● 3 ● 9

> printf("fun1 = %p \t &foo = %p\n",fun1, foo);

Here your are calling `foo` by passing Function Pointer with `pass by value` and

> printf("fun2 = %p \t foo = %p\n",fun2, &foo)

Here you are calling `&foo` by passing function Pointer with `pass by reference`

in both case your are calling the `printf` with function pointer only.

Remember `foo` itself is `function pointer value` and `not a variable.

Same happens with array. `int arr[10]` translates into get continuous block of 10 Integers and address of first element is stored into arr. so arr is also a pointer.

answered Apr 22 '13 at 7:44

rahul maindargi


3,976
1
7
19