

Re: typedef function with void parameters

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2007-12/msg02137.html

- *From:* Eric Sosman <esosman@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Thu, 13 Dec 2007 10:33:44 -0500
-

William Xu wrote:

Chris Dollin <chris.dollin@xxxxxx> writes:

So, which do you want -- a function taking a `void*` or a function taking a `char*`? They're not compatible types. Pick one.

Oh, i thought that most pointer types could be converted/stored into void pointer.

Yes, they can.

I'm trying to wrap different function pointers into FUNC, like,

```
,-----  
| int foo(char *p){ }  
| int bar(int *p){ }  
|| typedef int (*FUNC)(void *);  
|| FUNC f1 = foo, f2 = bar;  
`-----
```

So, this is not allowed?

No, it is not.

Here's why: You can convert a `void*` to or from any other data pointer type, but you need to know what the other type is in order to choose the proper conversion. Calling foo() or bar() with a `void*` argument is all right, because the compiler knows that it must convert `void*` to `char*` for foo() or convert `void*` to `int*` for bar(). But when calling f1() -- more precisely, when calling a function that f1 can point to -- you have told the compiler that no conversion is necessary:

Re: typedef function with void parameters

an f1-addressable function takes a `void*` argument, so there is no need to convert the `void*` that you supply. If f1 points (invalidly) at foo() or bar(), the compiler doesn't know that it should perform a conversion.

On many machines these "conversions" are just bit-for-bit copies of the pointers' values, but C does not require such a simple addressing scheme and is able to run on machines whose addressing is more intricate. So from the point of view of the C language, you must keep the compiler properly informed about the types of things.

Weak analogy: You know how to convert day-to-day natural numbers to and from Roman numerals, and to and from English utterances:

```
1234 <-> "MCCXXXIV"
1234 <-> "one thousand two hundred thirty-four"
```

Let's imagine that your bag of tricks includes C functions to do these conversions, so you can think of them as "built-in" like the conversions between `void*` and `double*`.

Now suppose you're writing a program that involves a few more functions, each taking an argument representing a number as a string. One of them wants a Roman representation, and the other wants an English string:

```
void needsRoman(const char *roman);
void needsEnglish(const char *english);
```

When you want to call one of these with an argument that is actually an integer, you need to do the proper conversion:

```
int x = 42;
needsRoman( intToRoman(x) );
needsEnglish( intToEnglish(x) );
```

What you have tried to do (the analogy approaches its tortured conclusion; the air is thick with anticipation and ennui) is to invent a construct that can pass `x` to either needsRoman() or needsEnglish() without knowing which string representation of `x` is appropriate. Can't be done: You need to know what the called function expects to receive before you can know how to meet its expectations.

—
Eric Sosman
esosman@xxxxxxxxxxxxxxxxxxxxxx