Correct format specifier to print pointer (address)?

Which format specifier should I be using to print the address of a variable? I am confused between the below lot.

> %u - unsigned integer
>
> %x - hexadecimal value
>
> %p - void pointer

Which would be the optimum format to print an address?

c pointers format memory-address

edited Sep 11 at 5:10
rsp
7,726 ●1 ●23 ●59

asked Jan 29 '12 at 13:49
San
1,065 ●5 ●21 ●34

4 Answers

The simplest answer, assuming you don't mind the vagaries and variations in format between different platforms, is the standard `%p` notation.

The C99 standard (ISO/IEC 9899:1999) says in §7.19.6.1 ¶8:

> `p` The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

(In C11 — ISO/IEC 9899:2011 — the information is in §7.21.6.1 ¶8.)

On some platforms, that will include a leading `0x` and on others it won't, and the letters could be in lower-case or upper-case, and the C standard doesn't even define that it shall be hexadecimal output though I know of no implementation where it is not.

It is somewhat open to debate whether you should explicitly convert the pointers with a `(void *)` cast. It is being explicit, which is usually good (so it is what I do), and the standard says 'the argument shall be a pointer to `void`'. On most machines, you would get away with omitting an explicit cast. However, it would matter on a machine where the bit representation of a `char *` address for a given memory location is different from the '*anything else pointer*' address for the same memory location. This would be a word-addressed, instead of byte-addressed, machine. Such machines are not common (probably not available) these days, but the first machine I worked on after university was one such (ICL Perq).

If you aren't happy with the implementation-defined behaviour of `%p`, then use C99 `<inttypes.h>` and `uintptr_t` instead:

```
printf("0x%" PRIXPTR "\n", (uintptr_t)your_pointer);
```

This allows you to fine-tune the representation to suit yourself. I chose to have the hex digits in upper-case so that the number is uniformly the same height and the characteristic dip at the start of `0xA1B2CDEF` appears thus, not like `0xa1b2cdef` which dips up and down along the number too. Your choice though, within very broad limits. The `(uintptr_t)` cast is unambiguously recommended by GCC when it can read the format string at compile time. I think it is correct to request the cast, though I'm sure there are some who would ignore the warning and get away with it most of the time.

Kerrek asks in the comments:

> I'm a bit confused about standard promotions and variadic arguments. Do all pointers get standard-promoted to void*? Otherwise, if `int*` were, say, two bytes, and `void*` were 4 bytes, then it'd clearly be an error to read four bytes from the argument, non?

I was under the illusion that the C standard says that all object pointers must be the same size, so `void *` and `int *` cannot be different sizes. However, what I think is the relevant section of the C99 standard is not so emphatic (though I don't know of an implementation where what I suggested is true is actually false):

> §6.2.5 Types
>
> ¶26 A pointer to void shall have the same representation and alignment requirements as a pointer to a character type.[39] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.
>
> [39] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

(C11 says exactly the same in the section §6.2.5, ¶28, and footnote 48.)

So, all pointers to structures must be the same size as each other, and must share the same alignment requirements, even though the structures the pointers point at may have different alignment requirements. Similarly for unions. Character pointers and void pointers must have the same size and alignment requirements. Pointers to variations on `int` (meaning `unsigned int` and `signed int`) must have the same size and alignment requirements as each other; similarly for other types. But the C standard doesn't formally say that `sizeof(int *) == sizeof(void *)`. Oh well, SO is good for making you inspect your assumptions.

The C standard definitively does not require function pointers to be the same size as object pointers. That was necessary not to break the different memory models on DOS-like systems. There you could have 16-bit data pointers but 32-bit function pointers, or vice versa. This is why the C standard does not mandate that function pointers can be converted to object pointers and vice versa.

Fortunately (for programmers targetting POSIX), POSIX steps into the breach and does mandate that function pointers and data pointers are the same size:

> §2.12.3 Pointer Types
>
> All function pointer types shall have the same representation as the type pointer to void. Conversion of a function pointer to `void *` shall not alter the representation. A `void *` value resulting from such a conversion can be converted back to the original function pointer type, using an explicit cast, without loss of information.
>
> Note: The ISO C standard does not require this, but it is required for POSIX conformance.

So, it does seem that explicit casts to `void *` are strongly advisable for maximum reliability in the code when passing a pointer to a variadic function such as `printf()`. On POSIX systems, it is safe to cast a function pointer to a void pointer for printing. On other systems, it is not necessarily safe to do that, nor is it necessarily safe to pass pointers other than `void *` without a cast.

edited Nov 6 '14 at 16:12

answered Jan 29 '12 at 14:16
Jonathan Leffler
497k ● 75 ● 576 ● 921

---

3 I'm a bit confused about standard promotions and variadic arguments. Do all pointers get standard-promoted to `void*`? Otherwise, if `int*` were, say, two bytes, and `void*` were 4 bytes, then it'd clearly be an error to read four bytes from the argument, non? – Kerrek SB Jan 29 '12 at 17:30

Note that an update to POSIX (POSIX 2013) has removed section 2.12.3, moving most of the requirements to the `dlsym()` function instead. One day I'll write up the change...but 'one day' is not 'today'. – Jonathan Leffler Jun 4 '14 at 18:18

Does this answer also apply to pointers to functions? Can they be converted to `void *`? Hmm I see your comment here. Since only one-wat conversion is needed (function pointer to `void *`), it works then? – chux Oct 3 '15 at 13:59 ✎

@chux: Strictly, the answer is 'no', but in practice the answer is 'yes'. The C standard does not guarantee that function pointers can be converted to a `void *` and back without loss of information. Pragmatically, there are very few machines where the size of a function pointer is not the same as the size of an object pointer. I don't think the standard provides a method of printing a function pointer on machines where tha conversion is problematic. – Jonathan Leffler Oct 3 '15 at 14:07

"and back without loss of information" is not relevant to printing. Does that help? – chux Oct 3 '15 at 14:19

`p` is the conversion specifier to print pointers. Use this.

```
int a = 42;

printf("%p\n", (void *) &a);
```

Remember that omitting the cast is undefined behavior and that printing with `p` conversion specifier is done in an implementation-defined manner.

answered Jan 29 '12 at 13:51
ouah
113k ● 12 ● 170 ● 260

Pardon, why omitting the cast is "undefined behavior"? Does this matter address of which variable it is, if all you need is the address, not the value? – valdo Jan 29 '12 at 13:54

7 @valdo because C says it (C99, 7.19.6.1p8) "p The argument shall be a pointer to void." – ouah Jan 29 '12 at 13:55

7 @valdo: It's not necessarily the case that all pointers are the same size / representation. – caf Jan 29 '12 at 14:06

Use `%p`, for "pointer", and don't use anything else*. You aren't guaranteed by the standard that you are allowed to treat a pointer like any particular type of integer, so you'd actually get undefined behaviour with the integral formats. (For instance, `%u` expects an `unsigned int`, but what if `void*` has a different size or alignment requirement than `unsigned int`?)

*) [See Jonathan's fine answer!] Alternatively to `%p`, you *can* use pointer-specific macros from `<inttypes.h>`, added in C99.

All object pointers are implicitly convertible to `void*` in C, but in order to pass the pointer as a variadic argument, you have to cast it explicitly (since arbitrary object pointers are only *convertible*, but not *identical* to void pointers):

```
printf("x lives at %p.\n", (void*)&x);
```

edited Jan 29 '12 at 17:29

answered Jan 29 '12 at 13:52
Kerrek SB
324k ● 54 ● 603 ● 841

2 All *object* pointers are convertible to `void *` (though for `printf()` you technically need the explicit cast, since it's a variadic function). Function pointers aren't necessarily convertible to `void *`. – caf Jan 29 '12 at 13:54

@caf: Oh, I didn't know about the variadic arguments - fixed! Thanks! – Kerrek SB Jan 29 '12 at 13:58

2 Standard C does not require that function pointers be convertible to `void *` and back to function pointer without loss; fortunately, though, POSIX does explicitly require that (noting that it is not part of standard C). So, in practice, you can get away with it (converting `void (*function)(void)` to `void *` and back to `void (*function)(void)`), but strictly it is not mandated by the C standard. – Jonathan Leffler Jan 29 '12 at 14:18

2 Jonathan and R.: This is all very interesting, but I'm pretty sure we're not trying to print function pointers here, so perhaps this isn't quite the right place to discuss this. I'd much rather see some support here for my insistence to not use `%u`! – Kerrek SB Jan 29 '12 at 16:08

2 `%u` and `%lu` are wrong on **all machines**, not some machines. The specification of `printf` is very clear that when the type passed does not match the type required by the format specifier, the behavior is undefined. Whether the size of the types matches (which could be true or false, depending on the machine) is irrelevant; it's the types that must match, and they never will. – R.. Jan 29 '12 at 18:09

As an alternative to the other (very good) answers, you could cast to `uintptr_t` or `intptr_t` (from `stdint.h` / `inttypes.h`) and use the corresponding integer conversion specifiers. This would allow more flexibility in how the pointer is formatted, but strictly speaking an implementation is not required to provide these typedefs.

answered Jan 29 '12 at 15:49
R..
140k ● 18 ● 219 ● 486

Nice, didn't know about `uintptr_t` . – cnicutar Jan 29 '12 at 20:41

consider `#include <stdio.h> int main(void) { int p=9; int* m=&s; printf("%u",m); }` **is it undefined behaviour to print address of variable using `%u` format specifier ?** Address of variable in most cases is positive so can I use `%u` instead of `%p` ? – Destructor Dec 19 '16 at 16:27

@Destructor: No, `%u` is a format for `unsigned int` type and cannot be used with a pointer argument to `printf` . – R.. Dec 19 '16 at 18:07