Should I cast the result of malloc (in C)?

Asked 15 years, 9 months ago Modified 5 months ago Viewed 401k times

2810

In this question, someone suggested in a comment that I should ***not*** cast the result of `malloc`. i.e., I should do this:

```
int *sieve = malloc(sizeof(*sieve) * length);
```

rather than:

```
int *sieve = (int *) malloc(sizeof(*sieve) * length);
```

Why would this be the case?

c malloc casting

Share
Improve this question
Follow

edited Mar 28 at 15:12

asked Mar 3, 2009 at 10:13

Patrick McDonald
65.3k ● 14 ● 112 ● 126

38 stackoverflow.com/q/7545365/168175 – Flexo - Save the data dump ♦ Jan 15, 2012 at 20:06

11 Casts are evil. I see so many cast in code just as a result of bad coding practice. Whenever you need to insert one the first thing you should ask yourselves is " what is wrong here" . Is everything declared as it should be ? If it is no cast would be needed so something is declared wrong. If you really do need to do some low level stuff on individual bytes in an int or so consider a union to access them. That'll declare them just fine. As a rule of thumb do not insert them unless the compiler complains. Then avoid them. This example will not complain. void pointer will promote to any type. – Hans Lepoeter Feb 21, 2021 at 21:52

2 @HansLepoeter in C++ , those are necessary for malloc, giving some basis to my notion that there's something wrong with it – An Ant Feb 27, 2021 at 11:51

6 @AnAnt C++ is not C. Their type systems are totally different. Observations about C++'s type system do not necessarily apply to C. In fact, they rarely do. – Braden Best Oct 14, 2022 at 6:23

1 @caulder "casting to a baseclass in C"--good luck with that! Hint: What's the name of C with classes? – Karoly Nyisztor Jul 18, 2023 at 15:28

30 Answers

Sorted by: Highest score (default)

2560

No; you *shouldn't* cast the result, since:

- It is unnecessary, as `void *` is automatically and safely promoted to any other pointer type in this case.
- It adds clutter to the code, casts are not very easy to read (especially if the pointer type is long).
- It makes you repeat yourself, which is generally bad.
- It can hide an error if you forgot to include `<stdlib.h>`. This can cause crashes (or, worse, *not* cause a crash until way later in some totally different part of the code). Consider what happens if pointers and integers are differently sized; then you're hiding a warning by casting and might lose bits of your returned address. Note: as of C99 implicit functions are gone from C, and this point is no longer relevant since there's no automatic assumption that undeclared functions return `int`.

As a clarification, note that I said "you *shouldn't* cast", not "you *don't need to* cast". In my opinion, it's a blunder to include the cast, even if you got it right. There are simply no benefits to doing it, but a bunch of potential risks, and including the cast indicates that you don't know about the risks.

Also note, as commentators point out, that the above talks about straight C, not C++. I very firmly believe in C and C++ as separate languages.

To add further, your code needlessly repeats the type information (`int`) which can cause errors. It's better to de-reference the pointer being used to store the return value, to "lock" the two together:

```
int *sieve = malloc(length * sizeof *sieve);
```

Some may say: "Well, previously the type was repeated, and now the variable name is repeated; isn't this just as repetitive as before? How is that any better?" The difference is that if you one day change the type of the variable and forget to change the type under the `sizeof` to match, you will silently get an allocation of the wrong size and no warning about it; but if you change the name of the variable, but forget to change the name under `sizeof` to match, it is more probable that the old name no longer resolves to anything, so your code will stop compiling, prompting you to fix the mistake.

This also moves the `length` to the front for increased visibility, and drops the redundant parentheses with `sizeof`; they are only needed when the argument is a

type name. Many people seem to not know (or ignore) this, which makes their code more verbose. Remember: `sizeof` is not a function! :)

While moving `length` to the front *may* increase visibility in some rare cases, one should also pay attention that in the general case, it should be better to write the expression as:

```
int *sieve = malloc(sizeof *sieve * length);
```

Since keeping the `sizeof` first, in this case, ensures multiplication is done with at least `size_t` math.

Compare: `malloc(sizeof *sieve * length * width)` vs. `malloc(length * width * sizeof *sieve)` the second may overflow the `length * width` when `width` and `length` are smaller types than `size_t`.

Share

Improve this answer

Follow

edited Sep 6, 2023 at 15:00

dumbass

27.2k ● 4 ● 35 ● 72

answered Mar 3, 2009 at 10:17

unwind

399k ● 64 ● 485 ● 617

41 Compilers have changed. An up-to-date compiler will warn you about a missing declaration of malloc. – n. m. could be an AI Dec 8, 2016 at 12:40

114 @n.m. Ok. I think it's bad to assume that anyone reading here has a particular compiler. Also, since C11 the entire "implicit function" concept is gone, I didn't know that. Still, I don't see the point in adding a pointless cast. Do you also do `int x = (int) 12;` just to make things clear? – unwind Dec 8, 2016 at 12:48

57 @n.m. if explicitly casting a void pointer "helped" solve a bug, you more likely encountered undefined behavior, which would mean the program in question likely has a far worse, undiscovered bug that you haven't run into yet. And one day, on a cold winter evening, you'll come home from work to find your GitHub page flooded with issue reports complaining about demons flying out of the users' noses – Braden Best Dec 17, 2016 at 18:49

41 @unwind Even I agree with you, `(int)12` is not comparable. `12` *is* an `int`, the cast does simply nothing. The retval of `malloc()` is `void *`, not the pointer type casted to. (If it is not `void *`. So the analogy to `(int)12` would be `(void*)malloc(…)` what nobody is discussing.) – Amin Negm-Awad Jan 8, 2017 at 4:30

13 "I think it's bad to assume that anyone reading here has a particular compiler. Also, since C11 the entire "implicit function" concept is gone, I didn't know that" - It's gone since C99 (not just C11). Any C99 compiler is *required* to issue a diagnostic if `#include <stdlib.h>` isn't included.Out of the 4 bullets, the first 3 are subjective (there's sufficient proof that some tend to favor it in this page) and the 4th has been "fixed" in C language about 10 years before this answer. While I am not advocating in favor of the cast, it's really a case of *making a mountain out of a mole* here. – P.P Jan 18, 2017 at 12:50

484

In C, you don't need to cast the return value of `malloc`. The pointer to void returned by `malloc` is automagically converted to the correct type. However, if you want your code to compile with a C++ compiler, a cast is needed. A preferred alternative among the community is to use the following:

```
int *sieve = malloc(sizeof *sieve * length);
```

which additionally frees you from having to worry about changing the right-hand side of the expression if ever you change the type of `sieve`.

Casts are bad, as people have pointed out. Especially pointer casts.

Share

Improve this answer

Follow

edited May 23, 2020 at 20:10

royhowie

11.2k ● 14 ● 53 ● 67

answered Mar 3, 2009 at 10:17

dirkgently

111k ● 16 ● 134 ● 190

134 @MAKZ I'd argue that `malloc(length * sizeof *sieve)` makes it look like `sizeof` is a variable - so I think `malloc(length * sizeof(*sieve))` is more readable. – Michael Anderson Apr 30, 2015 at 7:02

36 And `malloc(length * (sizeof *sieve))` more readable still. IMHO. – Toby Speight Aug 20, 2015 at 13:01

25 @Michael Anderson `()` issue aside, note that your suggested style switched the order., Consider when element count is computed like `length*width`, keeping the `sizeof` first in this case insures multiplication is done with at least `size_t` math. Compare `malloc(sizeof( *ptr) * length * width)` vs. `malloc(length * width * sizeof (*ptr))` - the 2nd may overflow the `length*width` when `width,length` are smaller types that `size_t`. – chux Dec 10, 2015 at 16:40

3 @chux it's not obvious, but the answer has been edited so that my comment is less pertinent - the original suggestion was `malloc(sizeof *sieve * length)` – Michael Anderson Dec 11, 2015 at 0:18

20 C is not C++. Pretending that they are will ultimately lead to confusion and sadness. If you're using C++, then a C-style cast is also bad (unless you're using a very old C++ compiler). And `static_cast>()` (or `reinterpret_cast<>()`)is not compatible with any dialect of C. – David C. Feb 12, 2016 at 23:12

412

You **do** cast, because:

- It makes your code **more portable** between C and C++, and as SO experience shows, a great many programmers claim they are writing in C when they are really writing in C++ (or C plus local compiler extensions).

- Failing to do so **can hide an error**: note all the SO examples of confusing when to write `type *` versus `type **`.
- The idea that it keeps you from noticing you failed to `#include` an appropriate header file misses **the forest for the trees**. It's the same as saying "don't worry about the fact you failed to ask the compiler to complain about not seeing prototypes -- that pesky stdlib.h is the REAL important thing to remember!"
- It forces an **extra cognitive cross-check**. It puts the (alleged) desired type right next to the arithmetic you're doing for the raw size of that variable. I bet you could do an SO study that shows that `malloc()` bugs are caught much faster when there's a cast. As with assertions, annotations that reveal intent decrease bugs.
- Repeating yourself in a way that the machine can check is often a **great** idea. In fact, that's what an assertion is, and this use of cast is an assertion. Assertions are still the most general technique we have for getting code correct, since Turing came up with the idea so many years ago.

edited Nov 2, 2016 at 14:54

answered Feb 14, 2013 at 16:15

Ron Burk
6,231 ● 1 ● 20 ● 20

42 @ulidtko In case you did not know, it's possible to write code which compiles both as C and as C++. In fact most header files are like this, and they often contain code (macros and inline functions). Having a `.c` / `.cpp` file to compile as both is not useful very often, but one case is adding C++ `throw` support when compiled with C++ compiler (but `return -1;` when compiled with C compiler, or whatever). – hyde Mar 26, 2013 at 11:09

43 If someone had malloc calls inline in a header I wouldn't be impressed, #ifdef __cplusplus and extern "C" {} are for this job, not adding in extra casts. – paulm May 6, 2013 at 17:55

22 Well, point 1 is irrelevant, since C != C++, the other points are also trivial, if you use *the variable* in your `malloc` call: `char **foo = malloc(3*sizeof(*foo));` if quite full-proof: 3 pointers to char pointers. then loop, and do `foo[i] = calloc(101, sizeof(*(foo[i])));`. Allocate array of 101 chars, neatly initialized to zeroes. No cast needed. change the declaration to `unsigned char` or any other type, for that matter, and you're still good – Elias Van Ootegem Dec 23, 2013 at 15:37

42 When I tought I got it, there it comes! Fantastic answer. Its the first time here in StackOverflow that I +1 two opposite answers! +1 No, you dont cast, and +1 Yes, you do cast! LOL. You guys are terrific. And for me and my students, I made my mind: I do cast. The kind of errors students make are more easily spotted when casting. – DrBeco Sep 26, 2014 at 3:22

21 @Leushenko: Repeating yourself in a way that cannot be validated by machine nor by local inspection is bad. Repeating yourself in ways that can be validated by such means is less bad. Given `struct Zebra *p; ... p=malloc(sizeof struct Zebra);`, the malloc can't avoid duplciating information about p's type, but neither the compiler nor local code inspection would detect any problem if one type changed but the other didn't. Change the code to `p=(struct Zebra*)malloc(sizeof struct Zebra);` and the compiler will squawk if the cast type doesn't match `p`, and *local* inspection will reveal... – supercat Sep 18, 2015 at 17:46

199

As others stated, it is not needed for C, but necessary for C++. If you think you are going to compile your C code with a C++ compiler, for whatever reasons, you can use a macro instead, like:

```
#ifdef __cplusplus
# define MALLOC(type) ((type *)malloc(sizeof(type)))
# define CALLOC(count, type) ((type *)calloc(count, sizeof(type)))
#else
# define MALLOC(type) (malloc(sizeof(type)))
# define CALLOC(count, type) (calloc(count, sizeof(type)))
#endif
# define FREE(pointer) free(pointer)
```

That way you can still write it in a very compact way:

```
int *sieve = MALLOC(int); // allocate single int => compare to stack int sieve
= ???;
int *sieve_arr = CALLOC(4, int); // allocate 4 times size of int => compare to
stack (int sieve_arr[4] = {0, 0, 0, 0};
// do something with the ptr or the value
```

```
FREE(sieve);
FREE(sieve_arr);
```

and it will compile for C and C++.

Share

Improve this answer

Follow

edited Jan 30, 2023 at 10:32

Sinipelto

326 ● 3 ● 11

answered Mar 3, 2009 at 11:17

quinmars

11.6k ● 8 ● 34 ● 41

22 Since you're using a macro anyway, why don't you use `new` in the definition of C++? – Hosam Aly Mar 4, 2009 at 6:13

78 Because there is no reason to do so. It is mainly for C programs that are compiled with a C++ compiler. If you are going to use 'new', the only thing you get are problems. You need then also a macro for free. And you need a macro to free an array, a differentiation that doesn't exists in C. – quinmars Mar 4, 2009 at 8:51

9 Not to mention if it's not you who frees the memory but maybe a C library you are using, etc. Many possible problems without any gain. – quinmars Mar 4, 2009 at 8:53

96 @Hosam: Yes, it definitely is. If you use `new` you must use `delete` and if you use `malloc()` you must you `free()`. Never mix them. – Graeme Perrow Jul 16, 2011 at 17:10

25 If one is going to take this approach, calling the macro `NEW` is probably a bad idea since the resource is never returned using `delete` (or `DELETE`) so you're mixing your vocabulary. Instead, naming it `MALLOC`, or rather `CALLOC` in this case, would make more sense. – mah Apr 16, 2014 at 15:23

174

+50

From the Wikipedia:

> **Advantages to casting**
>
> - Including the cast may allow a C program or function to compile as C++.
> - The cast allows for pre-1989 versions of malloc that originally returned a char *.
> - Casting can help the developer identify inconsistencies in type sizing should the destination pointer type change, particularly if the pointer is declared far from the malloc() call (although modern compilers and static analyzers can warn on such behaviour without requiring the cast).
>
> **Disadvantages to casting**
>
> - Under the ANSI C standard, the cast is redundant.
> - Adding the cast may mask failure to include the header *stdlib.h*, in which the prototype for malloc is found. In the absence of a prototype for

> malloc, the standard requires that the C compiler assume malloc returns an int. If there is no cast, a warning is issued when this integer is assigned to the pointer; however, with the cast, this warning is not produced, hiding a bug. On certain architectures and data models (such as LP64 on 64-bit systems, where long and pointers are 64-bit and int is 32-bit), this error can actually result in undefined behaviour, as the implicitly declared malloc returns a 32-bit value whereas the actually defined function returns a 64-bit value. Depending on calling conventions and memory layout, this may result in stack smashing. This issue is less likely to go unnoticed in modern compilers, as they uniformly produce warnings that an undeclared function has been used, so a warning will still appear. For example, GCC's default behaviour is to show a warning that reads "incompatible implicit declaration of built-in function" regardless of whether the cast is present or not.
>
> - If the type of the pointer is changed at its declaration, one may also, need to change all lines where malloc is called and cast.

Although **malloc without casting is preferred method and most experienced programmers choose it**, you should use whichever you like having aware of the issues.

i.e: If you need to compile C program as C++ (Although it is a separate language) you must cast the result of use `malloc`.

Share

Improve this answer

Follow

edited Sep 16, 2019 at 10:34

klutt

31.2k ● 19 ● 61 ● 105

answered Oct 9, 2015 at 21:23

ashiquzzaman33

5,741 ● 5 ● 33 ● 45

3 What does "*Casting can help the developer identify inconsistencies in type sizing should the destination pointer type change, particularly if the pointer is declared far from the* `malloc()` *call*" mean? Could you give an example? – Spikatrix May 14, 2016 at 6:26

5 @CoolGuy: See an earlier comment on another answer. But note that the `p = malloc(sizeof(*p) * count)` idiom picks up changes in the type automatically, so you don't have to get warnings and go change anything. So this isn't a real advantage vs. the best alternative for not-casting. – Peter Cordes Oct 24, 2016 at 15:02

11 This is the proper answer: There are pros and cons, and it boils down to a matter of taste (unless the code must compile as C++ -- then the cast is mandatory).
– Peter - Reinstate Monica Nov 15, 2016 at 17:00

4 Point 3 is moot, since If the type of the pointer is changed at its declaration, one should check every instance of malloc, realloc and free inolving that type. Casting will force you to do just that. – Michaël Roy Jun 18, 2017 at 1:46

1 If one forgets to include stdlib.h, and the program compiles, how does it link without a definition for malloc? If it links and runs anyway, which instructions actually get run on that line

for any given CPU? I guess I should check godbolt... – SO_fix_the_vote_sorting_bug Aug 6, 2021 at 18:04

117

In C you can implicitly convert a `void` pointer to any other kind of pointer, so a cast is not necessary. Using one may suggest to the casual observer that there is some reason why one is needed, which may be misleading.

Share

Improve this answer

Follow

edited May 19, 2020 at 10:36

chqrlie

144k ● 12 ● 130 ● 207

answered Mar 3, 2009 at 10:18

pauljwilliams

19.2k ● 3 ● 53 ● 79

Without the explicitly annotated type in the declarator, the cast is usually more common misleading by easily confusing between the intent to allocate an object (uninterested the initial value) and to allocate merely some (uninitialized) memory. – FrankHB May 13, 2022 at 9:02

117

You don't cast the result of `malloc`, because doing so adds pointless clutter to your code.

The most common reason why people cast the result of `malloc` is because they are unsure about how the C language works. That's a warning sign: if you don't know how a particular language mechanism works, then *don't* take a guess. Look it up or ask on Stack Overflow.

Some comments:

- A void pointer can be converted to/from any other pointer type without an explicit cast (C11 6.3.2.3 and 6.5.16.1).
- C++ will however not allow an implicit cast between `void*` and another pointer type. So in C++, the cast would have been correct. But if you program in C++, you should use `new` and not `malloc()`. And you should never compile C code using a C++ compiler.

  If you need to support both C and C++ with the same source code, use compiler switches to mark the differences. Do not attempt to sate both language standards with the same code, because they are not compatible.
- If a C compiler cannot find a function because you forgot to include the header, you will get a compiler/linker error about that. So if you forgot to include `<stdlib.h>` that's no biggie, you won't be able to build your program.
- On ancient compilers that follow a version of the standard which is more than 25 years old, forgetting to include `<stdlib.h>` would result in dangerous behavior.

Because in that ancient standard, functions without a visible prototype implicitly converted the return type to `int`. Casting the result from `malloc` explicitly would then hide away this bug.

But that is really a non-issue. You aren't using a 25 years old computer, so why would you use a 25 years old compiler?

Share

Improve this answer

Follow

edited Sep 29, 2020 at 6:17

Vishnu CS

851 ● 1 ● 11 ● 25

answered Mar 20, 2014 at 15:53

Lundin

212k ● 45 ● 270 ● 426

14 "pointless clutter" is dismissive hyperbole that tends to derail any possibility of convincing anyone who doesn't already agree with you. A cast certainly isn't pointless; Ron Burk's and Kaz's answers make arguments in favor of casting that I very much agree with. Whether those concerns weigh more than the concerns you mention is a reasonable question to ask. To me, your concerns look relatively minor compared to theirs. – Don Hatch Nov 2, 2016 at 8:43

1 "A void pointer can be converted to/from any other pointer type without an explicit cast" is not supported by 6.3.2.3. Perhaps you are thinking of "pointer to any object type"? "void pointer" and "pointer to a function" are not so readily convertible. – chux Jul 27, 2017 at 21:07

Indeed the reference was incomplete. The relevant part for the "implicitness" is the rule of simple assignment 6.5.16.1. "one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of void". I've added this reference to the answer for completeness. – Lundin Apr 12, 2018 at 8:55

As I've commented to another answer, the reason is to prevent some confusion by abusing the C language rules. Also you cannot resolve the confusion without the complete rules of object creation of C (including how *effective type* works), so the reasoning is quite incomplete. OTOH, the similar effects in the source code can be achieved in C++ by a class type with member `template<T> operator T*() const {...}` to replace `void*` here, but C++ still incurs no such confusion due to the stricter rules. The answer is also incomplete in this sense. – FrankHB May 13, 2022 at 8:15

@FrankHB What do you mean? The effective type isn't affected by the cast. The returned object does not get an effective type until you write to it ("lvalue access"). – Lundin May 13, 2022 at 8:22

In C you get an implicit conversion from `void *` to any other (data) pointer.

107

Share

Improve this answer

Follow

edited May 19, 2020 at 10:37

chqrlie

144k ● 12 ● 130 ● 207

answered Mar 3, 2009 at 10:16

EFraim

13k ● 4 ● 47 ● 64

3 …and therefore? – dumbass Nov 20, 2023 at 12:43

1 That information is already included in unwind's answer. Sure, it was posted 1 minute after yours, but gave a clearer answer with a higher score, making your answer *void* of extra useful information. – Cœur Nov 20, 2023 at 17:10

@Cœur This is still an answer. Not answer that deserves to have 112 upvotes, but an answer nevertheless. – CPlus Nov 21, 2023 at 3:54

This is not an answer to the question that was posted by the asker, and doesn't even *imply* an answer to the question posted by the asker. In short, not an answer at all. – dumbass Jan 1 at 10:10

1 "Should I cast the result of malloc?" "In C you get an implicit conversion from `void *` to any other (data) pointer." So? Yes or no? Is that a good or a bad thing? What is the conclusion? The post doesn't say, it's not an answer. – dumbass Jan 17 at 13:50

78

Casting the value returned by `malloc()` is not necessary now, but I'd like to add one point that seems no one has pointed out:

In the ancient days, that is, before **ANSI C** provides the `void *` as the generic type of pointers, `char *` is the type for such usage. In that case, the cast can shut down the compiler warnings.

Reference: C FAQ

Share

Improve this answer

Follow

edited Feb 14, 2014 at 2:28

answered Jun 9, 2013 at 17:31

Yu Hao

122k ● 48 ● 246 ● 302

6 Shutting up compiler warnings is a bad idea. – Albert van der Horst Apr 1, 2016 at 17:33

13 @AlbertvanderHorst Not if you're doing so by solving the exact problem the warning is there to warn you of. – Dan Bechard Apr 11, 2016 at 17:17

1 @Dan . If by solving the exact problem is meant a rewrite of a subroutine to return modern ANSI C types instead of char *, I agree. I wouldn't call that shutting up the compiler. Do not give in to managers who insists that there are no compiler warnings , instead of using them by each recompilation to find possible problems. Groetjes Albert – Albert van der Horst Nov 2, 2016 at 16:51

How many people use pre-ANSI C compilers that don't provide `void *`? – dumbass Sep 9, 2023 at 14:01

Adding `-fpermissive -w` to your CXXFLAGS would be the "shutting up the compiler warning" solution. Fixing the problem by adding the requested explicit cast is not. – Matthew Cole May 13 at 22:04

62

Just adding my experience, studying computer engineering I see that the two or three professors that I have seen writing in C always cast malloc, however the one I asked (with an immense CV and understanding of C) told me that it is absolutely unnecessary but only used to be absolutely specific, and to get the students into the mentality of being absolutely specific. Essentially casting will not change anything in how it works, it does exactly what it says, allocates memory, and casting does not effect it, you get the same memory, and even if you cast it to something else by mistake (and somehow evade compiler errors) C will access it the same way.

Edit: Casting has a certain point. When you use array notation, the code generated has to know how many memory places it has to advance to reach the beginning of the next element, this is achieved through casting. This way you know that for a double you go 8 bytes ahead while for an int you go 4, and so on. Thus it has no effect if you use pointer notation, in array notation it becomes necessary.

Share

Improve this answer

Follow

edited Nov 23, 2014 at 23:37

answered Mar 28, 2014 at 18:21

user3079666

1,165 ● 10 ● 24

4 Except as already mentioned, the cast might hide bugs and make the code harder to analyse for the compiler or static analyser. – Lundin May 27, 2014 at 7:56

3 "Essentially casting will not change anything in how it works". Casting to the matching type should not change anything, but should the var's type change and the cast no longer match, could problems come up? IWOs, the cast and var type should be kept in sync - twice the maintenance work. – chux Aug 23, 2014 at 19:29

4 I can see why Profs prefer casting. Casting may be useful from an educational standpoint where it conveys to the instructor information and the student code does not need to be maintained - its throw-away code. Yet from a coding, peer-review and **maintenance** perspective, `p = malloc(sizeof *p * n);` is so simple and better. – chux Jul 27, 2017 at 21:38

59

It is not mandatory to cast the results of `malloc`, since it returns `void*` , and a `void*` can be pointed to any datatype.

Share

Improve this answer

Follow

edited Jun 3, 2015 at 6:10

WedaPashi

3,872 ● 1 ● 29 ● 44

answered Feb 7, 2013 at 22:22

user968000

1,843 ● 3 ● 24 ● 34

5 It's not the fact that `void*` can point to anything that enables this; it's the fact that a `void*` can be *implicitly converted* to any other pointer type. To clarify the distinction, in C++ a

`void*` can still point to anything, but implicit conversion was removed, so one must cast. – underscore_d Jul 1, 2020 at 8:38

43

This is what The GNU C Library Reference manual says:

> You can store the result of `malloc` into any pointer variable without a cast, because ISO C automatically converts the type `void *` to another type of pointer when necessary. But the cast is necessary in contexts other than assignment operators or if you might want your code to run in traditional C.

And indeed the ISO C11 standard (p347) says so:

> The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated)

Share
Improve this answer
Follow

edited Sep 5, 2018 at 19:48

answered Oct 9, 2015 at 17:47

Slothworks
1,081 ● 14 ● 18

40

A void pointer is a generic object pointer and C supports implicit conversion from a void pointer type to other types, so there is no need of explicitly typecasting it.

However, if you want the same code work perfectly compatible on a C++ platform, which does not support implicit conversion, you need to do the typecasting, so it all depends on usability.

Share
Improve this answer
Follow

edited Oct 24, 2019 at 13:04

answered Jul 13, 2014 at 6:42

Endeavour
555 ● 4 ● 7

3 It's not a normal use case to compile a single source as both C and C++ (as opposed, say, to using a header file containing declarations to link C and C++ code together). Using `malloc` and friends in C++ is a good warning sign that it deserves special attention (or re-writing in C). – Toby Speight Aug 20, 2015 at 12:57

It depends on the programming language and compiler. If you use `malloc` in C, there is no need to type cast it, as it will automatically type cast. However, if you are using C++, then you should type cast because `malloc` will return a `void*` type.

34

Share

Improve this answer

Follow

edited May 19, 2020 at 4:38

NAND

685 ● 1 ● 10 ● 22

answered Mar 17, 2014 at 13:16

Jeyamaran

783 ● 2 ● 9 ● 16

4 The function *malloc* returns a void pointer in C as well but the rules of the language are different from C++. – August Karlstrom Sep 5, 2015 at 9:19

32

The returned type is void*, which can be cast to the desired type of data pointer in order to be dereferenceable.

Share

Improve this answer

Follow

edited Oct 18, 2014 at 21:33

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Apr 26, 2013 at 16:43

swaps

596 ● 5 ● 4

3 `void*` *can* be cast to the desired type, but there is no need to do so as it will be automatically converted. So the cast is not necessary, and in fact undesirable for the reasons mentioned in the high-scoring answers. – Toby Speight Aug 20, 2015 at 12:59

1 but only if you need to dereference it "on the fly", if you create a variable instead it will be converted safely and automatically into the variable's effective type, without casting (in C). – Ferrarezi Jan 8, 2020 at 15:04

1 How does that in any way answer the question that was asked? – dumbass Sep 6, 2023 at 15:22

32

In the C language, a void pointer can be assigned to any pointer, which is why you should not use a type cast. If you want "type safe" allocation, I can recommend the following macro functions, which I always use in my C projects:

```
#include <stdlib.h>
#define NEW_ARRAY(ptr, n) (ptr) = malloc((n) * sizeof *(ptr))
#define NEW(ptr) NEW_ARRAY((ptr), 1)
```

With these in place you can simply say

```
NEW_ARRAY(sieve, length);
```

For non-dynamic arrays, the third must-have function macro is

```
#define LEN(arr) (sizeof (arr) / sizeof (arr)[0])
```

which makes array loops safer and more convenient:

```
int i, a[100];

for (i = 0; i < LEN(a); i++) {
   ...
}
```

Share

Improve this answer

Follow

edited Sep 8, 2015 at 8:36

answered Sep 4, 2015 at 11:52

August Karlstrom

11.4k ● 8 ● 44 ● 69

2 "a void pointer can be assigned to any *object* pointer" Function pointers are another issue, albeit not a `malloc()` one. – chux Jul 27, 2017 at 21:39

1 Assigning a `void*` to/from a function pointer may lose information so "a void pointer can be assigned to any pointer," is a problem in those cases. Assigning a `void*`, from `malloc()` to any *object* pointer is not an issue though. – chux Dec 26, 2017 at 19:09

27

This question is subject of opinion-based abuse.

Sometimes I notice comments like that:

> Don't cast the result of malloc

or

> Why you don't cast the result of malloc

on questions where OP uses casting. The comments itself contain a hyperlink to this question.

That is in *any* possible manner inappropriate and incorrect as well. There is no right and no wrong when it is truly a matter of one's own coding-style.

Why is this happening?

It's based upon two reasons:

1. This question is indeed opinion-based. Technically, the question should have been closed as opinion-based years ago. A "*Do I*" or "*Don't I*" or equivalent "*Should I*" or "*Shouldn't I*" question, you just can't answer focused without an attitude of one's own opinion. One of the reason to close a question is because it "might lead to opinion-based answers" as it is well shown here.
2. Many answers (including the most apparent and accepted answer of @unwind) are either completely or almost entirely opinion-based (f.e. a mysterious "clutter" that would be added to your code if you do casting or repeating yourself would be bad) and show a clear and focused tendency to omit the cast. They argue about the redundancy of the cast on one side but also and even worse argue to solve a bug caused by a bug/failure of programming itself - to not `#include <stdlib.h>` if one want to use `malloc()`.

I want to bring a true view of some points discussed, with less of my personal opinion. A few points need to be noted especially:

1. Such a very susceptible question to fall into one's own opinion needs an answer with neutral pros and cons. Not only cons or pros.

   A good overview of pros and cons is listed in this answer:

   https://stackoverflow.com/a/33047365/12139179

   (I personally consider this because of that reason the best answer, so far.)

---

2. One reason which is encountered at most to reason the omission of the cast is that the cast might hide a bug.

   If someone uses an implicit declared `malloc()` that returns `int` (implicit functions are gone from the standard since C99) and `sizeof(int) != sizeof(int*)`, as shown in this question

   Why does this code segfault on 64-bit architecture but work fine on 32-bit?

   the cast would hide a bug.

   While this is true, it only shows half of the story as the omission of the cast would only be a forward-bringing solution to an even bigger bug - not including `stdlib.h` when using `malloc()`.

   This will never be a serious issue, If you,

   1. Use a compiler compliant to C99 or above (which is recommended and should be mandatory), and

2. Aren't so absent to forgot to include `stdlib.h`, when you want to use `malloc()` in your code, which is a huge bug itself.

---

3. Some people argue about C++ compliance of C code, as the cast is obliged in C++.

   First of all to say in general: Compiling C code with a C++ compiler is not a good practice.

   C and C++ are in fact two completely different languages with different semantics.

   But If you really want/need to make C code compliant to C++ and vice versa use compiler switches instead of any cast.

   Since the cast is with tendency declared as redundant or even harmful, I want to take a focus on these questions, which give good reasons why casting can be useful or even necessary:

   - https://stackoverflow.com/a/34094068/12139179
   - https://stackoverflow.com/a/36297486/12139179
   - https://stackoverflow.com/a/33044300/12139179

---

4. The cast can be non-beneficial when your code, respectively the type of the assigned pointer (and with that the type of the cast), changes, although this is in most cases unlikely. Then you would need to maintain/change all casts too and if you have a few thousand calls to memory-management functions in your code, this can really summarizing up and decrease the maintenance efficiency.

Summary:

Fact is, that the cast is redundant per the C standard (already since ANSI-C (C89/C90)) if the assigned pointer point to an object of fundamental alignment requirement (which includes the most of all objects).

You don't need to do the cast as the pointer is automatically aligned in this case:

> "The order and contiguity of storage allocated by successive calls to the aligned_alloc, calloc, malloc, and realloc functions is unspecified. **The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement** and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated)."

> Source: C18, §7.22.3/1

> "A *fundamental alignment* is a valid alignment less than or equal to `_Alignof (max_align_t)`. Fundamental alignments shall be supported by the implementation for objects of all storage durations. The alignment requirements of the following types shall be fundamental alignments:
>
> — all atomic, qualified, or unqualified basic types;
>
> — all atomic, qualified, or unqualified enumerated types;
>
> — all atomic, qualified, or unqualified pointer types;
>
> — all array types whose element type has a fundamental alignment requirement;57)
>
> — all types specified in Clause 7 as complete object types;
>
> — all structure or union types all of whose elements have types with fundamental alignment requirements and none of whose elements have an alignment specifier specifying an alignment that is not a fundamental alignment.
>
> 57. As specified in 6.2.1, the later declaration might hide the prior declaration."
>
> Source: C18, §6.2.8/2

However, if you allocate memory for an implementation-defined object of extended alignment requirement, the cast would be needed.

> An *extended alignment* is represented by an alignment greater than `_Alignof (max_align_t)`. It is implementation-defined whether any extended alignments are supported and the storage durations for which they are supported. A type having an extended alignment requirement is an over-aligned type.58)
>
> Source. C18, §6.2.8/3

Everything else is a matter of the specific use case and one's own opinion.

Please be careful how you educate yourself.

I recommend you to read *all* of the answers made so far carefully first (as well as their comments which may point at a failure) and then build your own opinion if you or if you not cast the result of `malloc()` at a specific case.

Please note:

There is no right and wrong answer to that question. It is a matter of style and you yourself decide which way you choose (if you aren't forced to by education or job of course). Please be aware of that and don't let trick you.

Last note: I voted to lately close this question as opinion-based, which is indeed needed since years. If you got the close/reopen privilege I would like to invite you to do so, too.

Share

Improve this answer

Follow

edited Jul 4, 2020 at 16:19

answered Jun 12, 2020 at 19:20

RobertS supports Monica Cellio

15k ● 12 ● 41 ● 91

2 This is pretty much the same thing as said in this old answer though: stackoverflow.com/a/22538350/584518. – Lundin Jun 18, 2020 at 14:03

1 @Lundin You must have pasted the wrong link, this answer is completely unrelated to this one afaics – Ctx Jun 18, 2020 at 14:15

1 @RobertSsupportsMonicaCellio What would you call the cast in this expression: `int x, y; x = (int)y;` then? A matter of subjective coding style? – Lundin Jun 18, 2020 at 14:49

5 This is a bad answer because it relies on the implicit claim that all arguments in this debate are of equal value, whereas this is obviously not the case. The arguments *in favour* of the cast — with one niche exception (adherence to external code style requirements) — are simply bad arguments, for various reasons (from subjective to factually wrong). It's fallacious to conclude that, just because sides have nominal "arguments", the decision is therefore a toss-up, or opinion-based. In the same vein you'd both-side the non-scientific debates about biological evolution or global warming. – Konrad Rudolph Jul 29, 2020 at 15:48

4 @KonradRudolph I don't see how opinion-based arguments in favor of the omission of the cast would have more value than arguments about that it is being allowed and can be used and I also do not understand why *all* of the given arguments for casting are "*bad*" arguments. The classification as "*bad*" is also subjective and opinion-based and what I wanted to prevent with this answer with just plain facts. – RobertS supports Monica Cellio Aug 3, 2020 at 15:40

23

People used to GCC and Clang are spoiled. It's not all that good out there.

I have been pretty horrified over the years by the staggeringly aged compilers I've been required to use. Often companies and managers adopt an ultra-conservative approach to changing compilers and will not even *test* if a new compiler (with better standards compliance and code optimization) will work in their system. The practical reality for working developers is that when you're coding you need to cover your bases and, unfortunately, casting mallocs is a good habit if you cannot control what compiler may be applied to your code.

I would also suggest that many organizations apply a coding standard of their own and that *that* should be the method people follow if it is defined. In the absence of explicit guidance I tend to go for most likely to compile everywhere, rather than slavish adherence to a standard.

The argument that it's not necessary under current standards is quite valid. But that argument omits the practicalities of the real world. We do not code in a world ruled exclusively by the standard of the day, but by the practicalities of what I like to call "local management's reality field". And that's bent and twisted more than space time ever was. :-)

YMMV.

I tend to think of casting malloc as a defensive operation. Not pretty, not perfect, but generally safe. (Honestly, if you've not included stdlib.h then you've *way* more problems than casting malloc !).

Share Improve this answer Follow

answered Dec 4, 2015 at 17:27

StephenG - Help Ukraine

965 ● 6 ● 13

22

No, you don't cast the result of `malloc()`.

In general, you **don't cast to or from** `void *`.

A typical reason given for not doing so is that failure to `#include <stdlib.h>` could go unnoticed. This isn't an issue anymore for a long time now as C99 made *implicit function declarations* illegal, so if your compiler conforms to at least C99, you will get a diagnostic message.

But there's a **much stronger reason** not to introduce unnecessary pointer casts:

In C, a **pointer cast is almost always an error**. This is because of the following rule (**§6.5 p7** in N1570, the latest draft for C11):

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
> — a type compatible with the effective type of the object,
> — a qualified version of a type compatible with the effective type of the object,
> — a type that is the signed or unsigned type corresponding to the effective type of the object,
> — a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
> — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
> — a character type.

This is also known as the *strict aliasing rule*. So the following code is *undefined behavior*:

```
long x = 5;
double *p = (double *)&x;
double y = *p;
```

And, sometimes surprisingly, the following is as well:

```
struct foo { int x; };
struct bar { int x; int y; };
struct bar b = { 1, 2};
struct foo *p = (struct foo *)&b;
int z = p->x;
```

Sometimes, you **do** need to cast pointers, but given the *strict aliasing rule*, you have to be very careful with it. So, any occurrence of a pointer cast in your code is a place you **have to double-check for its validity**. Therefore, you never write an unnecessary pointer cast.

tl;dr

In a nutshell: Because in C, **any** occurrence of a *pointer cast* should raise a red flag for code requiring special attention, you should never write *unnecessary* pointer casts.

Side notes:

- There are cases where you actually *need* a cast to `void *`, e.g. if you want to print a pointer:

  ```
  int x = 5;
  printf("%p\n", (void *)&x);
  ```

  The cast is necessary here, because `printf()` is a variadic function, so implicit conversions don't work.

- In C++, the situation is different. Casting pointer types is somewhat common (and correct) when dealing with objects of derived classes. Therefore, it makes sense that in C++, the conversion to and from `void *` is **not** implicit. C++ has a whole set of different flavors of casting.

Share

Improve this answer

Follow

edited Nov 8, 2017 at 8:54

answered Aug 16, 2017 at 12:25

user2371524

1 In your examples you avoid void *. there is a difference between cast from double * to int * and vice versa. malloc returns pointel aligned to the largest standard type so there are no aliasing rules broken even if someone casts form this aligned pointer to other type. – 0_____ Aug 22, 2017 at 16:38

Aliasing has **nothing** at all to do with alignment and for the rest of your comment -- you obviously didn't get the point. – user2371524 Aug 22, 2017 at 16:43

@PeterJ: just in case, the point is to *avoid* an unnecessary pointer cast, so it doesn't *look like* a piece of code you have to pay special attention to. – user2371524 Aug 22, 2017 at 16:48

The strict aliasing issue doesn't really have anything to do with void pointers. In order to get bugs caused by strict aliasing violations, you must de-reference the pointed-at data. And since you can't de-reference a void pointer, such bugs are per definition not related to the void pointer but to something else. – Lundin Aug 23, 2017 at 6:33

Rather, you would have to make a rule to ban all pointer casts. But then how would you write things like serialization routines and hardware-related programming? Things that are C's strength. Such casts are fine if you know what you are doing. – Lundin Aug 23, 2017 at 6:36

18

I put in the cast simply to show disapproval of the ugly hole in the type system, which allows code such as the following snippet to compile without diagnostics, even though no casts are used to bring about the bad conversion:

```
double d;
void *p = &d;
int *q = p;
```

I wish that didn't exist (and it doesn't in C++) and so I cast. It represents my taste, and my programming politics. I'm not only casting a pointer, but effectively, casting a ballot, and casting out demons of stupidity. If I can't *actually* cast out stupidity, then at least let me express the wish to do so with a gesture of protest.

In fact, a good practice is to wrap `malloc` (and friends) with functions that return `unsigned char *`, and basically never to use `void *` in your code. If you need a generic pointer-to-any-object, use a `char *` or `unsigned char *`, and have casts in both directions. The one relaxation that can be indulged, perhaps, is using functions like `memset` and `memcpy` without casts.

On the topic of casting and C++ compatibility, if you write your code so that it compiles as both C and C++ (in which case you *have to* cast the return value of `malloc` when assigning it to something other than `void *`), you can do a very helpful thing for yourself: you can use macros for casting which translate to C++ style casts when compiling as C++, but reduce to a C cast when compiling as C:

```
/* In a header somewhere */
#ifdef __cplusplus
#define strip_qual(TYPE, EXPR) (const_cast<TYPE>(EXPR))
#define convert(TYPE, EXPR) (static_cast<TYPE>(EXPR))
#define coerce(TYPE, EXPR) (reinterpret_cast<TYPE>(EXPR))
#else
#define strip_qual(TYPE, EXPR) ((TYPE) (EXPR))
#define convert(TYPE, EXPR) ((TYPE) (EXPR))
#define coerce(TYPE, EXPR) ((TYPE) (EXPR))
#endif
```

If you adhere to these macros, then a simple `grep` search of your code base for these identifiers will show you where all your casts are, so you can review whether any of them are incorrect.

Then, going forward, if you regularly compile the code with C++, it will enforce the use of an appropriate cast. For instance, if you use `strip_qual` just to remove a `const` or `volatile`, but the program changes in such a way that a type conversion is now involved, you will get a diagnostic, and you will have to use a combination of casts to get the desired conversion.

To help you adhere to these macros, the the GNU C++ (not C!) compiler has a beautiful feature: an optional diagnostic which is produced for all occurrences of C style casts.

```
     -Wold-style-cast (C++ and Objective-C++ only)
         Warn if an old-style (C-style) cast to a non-void type is used
         within a C++ program.  The new-style casts (dynamic_cast,
         static_cast, reinterpret_cast, and const_cast) are less vulnerable
         to unintended effects and much easier to search for.
```

If your C code compiles as C++, you can use this `-Wold-style-cast` option to find out all occurrences of the `(type)` casting syntax that may creep into the code, and follow up on these diagnostics by replacing it with an appropriate choice from among the above macros (or a combination, if necessary).

This treatment of conversions is the single largest standalone technical justification for working in a "Clean C": the combined C and C++ dialect, which in turn technically justifies casting the return value of `malloc`.

Share Improve this answer Follow

answered Mar 30, 2016 at 0:23

Kaz
58.3k ● 10 ● 111 ● 161

As other pointed out, I would usually recommend to not mix C and C++ code. However, if you have good reason to do it, then macros might be useful. – Phil1970 Jun 10, 2016 at 15:53

@Phil1970 It's all written in one cohesive dialect, which happens to be portable to C and C++ compilers, and takes advantage of some capabilities of C++. It must be all compiled as C++, or else all compiled as C. – Kaz Jun 10, 2016 at 16:45

I.e. what I was trying to say in the previous comment is that there is no mixing of C and C++. The intent is that the code is all compiled as C or all compiled as C++. – Kaz Jun 14, 2016 at 17:35

17

I prefer to do the cast, but not manually. My favorite is using `g_new` and `g_new0` macros from glib. If glib is not used, I would add similar macros. Those macros reduce code duplication without compromising type safety. If you get the type wrong, you would get an implicit cast between non-void pointers, which would cause a warning (error in C++). If you forget to include the header that defines `g_new` and `g_new0`, you would get an error. `g_new` and `g_new0` both take the same arguments, unlike `malloc` that takes fewer arguments than `calloc`. Just add `0` to get zero-initialized memory. The code can be compiled with a C++ compiler without changes.

Share
Improve this answer
Follow

edited Dec 26, 2017 at 18:16

Nae
15.3k ● 8 ● 61 ● 85

answered Jun 29, 2016 at 7:30

proski
3,911 ● 31 ● 28

16

The best thing to do when programming in C whenever it is possible:

1. Make your program compile through a C compiler with all warnings turned on `-Wall` and fix all errors and warnings
2. Make sure there are no variables declared as `auto`

3. Then compile it using a C++ compiler with `-Wall` and `-std=c++11`. Fix all errors and warnings.
4. Now compile using the C compiler again. Your program should now compile without any warning and contain fewer bugs.

This procedure lets you take advantage of C++ strict type checking, thus reducing the number of bugs. In particular, this procedure forces you to include `stdlib.h` or you will get

> `malloc` was not declared within this scope

and also forces you to cast the result of `malloc` or you will get

> invalid conversion from `void*` to `T*`

or what ever your target type is.

The only benefits from writing in C instead of C++ I can find are

1. C has a well specified ABI
2. C++ may generate more code [exceptions, RTTI, templates, *runtime* polymorphism]

Notice that the second cons should in the ideal case disappear when using the subset common to C together with the *static* polymorphic feature.

For those that finds C++ strict rules inconvenient, we can use the C++11 feature with inferred type

```
auto memblock=static_cast<T*>(malloc(n*sizeof(T))); //Mult may overflow...
```

Share

Improve this answer

Follow

edited Oct 24, 2016 at 15:44

answered Jun 12, 2015 at 15:23

user877329

6,101 ● 8 ● 51 ● 93

23 Use a C compiler for C code. Use a C++ compiler for C++ code. No ifs, no buts. Rewriting your C code in C++ is another thing entirely, and may - or may not be - worth the time and the risks. – Toby Speight Jun 25, 2015 at 21:06

4 I'd like to add to @TobySpeight advice: If you need to use C code in a C++ project, you can usually compile the C code as C (e.g. `gcc -c c_code.c`), the C++ code as C++ (e.g. `g++ -c cpp_code.cpp`), **and then link them together** (e.g. `gcc c_code.o cpp_code.o` or vice-versa depending upon the project dependencies). Now there should be no reason to deprive yourself of any nice features of either language... – autistic Dec 7, 2015 at 16:06

2 @user877329 It's a more sensible alternative to painstakingly adding casts to code that reduce the code's legibility, only for the sake of being "C++ compatible". – autistic Dec 8, 2015 at 0:54

1 Other advantages to C over C++: C99 variable-length arrays for very efficient allocation/deallocation of scratch space. You can't accidentally write non-constant initializers when you didn't mean to (e.g. if you thought you were being clever by putting `static const __m128 ones = _mm_set1_ps(1.0f);` at the global scope so multiple functions could share a constant, the fact that constructors aren't a thing in C stops you from generating worse code. (This is really finding a silver lining to a C limitation...)) – Peter Cordes Oct 24, 2016 at 15:11

2 Where does this post in any way answer the question that was asked? – dumbass Sep 6, 2023 at 15:20

15

Casting is only for C++ not C.In case you are using a C++ compiler you better change it to C compiler.

Share Improve this answer Follow

answered Sep 10, 2015 at 15:58

user3949394

13

The casting of malloc is unnecessary in C but mandatory in C++.

Casting is unnecessary in C because of:

- `void *` is automatically and safely promoted to any other pointer type in the case of C.
- It can hide an error if you forgot to include `<stdlib.h>`. This can cause crashes.
- If pointers and integers are differently sized, then you're hiding a warning by casting and might lose bits of your returned address.
- If the type of the pointer is changed at its declaration, one may also need to change all lines where `malloc` is called and cast.

On the other hand, casting may increase the portability of your program. i.e, it allows a C program or function to compile as C++.

Share

Improve this answer

Follow

edited Sep 9, 2018 at 14:00

answered Oct 3, 2017 at 13:13

Aashish Kumar

2,869 ● 3 ● 29 ● 44

> In C++, one would normally use `new` rather than `malloc()` so that the object is properly constructed. Actually, in real C++ codebases, we avoid using `new` in favour of owning pointers such as `std::unique_ptr`. Instead of compiling C code as C++, it's much better to compile the C and C++ code separately, and then link them appropriately. – Toby Speight Apr 27 at 6:17

10

A void pointer is a generic pointer and C supports implicit conversion from a void pointer type to other types, so there is no need of explicitly typecasting it.

However, if you want the same code work perfectly compatible on a C++ platform, which does not support implicit conversion, you need to do the typecasting, so it all depends on usability.

Share Improve this answer Follow

answered Jul 18, 2016 at 10:36

dhana govindarajan

226 ● 2 ● 7

10

1. As other stated, it is not needed for C, but for C++.
2. Including the cast may allow a C program or function to compile as C++.
3. In C it is unnecessary, as void * is automatically and safely promoted to any other pointer type.
4. But if you cast then, it can hide an error if you forgot to include **stdlib.h**. This can cause crashes (or, worse, not cause a crash until way later in some totally different part of the code).

   Because **stdlib.h** contains the prototype for malloc is found. In the absence of a prototype for malloc, the standard requires that the C compiler assumes malloc returns an int. If there is no cast, a warning is issued when this integer is assigned to the pointer; however, with the cast, this warning is not produced, hiding a bug.

Share Improve this answer Follow

answered Nov 8, 2016 at 10:09

Mohit

915 ● 9 ● 7

The concept behind void pointer is that it can be casted to any data type that is why malloc returns void. Also you must be aware of automatic typecasting. So it is not

9

mandatory to cast the pointer though you must do it. It helps in keeping the code clean and helps debugging

Share Improve this answer Follow

answered Jul 29, 2015 at 5:53

iec2011007

1,846 ● 4 ● 26 ● 38

15 "*It is not mandatory -- though you must do it*" - I think there's a contradiction there! – Toby Speight Aug 20, 2015 at 12:54

10 I think you should read this post to someone, and see if they understand what you are trying to say. Then rewrite it, making it clear what you want to say. I really can't understand what your answer is. – Bill Woodger Sep 8, 2015 at 12:28

4

The main issue with `malloc` is to get the *right size*.

The memory returned form `malloc()` is *untyped*, and it will not magically gain an *effective type* due to a simple cast.

I guess that both approaches are fine and the choice should depend on programmer intention.

1. If allocating memory for a **type**, then use a cast.

```
ptr = (T*)malloc(sizeof(T));
```

2. If allocating memory for a given pointer, then don't use a cast.

```
ptr = malloc(sizeof *ptr);
```

Ad 1

The first method assures the correct size by allocating memory for a given type, and then casting it to assure that it is assigned to the right pointer. If incorrect type of `ptr` is used then the compiler will issue a warning/error. If the type of `ptr` is changed, then the compiler will point the places where the code needs refactoring.

Moreover, the first method can be combined into a macro similar to `new` operator in C++.

```
#define NEW(T) ((T*)malloc(sizeof(T)))
...
ptr = NEW(T);
```

Moreover this method works if `ptr` is `void*`.

Ad 2

The second methods does not care about the types, it assures the correct size by taking it from the pointer's type. The main advantage of this method is the automatic adjustment of storage size whenever the type of `ptr` is changed. It can save some time (or errors) when refactoring.

The disadvantage is that the method does not work if `ptr` is `void*` but it may be perceived as a good thing. And that it does not work with C++ so it should not be used in inlined functions in headers that are going to be used by C++ programs.

Personally, I prefer the second option.

Share Improve this answer Follow

answered Jan 17, 2022 at 11:11

tstanisl

14.1k ● 2 ● 28 ● 43

0

For me, the take home and conclusion here is that casting `malloc` in C is totally NOT necessary but if you however cast, it wont affect `malloc` as `malloc` will still allocate to you your requested blessed memory space. Another take home is the reason or one of the reasons people do casting and this is to enable them compile same program either in C or C++.

There may be other reasons but other reasons, almost certainly, would land you in serious trouble sooner or later.

Share Improve this answer Follow

answered Mar 5, 2020 at 15:56

pasignature

585 ● 1 ● 8 ● 14

0

No with the exception of when you inline a function in a header. if you do and you intend your code to be used by C++ then you need to cast your mallocs so that the C++ code can compile with a C++ compiler.

but if you are using headers like this you need to start having the C++ includes in your headers for things like assert/cassert and that's just plain annoying.

honestly let some C++ dev fix the code to work in your header if they want to use it, its not that hard to do and they can allways just use -fpermissive

Share Improve this answer Follow

answered Jul 17 at 11:44

Nevo Krien
37 • 4

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.