rust-lang / **rust**

Tracking issue for 128-bit integer support (RFC 1504) #35118

Open **nikomatsakis** opened this issue on 29 Jul 2016 · 119 comments

nikomatsakis commented on 29 Jul 2016 • edited by nagisa — Contributor

Tracking issue for rust-lang/rfcs#1504.

cc **@Amanieu**

Blocking stabilization:

- [x] #41799 (Casting u128::MAX to f32 is undefined)
- [x] ~~Interaction with FFI? (#35118 (comment))~~ - #44261
- [x] ~~separately feature gate `repr(i128)`~~ - #44262
- [x] #45676 (u/i)128 lowering for backends without native support
  - lowering still does not work for emscripten even with the work around
- [ ] Enums with 128-bit discriminant: `repr128` feature

👍 23

nikomatsakis added **T-lang** **B-unstable** **B-RFC-approved** labels on 29 Jul 2016

nikomatsakis referenced this issue in **rust-lang/rfcs** on 29 Jul 2016

Add support for 128-bit integers #1504 — Merged

durka commented on 2 Aug 2016 • edited — Contributor

Is `#[repr(u128)] enum SuchWideVeryDiscriminantWow { ... }` allowed?

👍 12 😄 13

Amanieu commented on 2 Aug 2016 — Contributor

That's a good point, I don't see any reason why it shouldn't be allowed.

durka commented on 2 Aug 2016 — Contributor

The return type of the `discriminant_value` intrinsic needs to be updated, then :)

nagisa commented on 2 Aug 2016 — Contributor

Intrinsics are stuff internal to the compiler, therefore changes to them doesn't need discussion in the RFCs.

durka commented on 2 Aug 2016 — Contributor

Assignees

No one assigned

Labels

B-RFC-approved
B-unstable
C-tracking-issue
T-lang
T-libs
disposition-merge
finished-final-commer

Projects

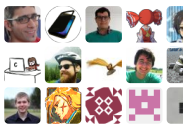
None yet

Milestone

No milestone

Notifications

34 participants



and others

I know, I just wanted to mention it here since I didn't see enums discussed in the RFC.

japaric referenced this issue on 14 Aug 2016

Provide support for arbitrary width atomics from 16bit to 2x word size (u16, u32, u64, u128) #24564 — Closed

nikomatsakis referenced this issue in **rust-lang/rfcs** on 23 Aug 2016

Add i128 and u128 types #521 — Closed

nagisa referenced this issue on 24 Aug 2016

Initial implementation of the 128-bit integers #35954 — Closed

est31 referenced this issue on 20 Nov 2016

i128 and u128 support #37900 — Closed

2 of 2 tasks complete

bors added a commit that referenced this issue on 4 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `e618fe0`

bors added a commit that referenced this issue on 4 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `229be52`

bors added a commit that referenced this issue on 4 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `cfba287`

bors added a commit that referenced this issue on 4 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `854bf23`

bors added a commit that referenced this issue on 8 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `d9738bb`

bors added a commit that referenced this issue on 8 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `03fc80e`

bors added a commit that referenced this issue on 8 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `ab66ecc`

bors added a commit that referenced this issue on 14 Dec 2016

`Auto merge of #37900 - est31:i128, r=eddyb` ··· `255dba5`

est31 referenced this issue on 20 Dec 2016

i128 and u128 support #38482 — Merged

bors added a commit that referenced this issue on 20 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `764d2f6`

bors added a commit that referenced this issue on 21 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `1602354`

cmr commented on 21 Dec 2016 — Contributor

How should FFI with this type be handled? Is there any standard ABI support for these types? AFAICT, the answer is "no", which means this type should be FFI-unsafe, and the FFI unsafety lint should be updated to reject `Ty{I,Ui}nt` with 128-bit sizes.

cmr commented on 21 Dec 2016 — Contributor

cc **@est31**

retep998 commented on 21 Dec 2016 • edited — Member

On Windows there is most definitely no standard i128 yet because the standard compiler (msvc) does not support `__int128` yet on x86. There are some really good guesses though based on the MSDN documentation.

est31 commented on 21 Dec 2016 — Contributor

> Is there any standard ABI support for these types?

It depends on the architecture. SysV defines the ABI for 64 bit architectures. For x86, its most likely its not defined. Same goes for windows: it should in theory be defined for 64 bit (just sadly nobody adheres to it, its a gigantic mess), but not on x86.

This directly maps the support of the i128 type in C, and I think generally it makes little sense to have FFI for types that don't exist in C.

est31 commented on 21 Dec 2016 — Contributor

bug report in llvm about the x86_64 ABI problems: https://llvm.org/bugs/show_bug.cgi?id=31362

nagisa commented on 21 Dec 2016 — Contributor

Also one in GCC: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=78799

bors added a commit that referenced this issue on 21 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `517d2bd`

bors added a commit that referenced this issue on 30 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `7bfa313`

bors added a commit that referenced this issue on 30 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `04d4b1b`

bors added a commit that referenced this issue on 31 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `ab2adfd`

bors added a commit that referenced this issue on 31 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `88f2ea6`

bors added a commit that referenced this issue on 31 Dec 2016

`Auto merge of #38482 - est31:i128, r=eddyb` ··· `38bd207`

This was referenced on 1 Jan 2017

[i128] C FFI mismatch on GNU/Linux with gcc #38762 — Open

[i128] ICE when calling function with #[repr(C)]'d tuple struct #38763 — Closed

robinst referenced this issue in **rust-lang/rfcs** on 4 Jan 2017

Add as_millis function to std::time::Duration #1545 — Open

est31 commented on 5 Jan 2017 — Contributor

cc #38824

est31 referenced this issue on 6 Jan 2017

members of ::std::u128 module must not be stable #38860 — Closed

clarcharr referenced this issue in **google/code-prettify** on 7 Jan 2017

Format i128 and u128 in Rust. #465 — Merged

cmr commented on 8 Jan 2017 • edited — Contributor

I've been using `#[repr(C)]` not for C FFI, but for Rust-Rust FFI, to avoid compiler-layout-dependence in some hopefully-ABI-stable code, so it'd be kinda sad to see that forbidden (mentioned on #38824)

👍 1

cmr commented on 8 Jan 2017 — Contributor

FWIW, I'm using it for nanosecond-resolution timestamps in the (to-be-published) `tempo` crate.

est31 commented on 8 Jan 2017 — Contributor

@cmr interesting use case, didn't think of it!

If we allow `#[repr(C)]` with i128, it still won't give you 100% safety with regards of stability though: On platforms where C has no i128 type, we can only make a good guess about what scheme to follow. If in a hypothetical future the scheme is defined, we will have to follow it, and maybe change our current layout, breaking the ABI.

But I guess this will still be better stability wise than the option of leaving an undefined repr.

Amanieu referenced this issue on 10 Jan 2017

Add 128-bit atomics #38959 — Merged

This was referenced on 11 Jan 2017

Cannot write integer literal for the minimum i128 (-0x8000_0000_0000_0000_0000_0000_0000_0000i128) #38987 Closed

Constants seem to be broken in nightly Rust on powerpc-unknown-linux-gnu #39061 Closed

Built-in u128 performance compared to extprim crate #39078 Open

clarcharr commented on 18 Jan 2017 Contributor

@durka imho it makes the most sense to keep the bound at `u64` and just let the compiler fit all of the variants into that `u64`

because while the user "theoretically" can put more than a 64-bit integer's worth of variants in, the compiler absolutely can't handle that because it's running on a 64-bit system. it may require the compiler not using the user's defined integer values for the enum as the discriminant values, though, at least in the case of `u128`

I honestly don't mind things being a tad slower for 128-bit enums whose variants are not unique in the first 64 bits because that's such an edge case that it doesn't really matter as long as it's reasonably fast

durka commented on 18 Jan 2017 Contributor

@clarcharr you don't need to have so many variants, as you could do `#[repr(u128)] enum Foo { A = 0x_very_large_number_here }`

clarcharr commented on 18 Jan 2017 • edited Contributor

@durka that was actually the point; for `u128` you'd incur a slight performance penalty to allow the discriminant to not reflect the actual value of the enum, to avoid the performance penalty of the `discriminant` intrinsic using non-native integer arithmetic every single time on 64-bit CPUs

(I could be totally wrong about the performance loss though; maybe it's not that much)

durka commented on 18 Jan 2017 Contributor

Well, my ideal solution to the issue is to use an intrinsic trait to let
`discriminant_value` have a differing return value per enum :)
...

est31 commented on 18 Jan 2017 Contributor

@durka a trait would be ideal, but wouldn't that require an RFC? And, for backwards compatibility sake (see #39137) it would probably have to return u64 for all non 128-bit types.

clarcharr commented on 18 Jan 2017 Contributor

@est31 see **@nagisa**'s comment; intrinsics don't require RFCs. and the `Discriminant` struct is intentionally opaque to avoid backwards-compatibility problems

est31 commented on 18 Jan 2017 Contributor

@clarcharr apparently they do: rust-lang/rfcs#1696

clarcharr commented on 18 Jan 2017

Contributor

@est31 I think you're mistaken; that's not an intrinsic, that's a function in `std::mem`. intrinsics like `discriminant_value` are not stable and can be changed without RFC, whereas functions like that are on the path for stability so that users can use them. that's why additional care is put into making the struct returned by `discriminant` opaque so that users can't just inspect it and assume that it'll always be a `u64`

kkimdev referenced this issue in **BurntSushi/byteorder** on 21 Jan 2017

u128 i128 support #65 Closed

clarcharr commented on 27 Jan 2017

Contributor

As brought up in #39324, I think that a generic `cfg(has_i128)` flag or something similar is necessary if we want to ensure forwards-compatibility for platforms which don't have 128-bit integer support.

Right now I'm thinking things like embedded devices. For example, in the PR I linked, it might be reasonable to assume that a system would have IPv6 support without needing full-blown 128-bit integer support.

clarcharr commented on 27 Jan 2017

Contributor

Additionally, also from that PR, I think that'd be nice if there were a better error for 128-bit integer literals that aren't annotated with `u128` or `i128`. Right now it just says that the integer is too big but doesn't clarify that it fits in a 128-bit integer.

Either that, or we could allow non-annotated integer literals to coerce to `i128` and then add a lint-by-default that warns that the literal is larger than `u64`.

nagisa commented on 27 Jan 2017

Contributor

You can *always* implement i128 operations in terms of operations on i64, and then operations on i64 in terms of operations on i32, and then i32 in terms of operations on i16. You can also in the other direction and implement i256 operations in terms of operations on i128, i512 in terms of i256 ad nauseam. They might not be fast, but certainly not impossible to support.

One may have concerns about ABI for i128 not being specified for some targets, but that does not prevent i128 from being used within Rust code with some arbitrary, but consistent ABI.

So… if rust isn't supposed to run on machines with literally less than 16 bytes of memory (plus the memory necessary to do the operations, of course) it is literally impossible to have a target which *cannot* support i128.

If you don't know of any such machine, then please stop throwing portability concerns all over the place, thanks! (**@brson**, **@alexcrichton**)

There's #38824, which some may be inclined to cite as an example of *platforms* which do not support i128. They do support it just fine. LLVM backends for those targets are buggy, that's all.

👍 8

brson commented on 28 Jan 2017

Contributor

Thanks for the clarifications **@nagisa**.

Here's what I'm thinking more precisely: when our original selection of atomics were conservative because of portability concerns. When we added additional atomics to the language, we put them behind CPU features (see here for an example).

128-bit integers strike me as a very similar case. They support a feature that is not universal and will need to be emulated on many architectures if they are not cpu-specific.

Often in Rust we put value on having the language closely represent the hardware it targets. There's e.g. no way to emulate atomics in Rust at all today.

So I hope you will see the similarity between these two cases and why I might expect 128-bit integers to be treated similarly.

brson referenced this issue on 28 Jan 2017

Ipv6Addr <-> u128 #39324 Merged

retep998 commented on 28 Jan 2017 Member

I don't see atomics and i128 support as the same thing. Atomics are the kind of thing where you *need* hardware support or you simply cannot do the operation atomically. i128 meanwhile can very easily be emulated with smaller integer operations. Aside from LLVM failing hard, the worst that can happen is an i128 operation will run slow. We already emulate many i64 ops on 32-bit platforms, so what makes i128 so special?

👍 13

est31 commented on 28 Jan 2017 Contributor

@brson

> 128-bit integers strike me as a very similar case. They support a feature that is not universal and will need to be emulated on many architectures if they are not cpu-specific.

I think the main use case for 128-bit integers is to be able to do operations on them without having to emulate them yourself or using libraries that do it for you. If your program then can't run on various targets its not very good I think.

I also agree with **@retep998** that 128-bit integers are less similar to atomics, and more similar to 64-bit or 32-bit integers, which are currently already emulated on 32 bit and 16 bit CPUs.

In fact, in my PR to the soon-to-be-used compiler-builtins library, I was able to use the same generic macro (link goes to the multiplication, but the other operations were similar) for both i128 and 64 bit integer operations, for `mulo{s,d,t}i4` even for 32 bit integers.

Also, the main problem that breaks i128 integer support on these platforms is not missing emulation of operations. Its already available and well working for rust's tier1 32 bit targets. The problem is rather the missing support by LLVM to handle such large integers in the only task that is left to LLVM: hauling the value around. From heap to stack, from one function to the other, doing the calculations like how much to reserve in the stack. All these are tasks that should easily scale to 128 bit integers, all it needs is patching LLVM.

The core reason for why LLVM doesn't support 128-bit integers on those targets is that C doesn't officially specify them, and only some compilers provide it unofficially. If we make 128-bit integers platform dependent, we won't improve much on C in this regard.

👍 4

SimonSapin commented on 28 Jan 2017 Contributor

Aren't `u64` and `i64` already implemented "in software" on some of our supported targets? If so, why should 128 be any different?

ollie27 referenced this issue on 30 Jan 2017

Fix TryFrom for i128/u128 #39408 Merged

aturon added the **T-libs** label on 30 Jan 2017

bors added a commit that referenced this issue on 31 Jan 2017

`Auto merge of #39408 - ollie27:i128_try_from, r=sfackler` ··· `e92e317`

alexcrichton commented on 4 Feb 2017 **Member**

PR #39324 was discussed in libs triage a little while ago, specifically related to the portability and implementation concerns of i128 and u128. The discussion largely just concluded that we'll continue on here, and I don't recall any particulars which haven't already been mentioned here.

Personally I figured I'd add some thoughts as well to this. I voiced concern on the original RFC about the seriousness of adding a new primitive type to the language, as such a position for a type has very strong implications about how well it's supported in both the language and libraries.

Both **@nagisa**'s and **@est31**'s work here implementing these types has been super impressive, however. I think they've done a great job of fleshing out what I at least would expect is full support for these types. All the necessary traits and such in the standard library are implemented and tested, the compiler works with various literals and such, etc. Overall, I definitely wanted to reiterate very nice work to everyone who's been working on these types!

I very much sympathize with **@brson**'s concern about the portability of these types. The current implementation and testing convinces me that our current suite of platforms can cope with i128/u128 quite well. As a primitive type, though, are we ready to expand this to all future targets that Rust may support?

I think one example here is that 128-bit integers aren't supported on Emscripten for now. Does this mean we need to add a `#[cfg]` for targets that don't support i128 for whatever reason? Does this mean that we should block support for a platform until it supports i128? I'm not sure!

In general i128 seems to me as an entirely separate class of support than, say, i32, and i64. The smaller types have been supported by C/C++ and nearly all compilers for *decades*. This means that they're battle tested, proven to work, and clearly portable across platforms. It's my understanding though that i128 is much newer. This may imply LLVM bugs, less platform support (e.g. emulation not implemented), or just other miscellaneous bugs are lying in wait. The heroic effort required to land i128 and u128 I feel is a testament to this, there were quite a few bugs that needed working through.

All that is basically boiling down to the point that I don't think we can just bat away portability as a concern. Emscripten seems to at least be an empirical data point of a platform that doesn't support i128/u128 right now. I don't equate portability with "does the platform have 128 bit registers" as almost none do, to be clear, just whether the compiler can emit correct operations for the type on the platform.

frewsxcv added a commit to frewsxcv/rust that referenced this issue on 5 Feb 2017

`Rollup merge of #38959 - Amanieu:atomic128, r=alexcrichton` ··· `868525b`

frewsxcv added a commit to frewsxcv/rust that referenced this issue on 5 Feb 2017

`Rollup merge of #38959 - Amanieu:atomic128, r=alexcrichton` ··· `af81dc0`

frewsxcv added a commit to frewsxcv/rust that referenced this issue on 5 Feb 2017

`Rollup merge of #38959 - Amanieu:atomic128, r=alexcrichton` ··· `c4c6c49`

bors added a commit that referenced this issue on 5 Feb 2017

`Auto merge of #39408 - ollie27:i128_try_from, r=alexcrichton` ··· `fc02736`

est31 commented on 7 Feb 2017 Contributor

I've filed a bug report for emscripten at kripken/emscripten-fastcomp#169

scottmcm commented on 7 Feb 2017 **Member**

> 128-bit integers aren't supported on Emscripten

Javascript targets already need to emulate the 64-bit types, right? It feels like gating types on any definition of "nice platform support" (be that registers, instructions, or what) means that the 64-bit ones also ought to be `cfg` d. (Silly thought experiment: if I made a rust to T-SQL stored procedure compiler, would the unsigned types need to be `cfg` d?) The restrictions the library imposes on usize assumptions (might be smaller than `u16`) suggest that perhaps even 32-bit stuff ought to be `cfg` d under that gate criteria.

I think that `cfg` hiding `i32` (or even `u64`) is pretty crazy, so that implies that things can be primatives if they're broadly reasonable things to use. And are integers bigger than 64-bit (but not BigInteger) reasonable? I think they are; for example `std::time::Duration` is already emulating `u96` (well, partially, and more like `u93.897` ...) in the standard library. ZFS, with >64-bit storage, is over 10 years old now.

Let's not add 256-bit integers, though :P

(Hmm, a three-i53 Javascript emulation of u128 would probably be no worse, and plausibly would be better, than the emulated-u64 plus u32 stuff it needs to do now for Duration—not that duration math is going to be anyone's performance bottleneck.)

👍 2

nikomatsakis commented on 7 Feb 2017 Contributor

@scottmcm I think that **@alexcrichton**'s point was not whether some emulation is required, but whether emulation **exists** and is reliable:

> I don't equate portability with "does the platform have 128 bit registers" as almost none do, to be clear, just whether the compiler can emit correct operations for the type on the platform.

As he wrote:

> In general i128 seems to me as an entirely separate class of support than, say, i32, and i64. The smaller types have been supported by C/C++ and nearly all compilers for decades. This means that they're battle tested, proven to work, and clearly portable across platforms.

nikomatsakis commented on 7 Feb 2017 Contributor

My take: I am persuaded by the analogy to emulating u64 on a 32-bit system, but I am also persuaded that 128-bit support is going to be less widespread and solid than 64-bit support. I want to be clear on what exactly we are debating about:

- Whether a `i128` type is available in the lang at all?
- Whether a `i128` type is *required* for all platforms?

I tend to think of things these days in terms of the portability RFC that @aturon has pending. Basically, there are "mainstream" platforms that code targets by default -- if you wish to target something else, you would "opt into" that in some way. This includes both a narrower range of features ("I only want to use things appropriate for 8-bit ATARI CPUs") as well as a wider range of features ("I am focused on Windows, so I don't care about unix compatibility").

In this case, I would think that we should definitely make it possible to eschew using `i128` if it seems inappropriate. The RFC leaves that case a bit under-specified, and naturally it is focused on libstd and not the lang, but it seems like the basic idea would be to have a "target feature" for 128-bit support.

Anyway, under this perspective, the main question is whether this target feature ought to be part of our default configuration. It seems to me that this is a fairly straight-forward question -- in theory. =) That is, it follows somewhat mechnically from how well-supported i128 is on the "main platforms". I think historically that's been basically "common desktop/laptop CPUs", but I feel like this is something which isn't totally *decided*.

alexcrichton commented on 7 Feb 2017 **Member**

Yes to be clear I am personally ok with emulation of i128 on a platform, e.g. x86_64. To support a platform we just have to have working emulation! Emscripten for example I believe is an empirical example of where the emulation does not work (or just isn't implemented).

@nikomatsakis I think you raise some good points. I do think that i128 should be in the language itself, as it has clear benefits with hardware support in various instructions. Even if it's emulated, the compiler-emulated version has historically purportedly been superior to library emulation. To me this is a question of how we talk about platform compatibility of i128.

I do think you also raise a good point with **@aturon**'s RFC, and in that case it's just a question (for now) as to whether i128 is in the mainstream "std" scenario or not. Put another way, does this compile by default?

```
fn main() {
    println!("{}", 1i128);
}
```

My gut says that "yes", we want i128 in the mainstream scenario. The impressive work done to pass all the test suites on all our tier 1 platforms I think is a testament to that! I do think, however, that we'll want to document that *maximally portable code* may not wish to use i128. New Rust platforms may not have the emulation working quite yet or may not just be battle tested much.

In that sense I see i128 in the same class of support as `std::thread`. It's available for mainstream use cases and we'll test to make sure it works. If you want to work everywhere (or as many places as possible) you may be best off avoiding it. We wouldn't require i128 support to add a target to the compiler, just as some targets don't have `std::thread` or even much float support.

👍 2

aturon commented on 7 Feb 2017 **Member**

@alexcrichton

> My gut says that "yes", we want i128 in the mainstream scenario. The impressive work done to pass all the test suites on all our tier 1 platforms I think is a testament to that! I do think, however, that we'll want to document that *maximally portable code* may not wish to use i128. New Rust platforms may not have the emulation working quite yet or may not just be battle tested much.

This is precisely my view as well. In particular, that means we should feel free to use the type where appropriate in `std`; I believe there are some methods we'd like to add to `Duration`, for example, that would benefit from this type.

So I personally would say that once we feel completely confident in the implementation on tier 1 platforms, we can go forward with stabilization.

👍 5

alexcrichton commented on 7 Feb 2017 **Member**

@aturon I might personally be less zealous about using i128/u128 in types throughout libstd (due to the possible portability concerns), but we can always cross that bridge when we get there :)

est31 commented on 7 Feb 2017 • edited ▾

Contributor

> I might personally be less zealous about using i128/u128 in types throughout libstd

People being wary with i128 because it will render their code non portable (and starting/continuing to use i128 emulated by libraries) is my main reason why I'm against disabling i128 on platforms without a supporting backend. If we are going to accept that some backends don't implement i128 emulation (I think both **@alexcrichton** and **@aturon** do), then we should at least emulate it on the rust side, so that the backend sees the equivalent of `(u64,u64)` or something.

👍 9

cuviper referenced this issue in **rust-num/num** on 23 Apr 2017

Update to Serde 1.0 #281 Closed

nagisa commented on 3 May 2017

Contributor

Should we consider stabilisation now?

The thing has been baking for a while now and seems to mostly work (as evidenced by rustc itself, which is the largest user of i128, not breaking). The few issues that are still open are bugs, mostly on the LLVM side, and do not affect tier 1 platforms.

👍 4

bstrie referenced this issue on 3 May 2017

Remove unnecessary unstable features from libsyntax (and all transitive dependencies) to allow rustfmt to work on stable and enable the deprecation of syntex #41732 Closed

est31 commented on 3 May 2017

Contributor

> Should we consider stabilisation now?

What about the `repr(i128)` issue with enums? Is it solved in some way?

nagisa commented on 3 May 2017

Contributor

What's the issue with `repr(i128)` enums? We might need to "fix" `discriminant_value` for that, but that's it, I think?

est31 commented on 3 May 2017

Contributor

yeah. I remember having heard about some problem with debuginfo/gdb, but maybe I mixed up something, and even if that's a bug so probably can be fixed later on as well.

nagisa commented on 4 May 2017 • edited ▾

Contributor

Oh right, I remember now. There's a few APIs in LLVM that do not allow for i128, most notably those related to discriminants in debug info or some such.

I would probably keep `#[repr(i128)]` simply unstable for longer, in this case and stabilise i128 otherwise.

👍 5

kennytm referenced this issue on 4 May 2017

AtomicU128 uses __atomic_load function on OSX #39590 Open

est31 referenced this issue on 7 May 2017

Casting u128::MAX to f32 is undefined #41799 Closed

nagisa commented on 8 May 2017 Contributor

We'll have to figure out #41799 before stabilising.

👍 1

SimonSapin commented on 11 May 2017 Contributor

> What about the `repr(i128)` issue with enums? Is it solved in some way?

Is that ever useful? Can we disallow it, if it's buggy?

cuviper commented on 11 May 2017 Member

Perhaps `repr128` can be a separate unstable feature.

clarcharr commented on 11 May 2017 Contributor

I personally like the idea of keeping `repr(i128)` under a feature gate and pushing the discussion to after i128 is stabilised. Crates like `syn` and perhaps various `serde` implementations could benefit from stable i128.

SergioBenitez referenced this issue in **SergioBenitez/Rocket** on 19 May 2017

Compile with stable Rust #19 Open

7 of 18 tasks complete

est31 commented on 20 Jun 2017 • edited Contributor

Something else which might be blocking stabilisation: FFI support. There is no official C support for i128 and the unofficial one that exists is inconsistent on windows. I think on any target but `x86_64` we should treat any `i128` inside a `repr(C)` data structure as non C-type. Its only a matter of lints, so it *might* be resolved after stabilisation as well (lints can be extended after the fact afaik).

JoeyAcc referenced this issue in **chronotope/chrono** on 28 Jun 2017

Add Serialize/Deserialize for Duration #117 Open

pmarks referenced this issue in **kennytm/extprim** on 1 Jul 2017

support serde Serialize/Deserialize #10 Closed

durka referenced this issue on 22 Jul 2017

ICE getting discriminant of #[repr(i128)] enum #43398 Open

Mark-Simulacrum added the **C-tracking-issue** label on 22 Jul 2017

fitzgen referenced this issue in **rust-lang-nursery/rust-bindgen** on 25 Jul 2017

Tracking rustc bugs/features/RFCs that affect bindgen #849 Open

1 of 10 tasks complete

mrZalli referenced this issue in **ron-rs/ron** on 22 Aug 2017

[RFC] Separate number types #47 Open

alexcrichton added a commit to alexcrichton/rust that referenced this issue on 2 Sep 2017

`rustc: Separately feature gate repr(i128)` ... `51a478c`

alexcrichton referenced this issue on 2 Sep 2017

rustc: Separately feature gate repr(i128) #44262 Merged

GuillaumeGomez added a commit to GuillaumeGomez/rust that referenced this issue on 8 Sep 2017

`Rollup merge of #44262 - alexcrichton:repr-128-gate, r=nikomatsakis` `d455f27`

...

coder543 commented on 9 Sep 2017

I see that both #44261 and #44262 were opened a week ago. Is there any hope of resolving the final blocker (#41799) before the `impl` period begins? Having 128-bit integers be supported at the language level would be super nice.

alexcrichton commented on 9 Sep 2017 Member

@coder543 that issue, like #10184, I believe mostly just needs data collection and a proposal to move forward. Those can certainly happen at any time! (including the impl period)

frewsxcv added a commit to frewsxcv/rust that referenced this issue on 9 Sep 2017

`Rollup merge of #44262 - alexcrichton:repr-128-gate, r=nikomatsakis` `987e255`

...

frewsxcv added a commit to frewsxcv/rust that referenced this issue on 9 Sep 2017

`Rollup merge of #44262 - alexcrichton:repr-128-gate, r=nikomatsakis` `8ba533e`

...

GuillaumeGomez added a commit to GuillaumeGomez/rust that referenced this issue on 10 Sep 2017

`Rollup merge of #44262 - alexcrichton:repr-128-gate, r=nikomatsakis` `9af7de1`

...

kennytm referenced this issue on 12 Sep 2017

impl Hasher for {&mut Hasher, Box<Hasher>} #44015 Merged

alexcrichton commented on 12 Sep 2017 Member

@rfcbot fcp merge

Ok this has been sitting for some time now and hasn't seen a whole lot of activity, but that being said I don't think we've seen many surprises or bugs with the 128-bit integers so far. We've long had to write our own support in compiler-rt but that's now done in the compiler-builtins project where we write many of the intrinsics ourselves (and hopefully is a bit more cross-platform!).

As expected there are LLVM targets that have yet to implement support for i128/u128, ranging from embedded targets like AVR to "weird" targets like NVPTX to larger ones like Emscripten. Despite this, however, the presence of 128-bit integers I feel doesn't preclude Rust working on these targets. It's already the case that an arbitrary library won't work on one of these targets today, and I don't think i128/u128 will make the situation worse or better! Some points on this though:

- Right now libcore doesn't have a compilation profile where 128-bit integers are omitted, but I personally feel this is pretty reasonable to add in the future.
- Maximally portable code will likely still not use 128-bit integers, for example we likely won't use it in the standard library for these reasons (e.g. we want libstd working with Emscripten). Many codebases, however, don't need that level of portability, and can certainly benefit from i128/u128!

Of the remaining blockers listed for stabilization on this issue only one remains, #41799. This to me is an open question in Rust that's *already* a problem with issues like #10184 and #15536. Like the portability issue, I don't think i128/u128 are making this story worse than it is today, and presumably whatever solution we come up with in the future will naturally extend to 128-bit integers as well. Along these lines, I'd propose stabilizing 128-bit integer support before requiring this to be fixed.

So concretely what's being proposed for stabilization is:

- The `i128` and `u128` types
- Language level support for mathematical operations on these types and casts
- All various API support in libstd/libcore, e.g. the i64/u64 API but reflected on 128-bit types
  - i128 primitive type
  - u128 primitive type
  - i128 module
  - u128 module
  - impls like `From<u128> for Ipv6Addr`

What is **not** being stabilized is:

- `#[repr(i128)]` on an `enum`, this is behind a separate feature gate.
- 128-bit integers as "ffi safe types" so we have the freedom to tweak the API in the future as necessary.

I'm curious to hear what others think about this!

rfcbot commented on 12 Sep 2017 • edited by aturon

Team member **@alexcrichton** has proposed to merge this. The next step is review by the rest of the tagged teams:

- [x] **@BurntSushi**
- [x] **@Kimundi**
- [x] **@alexcrichton**
- [x] **@aturon**
- [x] **@cramertj**
- [x] **@dtolnay**
- [x] **@eddyb**
- [x] **@nikomatsakis**
- [x] **@nrc**
- [x] **@pnkfelix**
- [ ] **@sfackler**
- [x] **@withoutboats**

No concerns currently listed.

Once these reviewers reach consensus, this will enter its final comment period. If you spot a major issue that hasn't been raised at any point in this process, please speak up!

See this document for info about what commands tagged team members can give me.

rfcbot added the **proposed-final-comment-period** label on 12 Sep 2017

dtolnay commented on 12 Sep 2017 — Member

What does it look like when this fails on "weird" targets? So if I add u128 impls to Serde, does that mean Serde no longer compiles on many targets? Is there a target_feature or similar gate to do this correctly without a Cargo cfg?

SimonSapin commented on 12 Sep 2017 — Contributor

> As expected there are LLVM targets that have yet to implement support for i128/u128

Isn't there a "software" fallback for architectures that do not have CPU instructions for 128-bit arithmetic?

rkruppe commented on 12 Sep 2017 • edited — Contributor

@SimonSapin Yes, that's the compiler-rt/compiler_builtins thing mentioned above. However, targets still need to handle 128 bit types in some contexts (e.g., in ABI lowering) and emit calls to the software implementation of various operations. See #41132 for an example of what it looks like when the backend doesn't handle i128 (in short: a very ugly and nonsensical crash deep in the backend).

est31 commented on 12 Sep 2017 — Contributor

@SimonSapin I'm not aware of a target that has native 128-bit integer arithmetic support (native meaning in this context that common operations like addition, multiplication, division, etc are possible in one single instruction). Instead, operations are expressed through their smaller counterparts (example). For the platforms that have support for that emulation, the backend is compatible and either lowers operations to calls to `compiler_builtins` functions (provided by us), or figures out its own best way to do lowering if its very smart. However, the backend needs to provide *some* support of its own, like specifying how something should be returned, and obviously it shouldn't also give an assertion failure when being given 128 bit integers.

Thanks to register allocation, each target that has enough memory (as in RAM memory, not register memory) to hold 128 bit integer operands is generally able to provide such support. There is nothing *preventing* AVR or emscripten backends to implement 128 integer support, its more a question of doing the work.

In fact, this emulation is already present for 32 bit integers on targets that have no native 32 bit instructions. On such targets, 64 bit integers are expressed through 16 bit instructions as well! The actual algorithms we provide in libcompiler_builtins are generic and handle the `one operation on 2 * x bits -> multiple operations on x bits` step.

@alexcrichton

> Right now libcore doesn't have a compilation profile where 128-bit integers are omitted, but I personally feel this is pretty reasonable to add in the future.

I want to say that the proposal is sensible; wasm is obviously of higher priority than support for 128 bit integers. However if i128 is not guaranteed, it will have a chilling effect on adoption, where people avoid the language-native feature, in order to be cross-platform. Those who have a pressing need for 128 bit integers would instead choose emulation via external crates (there are ones already on crates.io for this). That's obviously not a good outcome for 128 bit integers!

In order to fight this, we have two options:

- Try to do some compiler-side emulation, where we give the backend the same stuff we'd give it for `(u64, u64)`. The advantage is that we could always provide i128 support to users. But this is likely a big amount of work, as it might consist of adding a lot of special cases.
- Fix the backends.

nagisa commented on 12 Sep 2017 • edited ▾ Contributor

,I'm super strongly opposed on support, conditional on the target. I'd much rather "fix" the shoddy LLVM backends somehow. If we manage to fix all the backends and slip in a test with a comment that Rust *requires* i128 to be properly supported to LLVM upstream, it would be ideal.

Otherwise the best next approach seems to just invent our own ABI for those bad targets and lower to compiler-rt calls ourselves during translation rather than relying on LLVM to do it. Hopefully just passing i128 by reference would be enough.

👍 9

alexcrichton commented on 12 Sep 2017 **Member**

@est31 yeah it's true that it may hinder adoption if we don't guarantee that, and that's a good question for stabilization! I'm personally proposing stabilization based on the condition it'd still have a warning "this may not be available on all platforms" as it, to me, doesn't seem like it should preclude usage on tier 1 platforms that have the support.

Note that a crates.io fallback though may not be the end of the world, it could presumably use emulation on any target that doesn't support i128 and use i128 natively on any target that does support it, presumably achieving the same level of performance?

I definitely agree that ideally rustc and/or LLVM would fix everything here for us, and this may be the question that makes or breaks the proposal for stabilization here. If we'd like to require that then we can't stabilize this as there's work yet to be done!

est31 commented on 13 Sep 2017 Contributor

> this may be the question that makes or breaks the proposal for stabilization here

Feel the same, I think we should do what **@nagisa** is suggesting. I think now that I have asked too early for stabilization...

> Note that a crates.io fallback though may not be the end of the world, it could presumably use emulation on any target that doesn't support i128 and use i128 natively on any target that does support it, presumably achieving the same level of performance?

A library solution would probably achieve the same performance level, but I think the whole point of a "native" i128 type is that its nicer. E.g. you can directly have literals like for any other integer type, or you have all the functions implemented that are implemented for normal integers, etc. Also libraries like serde will more likely give support to i128 if its part of the language...

sfackler commented on 18 Sep 2017 **Member**

I'm uncomfortable landing this if it's not supported in some way on all platforms.

👍 3

asterite referenced this issue in **crystal-lang/crystal** on 22 Sep 2017

Increase the precision of Time and Time::Span to nanoseconds #5022 Merged

alexcrichton commented on 27 Sep 2017

Member

@rfcbot fcp cancel

Ok I've now been convinced by **@sfackler** and other members of the libs team that we have a new blocker for stabilization, which is a "reasonable ish" story for enabling this type to work on all platforms. Today's incompatibility with Emscripten is pretty worrying, and the prospects of adding popular future platforms that also don't support 128-bit integers was also somewhat worrying.

I think that the best way forward for fixing this concern (and then moving back on the path to stabilization) would be to likely implement a lowering for 128-bit integers in rustc itself. It seems that if we implement this in LLVM it may lead to an implementation-per-platform whereas if we were to implement it at the rustc translation layer it may end up being much more platform-agnostic, only requiring us to implement it once.

👍 4 ❤️ 3

rfcbot commented on 27 Sep 2017

@alexcrichton proposal cancelled.

rfcbot removed the **proposed-final-comment-period** label on 27 Sep 2017

clarcharr commented on 27 Sep 2017 Contributor

@alexcrichton A tracking issue for that should probably be added to the OP.

👍 1

ebfull referenced this issue in **ebfull/pairing** on 27 Sep 2017

Change CI to compile on latest nightly #45 Open

oherrala added a commit to oherrala/untrustended that referenced this issue on 2 Oct 2017

`Add support for 128-bit integers (RFC 1504)` ··· ✓ `f97bc73`

est31 commented on 1 Nov 2017 Contributor

@clarcharr I've opened a tracking issue: #45676

est31 referenced this issue on 1 Nov 2017

Implement i128 lowering for backends without stable i128 support #45676 Open

isislovecruft commented on 2 Nov 2017

Hi! Yay! Thanks for working on this! We're using `u128`s in curve25519-dalek for 64-to-128-bit widening field arithmetic, which roughly doubles the speed of our crypto. Since Tor (my day job) wants to use ed25519-dalek as part of our switching to Rust, it would be *awesome* for us if `u128`s were stable in time such that it ends up in the next Debian release (most Tor relays run on Debian).

👍 1

est31 commented on 2 Nov 2017 Contributor

@isislovecruft I'm not familiar with inner debian workings, but stretch has entered soft freeze 8 months after the release of the previous version. So probably soft freeze will be somewhere in February. From that deadline you also need to subtract the delay created by the debian packaging team for the compiler. It takes 7-13 weeks for a change in the compiler on the master branch to appear inside an official release of Rust. It will be very tight...

fschutt commented on 7 Nov 2017 — Contributor

A question: Is multiplying a `i64 as i128 * i64 as i128 = i128` stable? I am looking to replace this:

https://github.com/fschutt/clipper-rs/blob/master/src/cpp/clipper.cpp#L354-L378

... and wanted to ask if (at least on x86 and x64) this is stable enough or if I should roll my own i128.

sfackler commented on 7 Nov 2017 — Member

@fschutt the i128 type is not stable.

👍 1

Natim referenced this issue in **skade/rust-three-days-course** on 8 Nov 2017

Remove the soon for u128 in the basic-types.chapter #108 — Merged

bstrie commented on 11 Nov 2017 — Contributor

@isislovecruft Is it extraordinarily imperative that this get into the next Debian? Normally unstable features will spend a full release cycle in FCP, which would mean that i128 would hit stable on Feb 15 at the absolute earliest. To hit the prior stable release on Jan 4 we'd need to promote this to beta during our next cutoff on Nov 23, giving us 13 days to make a decision (and note I speak on behalf of neither the lang team nor the libs team). Normally I'd say that's a completely unreasonable target, and it is *kinda* unreasonable (:P), but given how much we like Tor (and seeing Rust get used in big-name OSS projects in general, of course!), it *might* be possible to rush given that 1) this feature is, conceptually, minor; 2) the blocker at #41799 , despite being a soundness bug, is just a subset of literally Rust's oldest known soundness bug and so it *might* be argued that this adds no new unsoundness; and 3) the blocker at #45676 doesn't affect any tier-1 platforms and so it *might* be argued that we could weather a short period of this feature being somewhat non-portable in the wild. But rushing something like this would take a lot of convincing, so I'd only bother if it were incredibly imperative to Tor's ongoing use of Rust. :)

vitiral commented on 28 Nov 2017 — Contributor

it looks like #41799 has recently been closed + merged, which leaves just the > tier-1 platforms as the blocker (not saying this means this should be stabilized yet, just a heads up).

nagisa commented on 28 Nov 2017 — Contributor

We still need to figure out what platforms need the manual lowering.

est31 commented on 29 Nov 2017 — Contributor

We also need to check whether the stuff we implemented for those platforms is enough, or whether more work to support them is needed. We can only consider this to be finished if i128 is not commented out in libcore on any platform any more.

dhardy referenced this issue in **dhardy/rand** on 29 Nov 2017

Shootout: small, fast PRNGs #52 Open

nagisa referenced this issue on 10 Dec 2017

i128 lowering implementation won't work for `const fn` evaluated at runtime Closed
#46622

scottmcm commented on 20 Dec 2017 Member

Per #46290 (comment), this might also be blocked on LLVM 5.0 (cc #46819?)

nagisa commented on 20 Dec 2017 Contributor

That comment is sort-of irrelevant, as the failure occurs in emscripten, rather than LLVM. The linked comment refers specifically to using LLVM WASM backend without emscripten at all.

spacejam commented on 2 Jan

Should alignment be 16 rather than 8?

eddyb commented on 2 Jan Member

@**spacejam** Not a bug, replied to that issue.

👍 1

scottmcm commented on 18 Jan Member

I don't think this is done, and am happy to continue working towards it, but I need some more guidance here. Any thoughts on next steps? Anyone have a tier>1 platform and want to try it?

alexcrichton commented on 18 Jan Member

@**scottmcm** perhaps emscripten? (we're even running tests for emscripten!)

hdevalence commented on 29 Jan

It looks like all the issues on the "blocking stabilization" checklist at the top are checked off.

What's left to do to stabilize `u128` s?

est31 commented on 29 Jan Contributor

@**hdevalence** we should now get an overview over the various targets of rustc and check which ones are still lacking `u128` support. Those then need to be fixed. Then we can ship :). I do *not* want `u128` end up being a compatibility hazard.

withoutboats commented on 29 Jan Contributor

@**rfcbot** fcp merge

> Those then need to be fixed. Then we can ship :).

Since shipping takes roughly 3 months, it seems like a good idea to mobilize now and light a fire under ourselves to actually finish this.

We knew emscripten needed this, does the new wasm target need it as well? Comments on this thread make it seem like these are the only platforms we consider supported that this is a problem for, if there are others, add them to the list.

❤️ 8

rfcbot added the **proposed-final-comment-period** label on 29 Jan

rfcbot commented on 29 Jan • edited by cramertj

Team member **@withoutboats** has proposed to merge this. The next step is review by the rest of the tagged teams:

- [x] **@BurntSushi**
- [x] **@Kimundi**
- [x] **@KodrAus**
- [x] **@alexcrichton**
- [x] **@aturon**
- [x] **@cramertj**
- [x] **@dtolnay**
- [x] **@eddyb**
- [x] **@nikomatsakis**
- [x] **@nrc**
- [x] **@pnkfelix**
- [ ] **@sfackler**
- [x] **@withoutboats**

No concerns currently listed.

Once a majority of reviewers approve (and none object), this will enter its final comment period. If you spot a major issue that hasn't been raised at any point in this process, please speak up!

See this document for info about what commands tagged team members can give me.

est31 commented on 30 Jan • edited Contributor

I know of the following backends:

- [ ] NVPTX support (#38824, #41132)
- [ ] emscripten support (kripken/emscripten-fastcomp#169)
- [ ] atmel backend support (avr-rust#36 , avr-rust#57, avr-rust/libcore#5, avr-rust#94)
- [x] verify that unknown wasm support is working (verified: #35118 (comment))
- [x] mips (#41222)

But maybe there are more.

pnkfelix commented on 31 Jan **Member**

I am a little worried that I know I've been wrestling with a code gen bug in the code generated for `rustc` itself that seems somewhat likely to be related to issues with LLVM and `i128` . (See #47381)

But if anything, maybe that is incentive for us to stabilize this, so that other people will encounter the bugs, and we'll acquire a better suite of test inputs! :)

withoutboats commented on 2 Feb — Contributor

Alex says that 128s work fine on `wasm-unknown-unknown` & that there is a test for this, the only problem was with emscripten.

To be clear, the other two platforms are nvidia GPUs (NVPTX) and a 16bit chip (atmel), both tier 3 platforms. Both of these platforms are extremely unusual platforms, and I suspect there are platform compatibility problems with many libraries *already* (e.g. assuming that usize is at least 32 bits). I think it would be great to support 128bit integers on these platforms, but I do not think we should block stabilizing the feature on this.

👍 7

plietar referenced this issue in **librespot-org/librespot** on 8 Feb

API review for librespot-core #130 — Open

23 of 24 tasks complete

rfcbot commented on 16 Feb

🔔 **This is now entering its final comment period**, as per the review above. 🔔

rfcbot commented on 16 Feb

🔔 **This is now entering its final comment period**, as per the review above. 🔔

rfcbot added **final-comment-period** and removed **proposed-final-comment-period** labels on 16 Feb

cramertj commented on 16 Feb — Member

@rfcbot Are you having a bad day? **@sfackler** hasn't checked their box yet.

cramertj commented on 16 Feb — Member

@rfcbot fcp cancel

rfcbot commented on 16 Feb

@cramertj proposal cancelled.

rfcbot removed the **final-comment-period** label on 16 Feb

cuviper commented on 16 Feb — Member

That's probably the new FCP process in action:
https://internals.rust-lang.org/t/psa-tweaks-to-fcp-process/6775

cramertj commented on 16 Feb — Member

@cuviper oh gosh darn it! I didn't see that-- time to undo my cancels :) Thanks for the heads-up.

cramertj commented on 16 Feb — Member

@rfcbot fcp merge

rfcbot commented on 16 Feb • edited by cramertj

Team member **@cramertj** has proposed to merge this. The next step is review by the rest of the tagged teams:

- [x] **@alexcrichton**
- [x] **@aturon**
- [x] **@cramertj**
- [x] **@dtolnay**
- [x] **@eddyb**
- [x] **@nikomatsakis**
- [x] **@nrc**
- [x] **@pnkfelix**
- [ ] **@sfackler**
- [x] **@withoutboats**

No concerns currently listed.

Once a majority of reviewers approve (and none object), this will enter its final comment period. If you spot a major issue that hasn't been raised at any point in this process, please speak up!

See this document for info about what commands tagged team members can give me.

rfcbot added the **proposed-final-comment-period** label on 16 Feb

rfcbot commented on 16 Feb

🔔 **This is now entering its final comment period**, as per the review above. 🔔

rfcbot added the **final-comment-period** label on 16 Feb

rfcbot removed the **proposed-final-comment-period** label on 16 Feb

rfcbot commented on 26 Feb

The final comment period is now complete.

🎉 10

PlasmaPower commented on 16 Mar — Contributor

This should be stabilized, right?

👍 2

mark-i-m commented on 17 Mar — Contributor

I would like to give stabilizing a feature a try :)

Does anyone know what the difference between the `i128` feature and the `i128_type` feature is?

PlasmaPower commented on 17 Mar • edited Contributor

I've always used `i128_type`, but all the docs refer to `i128`. I think they might be aliases.

SimonSapin commented on 17 Mar Contributor

They're likely the language features (the primitive types) vs the library features (various APIs that involve those types).

PlasmaPower commented on 17 Mar Contributor

I thought so too, but both the i128 primitive and the std::i128 module pages both have `i128` listed as the feature gate in the documentation.

cuviper commented on 17 Mar Member

Yes, `i128_type` is the language feature, and `i128` is for the library implementations. The latter includes all of the inherent methods you find on the primitive page.

PlasmaPower commented on 17 Mar Contributor

Oh, that's a bit confusing. Is there a reason for their separation? I'm assuming we want to stabilize both?

durka commented on 17 Mar • edited Contributor

@PlasmaPower lang and lib features are always separate -- lang features are listed in `src/libsyntax/feature_gate.rs` and checked at various places in the compiler, while lib features are simply scraped from `#[unstable]` attributes in the code.

There's a third feature, `repr128`, that also points to this tracking issue. It covers `#[repr(u128)]` enums, which cause problems if you look too closely at their discriminants. This is just a note to whomever does the stabilization PR (**@mark-i-m**), to keep this issue open or make a new one for `repr128`.

mark-i-m commented on 17 Mar Contributor

@durka Thanks! I will stabilize `i128` and `i128_type` and *not* `repr128`.

👍 3

mark-i-m commented on 17 Mar Contributor

So do I make a PR for the book and reference before I make PR for the feature gate?

mark-i-m referenced this issue on 17 Mar

Stabilize 128-bit integers :tada: #49101 Merged

2 of 2 tasks complete

est31 commented on 17 Mar — Contributor

@**durka** in fact, this separation seems to have been caused by tidy behaviour. It is relaxed since some time already: #43247 So the separation is more of a historic artifact.

SimonSapin commented on 17 Mar — Contributor

Docs for the primitive type are at https://github.com/rust-lang/rust/blob/1.24.1/src/libstd/primitive_docs.rs#L766-L776. They also have a "library" `#[unstable]` attributes. As far as I know language feature flags don't show up in rustdoc at all.

This was referenced on 18 Mar

- **Add 128-bit to types** rust-lang-nursery/reference#273 — Merged
- **Add 128-bit ints to ch3** rust-lang/book#1230 — Merged
- **i128 is being stabilized** rust-lang-nursery/compiler-builtins#235 — Merged
- **stable_features allowed temporarily** rust-lang-nursery/compiler-builtins#236 — Merged

alexcrichton added a commit to alexcrichton/rust that referenced this issue on 23 Mar

`Rollup merge of #49101 - mark-i-m:stabilize_i128, r=nagisa` ··· Verified ✗ `20b5bf6`

kennytm added a commit to kennytm/rust that referenced this issue on 24 Mar

`Rollup merge of #49101 - mark-i-m:stabilize_i128, r=nagisa` ··· Verified `ef563a6`

bors added a commit that referenced this issue on 24 Mar

`Auto merge of #49101 - mark-i-m:stabilize_i128, r=nagisa` ··· ✗ `8d41c9f`

bors added a commit that referenced this issue on 25 Mar

`Auto merge of #49101 - mark-i-m:stabilize_i128, r=nagisa` ··· ✗ `97f3424`

bors added a commit that referenced this issue on 26 Mar

`Auto merge of #49101 - mark-i-m:stabilize_i128, r=nagisa` ··· ✓ `188e693`

mark-i-m commented on 27 Mar — Contributor

I think this is done 🎉

🎉 1 ❤️ 1

est31 commented on 27 Mar — Contributor

@**mark-i-m** congrats!

Just don't close this issue yet as like @**durka** pointed out, the `repr128` feature is still pointing to this tracking issue.

👍 1 ❤️ 1

mark-i-m commented on 27 Mar — Contributor

Could someone update the OP too create more check boxes for repr128?

ebfull commented on 29 Mar — Contributor

The checkbox for #45676 is checked in this issue but #45676 is not actually closed. What's the status of lowering on other platforms?

nagisa commented on 29 Mar — Contributor

I *think* that is fixed, although not in the nicest way. I closed the issue.

est31 commented on 29 Mar — Contributor

@ebfull see my comment here: #35118 (comment)

i128 works on the entire x86 family as well as the ARM family. This covers all of the tier 1 platforms.
The platforms where we lack i128 support are tier 2 or 3.

What you are doing in ebfull/pairing#80 is exactly what I wanted to prevent by blocking stabilisation until all platforms support i128 lol, but people thought otherwise.

ebfull commented on 30 Mar — Contributor

@est31 I'm not doing ebfull/pairing#80 *until* it works on all platforms, which I mention in that issue. In the mean time all I will do is remove the `i128_type` feature flag, but still require users to opt into usage of `u128`.

est31 commented on 31 Mar — Contributor

@ebfull not blaming you. I'm partially blaming myself, thought that lowering worked already. Partially I'm blaming those people who decided to stabilize before all platforms support it. And the backend vendors who refuse to take {u,i}128 seriously.

Sadly, this is more of a "stable beta" release of i128 than an arrival of a feature that can be relied upon.

👍 1

hdevalence commented on 2 Apr

It would be nice if there was a (documented) way to set a cargo feature as the default on a given architecture. From what I can tell this isn't quite possible, since the architecture-selection code happens with `#[cfg(...)]`s while the crate is being compiled, by which point the crate's features are already selected. Am I missing something?

The motivation is that even if `u128`s are available, it may not be desirable to use them, unless they actually correspond to instructions available on that platform.

durka commented on 2 Apr — Contributor

You can set cfg flags from the build script.
...

hdevalence referenced this issue in **dalek-cryptography/curve25519-dalek** on 6 Apr

Use the u64 backend by default on x86_64 #126 Open

0 of 3 tasks complete

dtolnay referenced this issue in **serde-rs/serde** on 17 May

i128 and u128 integers missing Deserialize impls #1136 Closed

Centril added **disposition-merge** **finished-final-comment-period** and removed **final-comment-period** labels 26 days ago

kngwyu referenced this issue in **PyO3/pyo3** 17 days ago

Add 128bit integer support #173 Merged