# `lang_items`

The tracking issue for this feature is: None.

---

The `rustc` compiler has certain pluggable operations, that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute `#[lang = "..."]` and there are various different values of `...`, i.e. various different 'lang items'.

For example, `Box` pointers require two lang items, one for allocation and one for deallocation. A freestanding program that uses the `Box` sugar for dynamic allocations via `malloc` and `free`:

```
#![feature(lang_items, box_syntax, start, libc, core_intrinsics, rustc_private)]
#![no_std]
use core::intrinsics;
use core::panic::PanicInfo;

extern crate libc;

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang = "exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // Check if `malloc` failed:
    if p as usize == 0 {
        intrinsics::abort();
    }

    p
}

#[lang = "box_free"]
unsafe fn box_free<T: ?Sized>(ptr: *mut T) {
    libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(_argc: isize, _argv: *const *const u8) -> isize {
    let _x = box 1;

    0
}

#[lang = "eh_personality"] extern fn rust_eh_personality() {}
#[lang = "panic_impl"] extern fn rust_begin_panic(info: &PanicInfo) -> ! { unsafe {
intrinsics::abort() } }
#[no_mangle] pub extern fn rust_eh_register_frames () {}
#[no_mangle] pub extern fn rust_eh_unregister_frames () {}
```

Note the use of `abort`: the `exchange_malloc` lang item is assumed to return a valid pointer, and so needs to do the check internally.

Other features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the `==`, `<`, dereferencing (`*`) and `+` (etc.) operators are all marked with lang items; those specific four are `eq`, `ord`, `deref`, and `add` respectively.
- stack unwinding and general failure; the `eh_personality`, `panic` and `panic_bounds_check` lang items.
- the traits in `std::marker` used to indicate types of various kinds; lang items `send`, `sync` and `copy`.
- the marker types and variance indicators found in `std::marker`; lang items `covariant_type`, `contravariant_lifetime`, etc.

Lang items are loaded lazily by the compiler; e.g. if one never uses `Box` then there is no need to define functions for `exchange_malloc` and `box_free`. `rustc` will emit an error when an item is needed but not found in the current crate or any that it depends on.

Most lang items are defined by `libcore`, but if you're trying to build an executable without the standard library, you'll run into the need for lang items. The rest of this page focuses on this use-case, even though lang items are a bit broader than that.

### Using libc

In order to build a `#[no_std]` executable we will need libc as a dependency. We can specify this using our `Cargo.toml` file:

```
[dependencies]
libc = { version = "0.2.14", default-features = false }
```

Note that the default features have been disabled. This is a critical step - **the default features of libc include the standard library and so must be disabled.**

### Writing an executable without stdlib

Controlling the entry point is possible in two ways: the `#[start]` attribute, or overriding the default shim for the C `main` function with your own.

The function marked `#[start]` is passed the command line parameters in the same format as C:

```
#![feature(lang_items, core_intrinsics, rustc_private)]
#![feature(start)]
#![no_std]
use core::intrinsics;
use core::panic::PanicInfo;

// Pull in the system libc library for what crt0.o likely requires.
extern crate libc;

// Entry point for this program.
#[start]
```

```
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "eh_personality"]
#[no_mangle]
pub extern fn rust_eh_personality() {
}

#[lang = "panic_impl"]
#[no_mangle]
pub extern fn rust_begin_panic(info: &PanicInfo) -> ! {
    unsafe { intrinsics::abort() }
}
```

To override the compiler-inserted `main` shim, one has to disable it with `#![no_main]` and then create the appropriate symbol with the correct ABI and the correct name, which requires overriding the compiler's name mangling too:

```
#![feature(lang_items, core_intrinsics, rustc_private)]
#![feature(start)]
#![no_std]
#![no_main]
use core::intrinsics;
use core::panic::PanicInfo;

// Pull in the system libc library for what crt0.o likely requires.
extern crate libc;

// Entry point for this program.
#[no_mangle] // ensure that this symbol is called `main` in the output
pub extern fn main(_argc: i32, _argv: *const *const u8) -> i32 {
    0
}

// These functions are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "eh_personality"]
#[no_mangle]
pub extern fn rust_eh_personality() {
}

#[lang = "panic_impl"]
#[no_mangle]
pub extern fn rust_begin_panic(info: &PanicInfo) -> ! {
    unsafe { intrinsics::abort() }
}
```

In many cases, you may need to manually link to the `compiler_builtins` crate when building a `no_std` binary. You may observe this via linker error messages such as "`undefined reference to `__rust_probestack'`".

## More about the language items

The compiler currently makes a few assumptions about symbols which are available in the executable to call. Normally these functions are provided by the standard library, but without it you must define your own. These symbols are called "language items", and they each have an internal name, and then a signature that an implementation must conform to.

The first of these functions, `rust_eh_personality`, is used by the failure mechanisms of the compiler. This is often mapped to GCC's personality function (see the libstd implementation for more information), but crates which do not trigger a panic can be assured that this function is never called. The language item's name is `eh_personality`.

The second function, `rust_begin_panic`, is also used by the failure mechanisms of the compiler. When a panic happens, this controls the message that's displayed on the screen. While the language item's name is `panic_impl`, the symbol name is `rust_begin_panic`.

Finally, a `eh_catch_typeinfo` static is needed for certain targets which implement Rust panics on top of C++ exceptions.

## List of all language items

This is a list of all language items in Rust along with where they are located in the source code.

- Primitives
  - `i8`: `libcore/num/mod.rs`
  - `i16`: `libcore/num/mod.rs`
  - `i32`: `libcore/num/mod.rs`
  - `i64`: `libcore/num/mod.rs`
  - `i128`: `libcore/num/mod.rs`
  - `isize`: `libcore/num/mod.rs`
  - `u8`: `libcore/num/mod.rs`
  - `u16`: `libcore/num/mod.rs`
  - `u32`: `libcore/num/mod.rs`
  - `u64`: `libcore/num/mod.rs`
  - `u128`: `libcore/num/mod.rs`
  - `usize`: `libcore/num/mod.rs`
  - `f32`: `libstd/f32.rs`
  - `f64`: `libstd/f64.rs`
  - `char`: `libcore/char.rs`
  - `slice`: `liballoc/slice.rs`
  - `str`: `liballoc/str.rs`
  - `const_ptr`: `libcore/ptr.rs`
  - `mut_ptr`: `libcore/ptr.rs`
  - `unsafe_cell`: `libcore/cell.rs`
- Runtime

    - `start`: `libstd/rt.rs`
    - `eh_personality`: `libpanic_unwind/emcc.rs` (EMCC)
    - `eh_personality`: `libpanic_unwind/gcc.rs` (GNU)
    - `eh_personality`: `libpanic_unwind/seh.rs` (SEH)
    - `eh_catch_typeinfo`: `libpanic_unwind/emcc.rs` (EMCC)
    - `panic`: `libcore/panicking.rs`
    - `panic_bounds_check`: `libcore/panicking.rs`
    - `panic_impl`: `libcore/panicking.rs`
    - `panic_impl`: `libstd/panicking.rs`
- Allocations
    - `owned_box`: `liballoc/boxed.rs`
    - `exchange_malloc`: `liballoc/heap.rs`
    - `box_free`: `liballoc/heap.rs`
- Operands
    - `not`: `libcore/ops/bit.rs`
    - `bitand`: `libcore/ops/bit.rs`
    - `bitor`: `libcore/ops/bit.rs`
    - `bitxor`: `libcore/ops/bit.rs`
    - `shl`: `libcore/ops/bit.rs`
    - `shr`: `libcore/ops/bit.rs`
    - `bitand_assign`: `libcore/ops/bit.rs`
    - `bitor_assign`: `libcore/ops/bit.rs`
    - `bitxor_assign`: `libcore/ops/bit.rs`
    - `shl_assign`: `libcore/ops/bit.rs`
    - `shr_assign`: `libcore/ops/bit.rs`
    - `deref`: `libcore/ops/deref.rs`
    - `deref_mut`: `libcore/ops/deref.rs`
    - `index`: `libcore/ops/index.rs`
    - `index_mut`: `libcore/ops/index.rs`
    - `add`: `libcore/ops/arith.rs`
    - `sub`: `libcore/ops/arith.rs`
    - `mul`: `libcore/ops/arith.rs`
    - `div`: `libcore/ops/arith.rs`
    - `rem`: `libcore/ops/arith.rs`
    - `neg`: `libcore/ops/arith.rs`
    - `add_assign`: `libcore/ops/arith.rs`
    - `sub_assign`: `libcore/ops/arith.rs`
    - `mul_assign`: `libcore/ops/arith.rs`
    - `div_assign`: `libcore/ops/arith.rs`
    - `rem_assign`: `libcore/ops/arith.rs`
    - `eq`: `libcore/cmp.rs`
    - `ord`: `libcore/cmp.rs`
- Functions
    - `fn`: `libcore/ops/function.rs`
    - `fn_mut`: `libcore/ops/function.rs`
    - `fn_once`: `libcore/ops/function.rs`

    - `generator_state`：`libcore/ops/generator.rs`
    - `generator`：`libcore/ops/generator.rs`
- Other
    - `coerce_unsized`：`libcore/ops/unsize.rs`
    - `drop`：`libcore/ops/drop.rs`
    - `drop_in_place`：`libcore/ptr.rs`
    - `clone`：`libcore/clone.rs`
    - `copy`：`libcore/marker.rs`
    - `send`：`libcore/marker.rs`
    - `sized`：`libcore/marker.rs`
    - `unsize`：`libcore/marker.rs`
    - `sync`：`libcore/marker.rs`
    - `phantom_data`：`libcore/marker.rs`
    - `discriminant_kind`：`libcore/marker.rs`
    - `freeze`：`libcore/marker.rs`
    - `debug_trait`：`libcore/fmt/mod.rs`
    - `non_zero`：`libcore/nonzero.rs`
    - `arc`：`liballoc/sync.rs`
    - `rc`：`liballoc/rc.rs`