한 번에 끝내는 블록체인 개발 A to Z

Chapter 4

How to Develop Solana Contract?

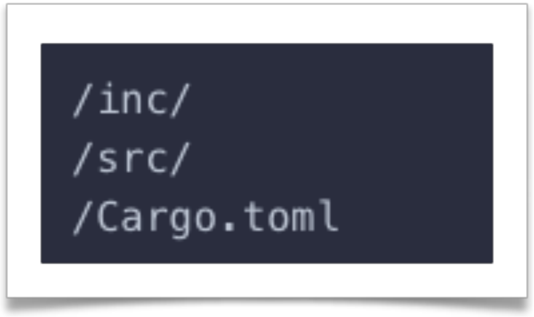
Chapter 4 How to Develop Solana Contract?

Developing with Rust

Project Layout

Solana supports writing on-chain programs using the Rust programming language.

Solana Rust programs may depend directly on each other in order to gain access to instruction helpers when making cross-program invocations. When doing so it's important to not pull in the dependent program's entrypoint symbols because they may conflict with the program's own. To avoid this, programs should define an **no-entrypoint** feature in **Cargo.toml** and use to exclude the entrypoint.



```
/inc/
/src/
/Cargo.toml
```

- Define the feature
- Exclude the entrypoint
- Include without entrypoint

Project Dependencies

At a minimum, Solana Rust programs must pull in the solana-program crate.

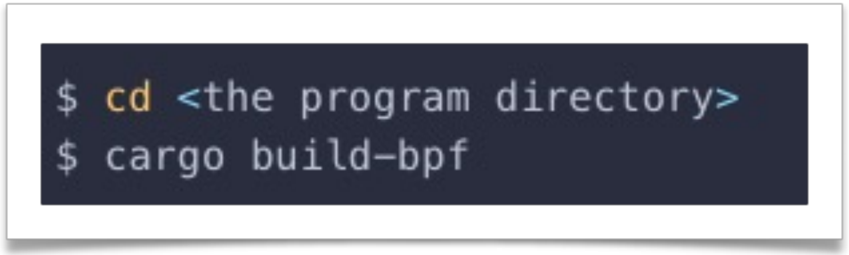
Solana BPF programs have some restrictions that may prevent the inclusion of some crates as dependencies or require special handling.

- Crates that requires the architecture be a subset
- Crates may depend on rand
- Crates may overflow the Stack (ex, large program size)

How to Build

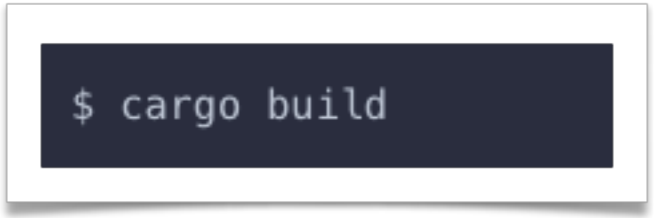
Env setup

- Install the latest Rust stable from https://rustup.rs/
- Install the latest Solana cmd tools from https://docs.solana.com/cli/install-solana-cli-tools



```
$ cd <the program directory>
$ cargo build-bpf
```

for the Solana BPF target which can be deployed to the cluster



```
$ cargo build
```

building programs against your host machine which can be used for unit testing

How to Test

Solana programs can be unit tested via the traditional **cargo test** mechanism by exercising program functions directly.

To help facilitate testing in an environment that more closely matches a live cluster, developers can use the **program-test crate**. The **program-test crate** starts up a local instance of the runtime and allows tests to send multiple transactions while keeping state for the duration of the test.

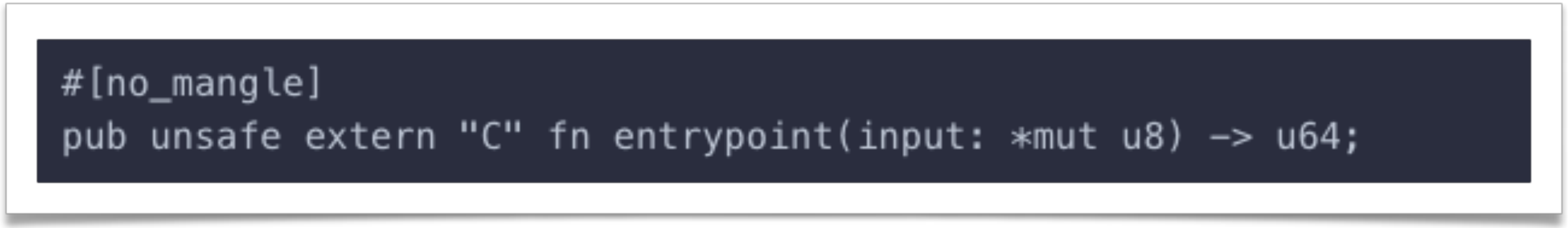
Example

Program Entrypoint

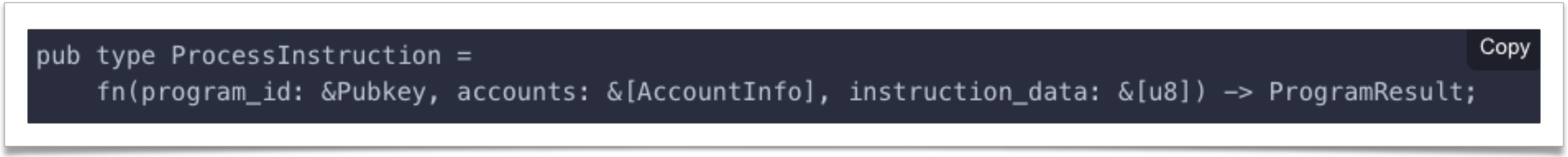
Programs export a known **entrypoint** symbol which the Solana runtime looks up and calls when invoking a program. This entrypoint takes a generic byte array which contains the serialized program parameters (program id, accounts, instruction data, etc…).

Example

*Mangling is when a compiler changes the name we've given a function to a different name that contains more information for other parts of the compilation process to consume but is less human readable.



```
#[no_mangle]
pub unsafe extern "C" fn entrypoint(input: *mut u8) -> u64;
```



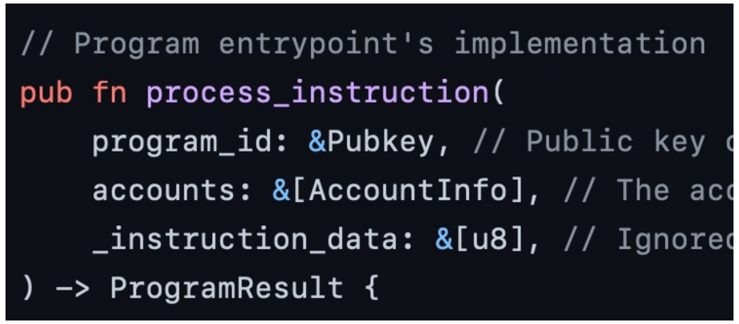
```
pub type ProcessInstruction =
    fn(program_id: &Pubkey, accounts: &[AccountInfo], instruction_data: &[u8]) -> ProgramResult;
```

Parameter Deserialization

Each loader provides a helper function that deserializes the program's input parameters into Rust types. The entrypoint macros automatically calls the deserialization helper.

Some programs may want to perform deserialization themselves and they can by providing their own implementation of the raw entrypoint. If a program implements their own deserialization function they need to ensure that any modifications the program wishes to commit be written back into the input byte array.

Details.

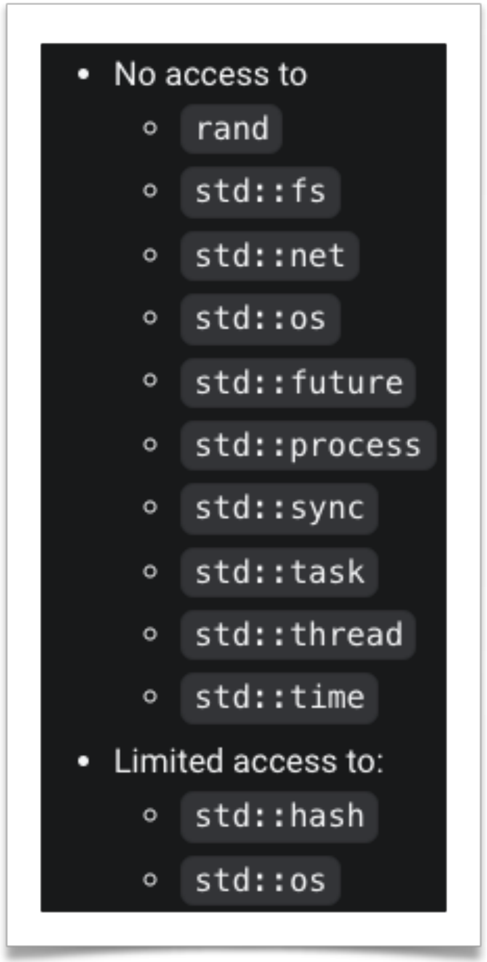


```
// Program entrypoint's implementation
pub fn process_instruction(
    program_id: &Pubkey, // Public key
    accounts: &[AccountInfo], // The acc
    _instruction_data: &[u8], // Ignore
) -> ProgramResult {
```

Restrictions

There are some limitations since these programs run in a resource-constrained, single-threaded environment, and must be deterministic:

- Bincode is extremely computationally expensive in both cycles and call depth and should be avoided
- String formatting should be avoided since it is also computationally expensive.
- No support for **println!**, **print!**, the Solana logging helpers should be used instead.
- The runtime enforces a limit on the number of instructions a program can execute during the processing of one instruction.



- No access to
  - `rand`
  - `std::fs`
  - `std::net`
  - `std::os`
  - `std::future`
  - `std::process`
  - `std::sync`
  - `std::task`
  - `std::thread`
  - `std::time`
- Limited access to:
  - `std::hash`
  - `std::os`