THE COIN · SMART CONTRACTS

# TCCL Cookbook

The Coin Cloud Language — write, test and deploy smart contracts on The Coin

### Inside this book

- Why TCCL is strict — and why that keeps your money safe
- Installing `tccl` and your first contract in 10 minutes
- A complete tour of the language
- Storage deposits, fuel and fees, with real numbers
- Deploying and calling contracts on the network
- Nine tested recipes: tokens, escrow, crowdfunding, polls, savings, treasuries, names and private payments
- A security checklist and a full language reference

```
# The smallest useful contract: a counter anyone can increase.
contract Counter

state count: int
state last_caller: address

event Increased(by: address, amount: int, total: int)

action increment(amount: int):
    require amount > 0, "amount must be positive"
    require amount <= 100, "at most 100 per call"
    count += amount
    last_caller = caller
    emit Increased(caller, amount, count)

view get() -> int:
    return count

view last() -> address:
    return last_caller
```

# Contents

# How to use this book

This cookbook teaches **TCCL — The Coin Cloud Language**, the smart-contract language of The Coin. It is written for developers who have programmed before in any language (Python, JavaScript, Rust, Go…) but have never written a smart contract. If you know a little Python you will feel at home immediately: TCCL uses indentation for blocks, `#` for comments and reads almost like pseudocode.

TCCL is intentionally **small and strict**. Almost everything that could be ambiguous must be written out: the type of every variable, which functions accept money, which functions may change state. The compiler refuses anything it cannot prove to be well-formed — and on The Coin the compiler is part of the consensus rules, so a contract that compiles on your laptop compiles identically on every node of the network.

The book is organised in four parts:

- **Getting started** (chapters 1–3): what TCCL is, how to install the tools and how to write, check and run a first contract in the local simulator.
- **The language** (chapters 4–6): a complete tour of the language, how contract storage and its refundable deposit work, and how fuel and fees are computed.
- **Building real contracts** (chapters 7–10): deploying to the network, nine complete recipes, a security checklist and a testing workflow.
- **Reference** (appendices): grammar, keywords, built-in functions, limits, error messages and the command-line tools.

> **NOTE**
> **Everything in this book is tested.** Every complete contract is a file in `crates/tccl/examples/` of the The Coin repository and is compiled and exercised by `cargo test -p tccl`. Every terminal transcript is real output of `tccl` 0.2.0 (language version 1), `thecoind` and `thecoin-wallet`, occasionally trimmed. Network transcripts were recorded on a private **regtest** network, which is why addresses start with `tcr1` instead of `tc1`.

The simple contracts in the first chapters can be written after an afternoon of reading. Contracts that hold other people's money deserve much more: read chapters 5, 6 and 9 carefully, write tests for every failure path, and have someone else review your code before you deploy. **A deployed contract cannot be changed.**

CHAPTER 1

# Why TCCL

## 1.1 A language for money

A smart contract is a small program that lives on the blockchain. It has an address, a balance of TCN and its own storage. Anyone can call it by sending a transaction, and every node of the network runs the call and must reach **exactly the same result**. Contracts escrow payments, issue tokens, run polls, pool funds privately — whatever rules their authors write down.

Those rules are enforced by code, not by people. When the code has a bug, nobody can step in to reverse a payment. That single fact shaped every decision in TCCL: the language makes the common mistakes of smart-contract development impossible or loud, keeps execution cheap enough for the modest machines that validate The Coin, and stays small enough to learn in a weekend.

## 1.2 Design principles

| Principle | What it means in practice |
|---|---|
| Strict static types | Every constant, state variable, parameter and local variable declares its type (`let total: int = 0`). Every expression has exactly one type and there are **no implicit conversions**: `text` never silently becomes `bytes`, `bool` never becomes `int`. |
| No floating point | Amounts are integers in **motes** (1 TCN = 100 000 000 motes). `int` is a 128-bit signed integer, big enough for any amount, with checked arithmetic: overflow and division by zero abort the call instead of wrapping around. |
| No null | Every type has a default value (`0`, `false`, `""`, empty bytes, the zero address, the empty list). Reading a map key that was never written returns the default — there is nothing to dereference and crash on. |
| Bounded execution | Every statement, expression, storage access and cryptographic operation consumes **fuel**. A call that runs out of fuel stops and is reverted. Values, lists, call depth and nesting are bounded, so no contract can exhaust a node. |
| Deterministic | No clock, no randomness, no network, no files, no floating point. The same call on the same state gives the same result on every node, today and in ten years. |
| Explicit permissions | Only `payable` functions accept TCN. `view` functions provably cannot change state, send coins or emit events — the compiler checks this transitively through helper functions. |
| All-or-nothing calls | If anything fails — a `require`, an overflow, running out of fuel — every change the call made is reverted: storage, transfers, events and the TCN sent with the call. |
| The compiler is consensus | A deployment transaction carries the **source code**. Every node compiles it with the same compiler; the source is stored in the blockchain and anyone can read and verify it. |

## 1.3 What TCCL deliberately leaves out

Many features common in general-purpose languages are missing on purpose. Each omission removes a whole class of bugs or attacks.

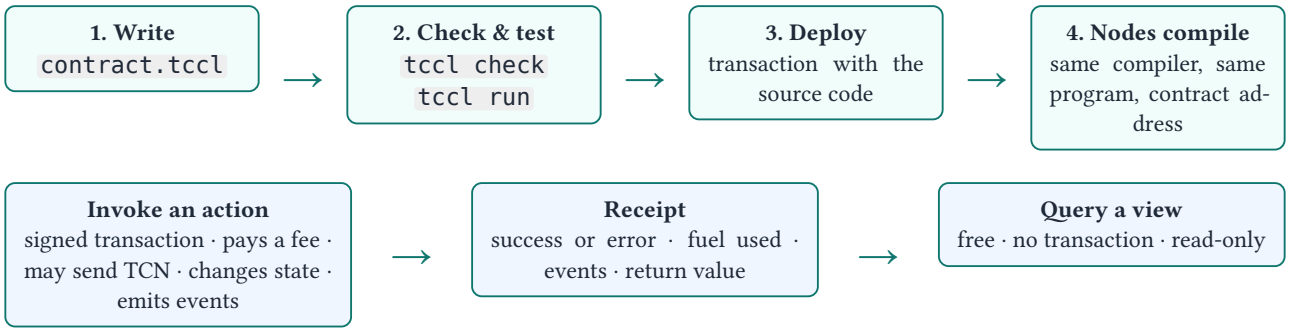
| Not in TCCL | Why |
|---|---|
| Floats and decimals | Rounding differences between machines would break consensus; money must be exact. |
| `null`, `None`, exceptions | Defaults and `require` cover every case with one simple rule: fail and revert. |
| Calls to other contracts | No callbacks means no re-entrancy attacks. `send` only moves TCN; it never runs code. |

| Not in TCCL | Why |
|---|---|
| Dynamic code, `eval`, imports | What you deploy is exactly what runs. Every contract is one self-contained file. |
| Classes, inheritance, closures | Less hidden control flow. Behaviour is visible in one place. |
| Randomness and time of day | Impossible to make deterministic and fair; use block `height` and commit–reveal schemes. |
| Implicit conversions | `"5" + 5` is a compile error, not a surprise. |
| Shadowing and redeclaration | A name means one thing inside a function; built-in names cannot be reused. |

## 1.4 Privacy through contracts

The Coin's base layer is transparent: balances and transfers are public, like Bitcoin. Privacy is **not** a special feature of the protocol — it is something developers build with TCCL. The language provides `ring_verify`, a verifier for linkable ring signatures, which proves "one of these N people authorised this" without revealing who. Chapter 8 shows a complete private payments pool built on it, and the reference wallet can use any pool that follows the same interface. Developers are free to design their own privacy contracts with different denominations and rules.

## 1.5 The life of a contract




Listing 1: From source file to running contract.

1. You write a `.tccl` file. One file is one contract.
2. You check it with `tccl check` and exercise it in the local simulator with `tccl run`.
3. You deploy it with `thecoin-wallet contract deploy`. The transaction contains the source code, the arguments for `init` and optionally some TCN.
4. Every node compiles the source, runs `init`, and stores the compiled program at an address derived from your address and transaction nonce.
5. Users call **actions** with transactions (paying a fee) and query **views** for free through any node's API. Every action produces a **receipt** with its result, fuel used and events.

CHAPTER 2

# Installing the tools

You need three programs. All of them are part of The Coin release:

| Program | Role |
|---|---|
| `tccl` | The TCCL developer tool: compile and check contracts, print their interface, run them in a local simulator, and create ring-signature keys for privacy contracts. Works offline. |
| `thecoin-wallet` | The reference wallet: deploys contracts, invokes actions, queries views and reads receipts. It keeps your keys locally and talks to a node over HTTP. |
| `thecoind` | The full node. The wallet needs the API of a node — your own or one you trust. |

## 2.1 With the node installer (Linux)

The one-line installer sets up a full node, a wallet and the developer tool on any Linux server with systemd (x86-64 or ARM64):

```
$ curl -fsSL https://the-coin.cloud/install.sh | sudo bash
```

It downloads the release archive, verifies its SHA-256 checksum (or builds from source when no binary is available for your machine) and installs `thecoind`, `thecoin-wallet` and `tccl` into `/usr/local/bin`. Useful options, passed after `bash -s --`:

| Option | Effect |
|---|---|
| `--yes` | Non-interactive: accept all defaults. |
| `--network testnet` | Install a **testnet** node — the right place to try contracts with worthless coins. |
| `--no-mine` | Run the node without mining. |
| `--from-source` | Build the binaries from the GitHub sources instead of downloading them. |

For example, a non-mining testnet node for development:

```
$ curl -fsSL https://the-coin.cloud/install.sh | sudo bash -s -- --network testnet --no-mine --yes
```

The installer creates a wallet at `~/.thecoin/wallet-<network>.json`, which is also the default location used by `thecoin-wallet`.

## 2.2 From source (any platform)

`tccl` is an ordinary Rust program with no system dependencies. With a Rust toolchain installed (rustup.rs):

```
$ git clone https://github.com/LucasBolla94/thecoin.git
$ cd thecoin
$ cargo build --release -p tccl -p thecoin-wallet -p thecoin-node
```

The three binaries are now in `target/release/`; copy them to a directory in your `PATH`.

## 2.3 Checking the installation

```
$ tccl --version
tccl 0.2.0 (language version 1)
```

Running `tccl` without arguments prints its help:

```
$ tccl
tccl — The Coin Cloud Language tool

USAGE:
  tccl check <file.tccl>                Compile and show the contract interface
  tccl abi <file.tccl>                  Print the interface as JSON
  tccl run <file.tccl> [options] deploy [args...]
  tccl run <file.tccl> [options] call <function> [args...]
  tccl run <file.tccl> [options] view <function> [args...]
  tccl ring keygen [--seed <hex32>]     Create a ring (privacy) key pair
  tccl ring sign --secret <hex> --ring <pk1,pk2,...> --index <i> --message <0xhex>

RUN OPTIONS:
  --state <file>     Simulator state file (default: tccl-state.json)
  --from <name>      Test account calling (default: alice); every account starts with 1 000 000 TCN
  --value <amount>   TCN sent with the call, e.g. 5tcn or 500000000 (motes)
  --height <n>       Set the simulated block height before the call
  --contract <hex>   Contract address in the simulator (default: the last deployed)

Arguments are parsed using the declared parameter types:
  int 42 | 2.5tcn    bool true    text "hello"    bytes 0xabcd    address tc1...    list [1, 2]
Account names are accepted for address parameters: @alice, @bob ...
```

## 2.4 Editor setup

TCCL source files use the `.tccl` extension and must be UTF-8. Configure your editor to:

- indent with **spaces** — a tab anywhere in the file is a compile error (`tabs are not allowed; indent with spaces`);
- use 4 spaces per level (any consistent amount works, 4 is the convention);
- highlight `.tccl` files as Python if there is no TCCL mode: the result is close enough.

> **NOTE**
> Contracts are limited to **48 000 bytes** of source code. That is a lot — the largest contract in this book has 3 327 bytes — but deploying costs fuel for every byte, comments included, so keep comments useful and short.

CHAPTER 3

# Your first contract

Let us write, check and run a contract before learning the details. Our first contract is a counter that anyone can increase, which remembers who increased it last.

## 3.1 The code

Create a file called `counter.tccl`:

```
# The smallest useful contract: a counter anyone can increase.
contract Counter

state count: int
state last_caller: address

event Increased(by: address, amount: int, total: int)

action increment(amount: int):
    require amount > 0, "amount must be positive"
    require amount <= 100, "at most 100 per call"
    count += amount
    last_caller = caller
    emit Increased(caller, amount, count)

view get() -> int:
    return count

view last() -> address:
    return last_caller
```

## 3.2 Line by line

**`# The smallest…`** A comment. Everything from `#` to the end of the line is ignored.

**`contract Counter`** Every file starts with the contract name. One file is one contract.

**`state count: int`** A **state variable**: stored on the blockchain and kept between calls. Its type is mandatory. It starts with the default value of its type, `0`.

**`state last_caller: address`** Another state variable. Addresses start as the zero address.

**`event Increased(…)`** Declares an **event**: a typed record that actions can emit. Events are stored in the transaction receipt so wallets, explorers and other programs can follow what happened.

**`action increment(amount: int):`** An **action** is a function that users call with a transaction. Actions may change state. Parameters always have types.

**`require amount > 0, "…"`** If the condition is false the call stops, **every change is reverted**, and the message becomes the error of the call.

**`count += amount`** Adds to the state variable. The arithmetic is checked: an overflow would abort the call.

**`last_caller = caller`** `caller` is the address that signed the transaction.

**`emit Increased(caller, amount, count)`** Records the event with its three fields.

**`view get() -> int:`** A **view** is a read-only function. Anyone can call it for free, without a transaction. It must declare its return type after `->`.

Notice what is **not** there: no constructor is needed (state starts at defaults), no visibility keywords (actions and views are public; `fn` helpers are private), no type inference.

## 3.3 Check it

`tccl check` compiles the contract exactly like the network would and prints its interface:

```
$ tccl check counter.tccl
✔ Counter compiles (484 bytes of source, 323 bytes compiled)
  state: count: int, last_caller: address
  action increment(amount: int)
  view get() -> int
  view last() -> address
```

## 3.4 Run it in the simulator

`tccl run` executes the contract in a local, in-memory blockchain. The simulator follows the same rules as the network: failed calls revert every change, TCN sent with a call is credited to the contract, and each successful call advances the block height by one. Its state is saved to `tccl-state.json` in the current directory, so you can run one command at a time.

Test accounts are named: `alice` (the default caller), `bob`, `carol` or any other name you pass with `--from`. Every test account starts with 1 000 000 TCN.

```
$ tccl run counter.tccl deploy
deployed Counter at daa436158c1dcdc0242085b6dd9451e33496cbee
ok · fuel used: 2420 · height: 2 · alice balance: 100000000000000 motes
$ tccl run counter.tccl call increment 5
event Increased(by: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, amount: 5, total: 5)
ok · fuel used: 1674 · height: 3 · alice balance: 100000000000000 motes
$ tccl run counter.tccl --from bob call increment 7
event Increased(by: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 7, total: 12)
ok · fuel used: 1674 · height: 4 · bob balance: 100000000000000 motes
$ tccl run counter.tccl view get
result: 12
ok · fuel used: 273 · height: 4 · alice balance: 100000000000000 motes
$ tccl run counter.tccl view last
result: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452
ok · fuel used: 273 · height: 4 · alice balance: 100000000000000 motes
```

Three things to notice:

- `deploy` used 2 420 units of **fuel**: 5 per byte of source code to compile the contract. `Counter` has no `init` function, so nothing else ran.
- Each `increment` used 1 674 fuel. On the real network fuel is paid for as part of the transaction fee (chapter 6). Views are free for the caller, but they still have a fuel limit.
- Balances are shown in **motes**: 100 000 000 000 000 motes = 1 000 000 TCN.

## 3.5 When a call fails

Call `increment` with a value above the limit:

```
$ tccl run counter.tccl call increment 500
FAILED: requirement failed: at most 100 per call · fuel used: 33 (all changes reverted)
```

The `require` stopped the call. Nothing changed: `get` still returns 12. On the network the wallet simulates every call before sending it and refuses to broadcast one that would fail, so you do not pay a fee for a mistake the wallet can see.

Compile errors are just as explicit. Each message has the file, line and column. Here are four typical first-day mistakes, each made in a copy of `counter.tccl` (the grey lines describe the change):

```
# line 12 changed to:  count += "amount"
$ tccl check broken1.tccl
error: broken1.tccl:line 12:14: expected int, found text
# line 13 changed to:  let last_caller = caller
$ tccl check broken2.tccl
error: broken2.tccl:line 13:21: expected ':' and a type (variables must declare their type), found
assign
# line 16 changed to:  view get():
$ tccl check broken3.tccl
error: broken3.tccl:line 16:1: a view must declare a return type ('-> type')
# "count = 0" added to the view last()
$ tccl check broken4.tccl
error: broken4.tccl:line 20:5: a view cannot change state
```

**TRY IT YOURSELF**

1. Add a view `average() -> int` that returns `count / calls`, with a new state variable `calls` increased by `increment`. What happens when you call it before any increment? (Division by zero aborts the call — add a `require` or an `if`.)
2. Add an action `reset()` that only the address stored in a new `owner` state variable may call. Set `owner = caller` inside an `init():` function.
3. Run `tccl abi counter.tccl` and look at the JSON interface that wallets and explorers use.

CHAPTER 4

# A tour of the language

This chapter covers the whole language. Keep it open while you write your first real contracts; Appendix A repeats the essentials as compact tables.

## 4.1 File layout

A TCCL file is a contract header followed by declarations, in any order. This small shop (`crates/tccl/examples/shop.tccl`) shows every kind of declaration:

```
contract Shop                       # 1. header: always the first line of code

const MAX_NAME: int = 64            # 2. constants
state owner: address                # 3. state variables
state prices: map[text, int]
event Sold(item: text, buyer: address, price: int)   # 4. events

init():                             # 5. functions: init, action, view, fn
    owner = caller

action set_price(item: text, price: int):
    require caller == owner, "only the owner"
    require valid_name(item), "item names have 1 to 64 bytes"
    require price >= 0, "price cannot be negative"
    prices[item] = price            # a price of 0 removes the item

action buy(item: text) payable:
    require prices.has(item), "unknown item"
    require value == prices[item], "wrong price"
    send(owner, value)
    emit Sold(item, caller, value)

view price_of(item: text) -> int:
    return prices[item]

fn valid_name(item: text) -> bool:
    return len(item) >= 1 and len(item) <= MAX_NAME
```

The rules of layout:

- **Indentation defines blocks**, like Python. A line ending with `:` opens a block that must be indented further; the block ends when the indentation returns. Use spaces only.
- **One statement per line.** There are no semicolons. Inside parentheses `( )` and brackets `[ ]` a statement may continue on the next lines.
- **Comments** start with `#` and run to the end of the line.
- **Names** start with a letter or `_` and continue with letters, digits or `_` (ASCII only). Names are case-sensitive.
- A contract needs **at least one** `init`, `action` or `view`.
- A name can be declared only once in the whole contract: a constant, a state variable, an event and a function cannot share a name, and local variables cannot reuse any of them.

> **NOTE**
> Functions may be declared in any order and may call helpers declared further down. Constants, however, are evaluated in order: a constant can only use constants declared **above** it.

### 4.1.1 Reserved names

Keywords cannot be used as names at all:

`contract` `const` `state` `event` `init` `action` `view` `fn` `payable` `let` `if` `elif` `else` `while` `for` `in` `break` `continue` `return` `require` `send` `emit` `destroy` `pass` `true` `false` `and` `or` `not`

In addition, these built-in names are **reserved**: they cannot be used for constants, state variables, events, functions, parameters or local variables.

`caller` `value` `balance` `height` `self` `TCN` `int` `bool` `text` `bytes` `address` `list` `map` `len` `sha256` `blake3` `to_bytes` `to_text` `to_int` `min` `max` `abs` `slice` `verify_ed25519` `ring_verify` `address_of` `zero_address` `range`

```
state balance: int     # error: 'balance' is a reserved name
```

## 4.2 Types

TCCL has seven types. Every value has exactly one of them.

Table 1: The types of TCCL.

| Type | Default | Example literal | Notes |
|---|---|---|---|
| `int` | `0` | `42`, `-7`, `1_000_000` | Signed 128-bit integer. Checked arithmetic. |
| `bool` | `false` | `true`, `false` | Only `and`, `or`, `not`, `==`, `!=`. |
| `text` | `""` | `"hello"` | UTF-8 string. `len` counts **bytes**. Escapes: `\n` `\t` `\"` `\\`. |
| `bytes` | empty | `0xdeadbeef` | Raw bytes, written as hex with an even number of digits. |
| `address` | zero address | `address("tc1…")` | A 20-byte account or contract address. |
| `list[T]` | `[]` | `[1, 2, 3]` | Ordered list of values of type `T` (any type except maps). |
| `map[K, V]` | — | — | Key → value table. **Only as a state variable.** `K` must be `int`, `bool`, `text`, `bytes` or `address`; `V` any type except a map. |

The first five are **scalar** types. Scalars can be compared with `==` and `!=`, used as map keys and given initial values. Lists and maps cannot be compared with `==`.

### 4.2.1 Integers and amounts

There is no decimal type. TCN amounts are integers in **motes**: 1 TCN = 100 000 000 motes. The built-in constant `TCN` is equal to `100_000_000`, so amounts read naturally:

```
const MIN_TIP: int = TCN / 100          # 0.01 TCN = 1 000 000 motes
const DEPOSIT: int = 10 * TCN           # 10 TCN
```

Underscores in numbers are ignored (`21_000_000`). A number cannot have a unit or a letter suffix: `5tcn` is a compile error in source code (`invalid number literal`) — write `5 * TCN`. The `5tcn` notation **is** accepted on the command line when passing arguments to `tccl run` and `thecoin-wallet`.

### 4.2.2 Addresses

Use the `address("…")` literal to write an address in source code. It is checked at compile time, and on the network it must use that network's prefix (`tc1` on mainnet, `tct1` on testnet, `tcr1` on regtest):

```
const TREASURY: address = address("tc1yfugpgq45fs8xe80x4jam2j2w2kqhk9qnp2umy")
```

`zero_address()` returns the all-zero address, which no one controls. It is the default value of `address` variables and a common "burn" or "none" marker.

## 4.3 Constants

```
const NAME: text = "Cloud Token"
const MAX_SUPPLY: int = 21_000_000 * TCN
const FEE_BP: int = 25                  # basis points
const YEAR: int = 365 * 24 * 60         # blocks, at one block per minute
const ENABLED: bool = not false
```

Constants must declare a type and are computed at compile time. Their value can only use literals, other constants declared above, arithmetic and comparison operators and `address("…")`. They cost no storage and almost no fuel: prefer them to state variables for anything that never changes.

## 4.4 State variables

State variables live in the contract's storage on the blockchain.

```
state owner: address                    # starts as the zero address
state paused: bool                      # starts as false
state name: text = "Cloud Token"        # scalar with an initial value
state holders: list[address]            # a list stored item by item
state balances: map[address, int]       # a map
state history: map[int, list[int]]      # a map whose values are lists
```

- Scalar state variables (`int`, `bool`, `text`, `bytes`, `address`) may have a constant initial value, written when the contract is deployed.
- Lists and maps cannot have initial values; they start empty.
- Whole lists and maps cannot be assigned or copied: work with their items (Section 4.10 and Section 4.11).
- A state variable that holds its type's default value takes **no storage at all**. Setting it back to the default deletes the storage entry (and refunds deposit, see chapter 5).

```
state fees: list[int] = []      # error: only int, bool, text, bytes and address state variables
can have an initial value
```

## 4.5 Functions

There are four kinds of functions:

Table 2: Function kinds.

| Kind | Purpose | Changes state | Payable | Called by |
|---|---|---|---|---|
| `init` | Runs once, at deployment. Optional. Takes arguments from the deploy transaction. | yes | allowed | the deployer |
| `action` | Public entry point, called by a transaction. | yes | allowed | anyone |
| `view` | Public read-only query. Must return a value. | **no** | no | anyone, free |
| `fn` | Private helper, callable only from code in the same contract. | yes | no | other functions |

The header of a function is: kind, name (except `init`), parameters with types, optional return type after `->`, optional `payable`, and a colon. The order matters: the return type comes **before** `payable`.

```
init():
    count = 0

action add(item: text) -> int payable:
    require value >= TCN, "listing costs 1 TCN"
    items[count] = item
    count += 1
    return count - 1

view get(id: int) -> text:
    return items[id]

fn is_valid(id: int) -> bool:
    return id >= 0 and id < count
```

Rules the compiler enforces:

- There can be at most one `init`, and it cannot return a value.
- A `view` must declare a return type. It cannot assign state, `send`, `emit`, `destroy` or read `value`, and it cannot call an `fn` that (directly or through other helpers) does any of those.
- Only `action` and `init` can be `payable`. Sending TCN to a function that is not payable fails with `function does not accept TCN (not payable)`.
- A function that declares a return type must **return on every path**: its last statement must be a `return`, or an `if`/`else` whose branches all end in a `return`.
- Entry points (`init`, `action`, `view`) cannot be called from code; move shared logic into an `fn`.
- Parameters are local variables: they can be reassigned inside the function.
- Actions may return a value too. It appears in the transaction receipt as `return_value`.

```
view sign(x: int) -> int:      # error: function 'sign' must return a int on every path
    if x > 0:
        return 1
    elif x < 0:
        return -1
```

Add an `else:` branch, or a final `return 0`, to fix it.

### 4.5.1 Recursion and call depth

Helpers may call each other and themselves, but the call stack is limited to **16 levels** (the entry point counts as the first). Deeper calls fail with `call depth limit reached`. Prefer loops.

## 4.6 Local variables and scope

Local variables are declared with `let`, a type and an initial value — all three are mandatory:

```
let fee: int = amount / 100
let net: int = amount - fee
let names: list[text] = []
let ok: bool = balances[who] >= amount
```

A variable is visible from its declaration to the end of the block that contains it. A name cannot be declared twice while it is visible — **there is no shadowing** — but two separate blocks may use the same name:

```
if x > 0:
    let label: text = "positive"
else:
    let label: text = "not positive"    # fine: the first 'label' is out of scope
```

```
let x: int = 5          # error: variable 'x' is already declared (x is a parameter)
```

Maps cannot be local variables; lists can.

## 4.7 Operators

Table 3: Operators from lowest to highest precedence.

| Precedence | Operators | Operand types → result |
|---|---|---|
| 1 (lowest) | `or` | `bool`, `bool` → `bool` (short-circuit) |
| 2 | `and` | `bool`, `bool` → `bool` (short-circuit) |
| 3 | `not` | `bool` → `bool` |
| 4 | `== !=` | two values of the **same scalar type** → `bool` |
| 4 | `< <= > >=` | `int`, `int` → `bool` |
| 5 | `+` | `int`+`int` → `int`; `text`+`text` → `text`; `bytes`+`bytes` → `bytes` |
| 5 | `-` | `int`, `int` → `int` |
| 6 | `* / %` | `int`, `int` → `int` |
| 7 | unary `-` | `int` → `int` |
| 8 (highest) | `x[i]`, `x.method(…)`, `f(…)` | indexing, methods and calls |

Things that surprise newcomers:

- **Checked arithmetic.** `+`, `-`, *, unary `-` and `abs` fail with `integer overflow` outside the 128-bit range. `/` and `%` by zero fail with `division by zero`.
- **Division truncates toward zero**: `-7 / 2` is `-3`, and `%` takes the sign of the left operand: `-7 % 2` is `-1`. For amounts, multiply first and divide last: `amount * 25 / 10_000`.
- **No chained comparisons.** `0 < x < 10` is a compile error; write `x > 0 and x < 10`.
- **`not` binds looser than comparisons**: `not a == b` means `not (a == b)`.
- **No `**`, `//`, bit operators or `+=` on booleans.** Compound assignments are `+=`, `-=` and `*=` for `int`, and `+=` for `text` and `bytes`.
- Text and bytes are joined with `+`; there is no string formatting. Use `to_text(n)` to turn an integer into text.
- At most 64 operators of the same precedence level can be chained in one expression, and an expression may be nested at most 128 levels deep. Split long formulas with `let`.

## 4.8 Control flow

### 4.8.1 if, elif, else

```
if amount >= 1_000 * TCN:
    level = "gold"
elif amount >= 100 * TCN:
    level = "silver"
else:
    level = "bronze"
```

Conditions must be `bool` — there is no "truthiness": `if count:` is an error, write `if count != 0:`.

### 4.8.2 while

```
let n: int = 1
while n < target:
    n *= 2
return n
```

### 4.8.3 for

`for` iterates over an integer range or over a list. `range(start, end)` counts from `start` up to `end - 1`; both arguments are required.

```
let total: int = 0
for i in range(0, len(prices)):
    total += prices[i]
for p in prices:
    total += p
return total
```

`break` leaves the innermost loop and `continue` jumps to its next iteration. `pass` is a statement that does nothing, for blocks that must not be empty.

> **FUEL TIP**
> Loops are allowed, but every iteration costs fuel, and a transaction can use at most 10 000 000 fuel. A loop over a list that **anyone can make longer** is a denial-of-service risk: one day it becomes too expensive to run and the function is unusable forever. Bound your lists with constants or process them in pages with `range(start, end)`. See chapter 9.

## 4.9 Statements that talk to the chain

### 4.9.1 require

```
require caller == owner, "only the owner"
require amount > 0            # without a message
```

If the condition is false, the call fails with `requirement failed: <message>` and all its effects are reverted. Without a message the error reads `requirement failed: requirement at line N failed`. The message may be any `text` expression. `require` is the main tool for validating input and permissions — use it generously, with messages that tell the user what to do.

### 4.9.2 send

```
send(owner, 5 * TCN)
```

`send(to, amount)` transfers `amount` motes from the contract's balance to `to`. It fails with `invalid amount` if the amount is zero or negative and with `insufficient contract balance` if the contract does not hold enough. Sending to an address never runs any code, even if that address is another contract: the receiver's balance simply increases.

### 4.9.3 emit

```
emit Paid(to, 2 * TCN, "invoice 42")
```

Arguments must match the event's fields in number and type. A call can emit at most 64 events on the network. Events are part of the transaction receipt, not of the contract's state: contracts cannot read past events.

### 4.9.4 destroy

```
require caller == owner, "only the owner"
destroy(owner)
```

`destroy(to)` removes the contract: its whole balance **and its storage deposit** are paid to `to`, its code and metadata are deleted, and later calls fail with `no contract at …`. It may only appear in an `action`, and it ends the call immediately. Scalar state variables are cleared automatically, but all lists must be popped and all map entries removed first — otherwise it fails with `contract cannot be destroyed while it still has storage (N entries)`.

## 4.10 Lists

A list holds values of one type. Lists come in two flavours that behave slightly differently:

- **Local lists** (variables, parameters, return values) live in memory. They are limited to 4 096 items and 65 536 bytes.
- **State lists** are stored item by item. They have no fixed size limit, but every read costs storage fuel.

| Operation | Local list | State list |
|---|---|---|
| Literal `[a, b, c]`, empty list `[]` (when the type is known) | yes | — |
| Read an item `xs[i]` | yes | yes |
| Replace an item `xs[i] = v`, `xs[i] += v` | yes | yes |
| Append `xs.push(v)` | yes | yes |
| Remove the last item `xs.pop()` (statement or expression) | no | yes |
| Length `len(xs)` or `xs.len()` | yes | yes |
| Iterate `for x in xs:` | yes | yes |
| Assign or copy the whole list | yes | no |

```
queue.push(who)
let first: address = queue[0]
let last: address = queue.pop()
let n: int = len(queue)
```

Indexes start at 0. An index outside the list fails with `index 5 out of bounds (length 3)`. Lists may contain lists (`list[list[int]]`); items of an inner list are read with `grid[i][j]`, but only outer items can be assigned.

To return a state list from a view, copy it into a local list:

```
let out: list[int] = []
for count in tally:
    out.push(count)
return out
```

## 4.11 Maps

Maps exist only as state variables. Reading a key that was never set returns the default value of the value type.

```
balances[to] += amount                  # missing keys read as 0
let mine: int = balances[caller]
if balances.has(to):
    pass
balances.remove(caller)                 # delete the entry
```

| Operation | Meaning |
|---|---|
| `m[k]` | Value for `k`, or the default value if the key is absent. |
| `m[k] = v`, `m[k] += v` | Set the value. Setting the **default** value deletes the entry. |
| `m.has(k)` | `true` if an entry for `k` is stored. |
| `m.remove(k)` | Delete the entry (a statement). |

**SECURITY TIP**

**A stored default is no entry at all.** Writing `0`, `false`, `""`, empty bytes, the zero address or an empty list deletes the map entry, so `has` returns `false` afterwards. That is exactly what you want for balances (and it frees storage), but it means a `map[address, bool]` can only remember `true`. If you need to distinguish "never set" from "set to zero", store a non-default marker or keep a separate `map[K, bool]` of known keys.

Maps cannot be iterated. If you need to list the keys, also keep them in a state list — and bound its length. Values of a map can be lists; to change one, read it into a local list, modify it and store it back:

```
let past: list[int] = history[caller]
past.push(amount)
history[caller] = past
```

**NOTE**

> The index of a compound assignment is evaluated **exactly once**: `counts[next_id()] += 1` calls `next_id()` one time and updates the entry it returned. Storing a computed key in a `let` first is still good style when you use it more than once.

## 4.12 Context values

Five read-only names describe the current call:

| Name | Type | Meaning |
|---|---|---|
| `caller` | `address` | The address that signed the transaction. In a view it is the zero address. |
| `value` | `int` | Motes sent with this call (0 if none). Not available in views. |
| `balance` | `int` | The contract's balance in motes, **including** the `value` of this call. |
| `height` | `int` | Height of the block that includes the call (for views: the next block). |
| `self` | `address` | The address of this contract. |

Time in TCCL is measured in blocks. The network targets one block per minute, so 60 blocks ≈ 1 hour, 1 440 ≈ 1 day, 10 080 ≈ 1 week and 525 600 ≈ 1 year. Block times vary; never promise exact dates.

## 4.13 Built-in functions

Table 4: Built-in functions. Every call also pays the normal expression fuel.

| Function | Description | Extra fuel |
|---|---|---|
| `len(x) -> int` | Length of a list (items), `text` or `bytes` (bytes). Also `x.len()`. | — |
| `min(a, b) -> int`<br>`max(a, b) -> int` | Smaller / larger of two integers. | — |
| `abs(a) -> int` | Absolute value (fails on overflow for the smallest `int`). | — |
| `to_text(n: int) -> text` | Decimal representation, e.g. `"-42"`. | — |
| `to_bytes(x) -> bytes` | `int`: 16 bytes big-endian two's complement; `address`: 20 bytes; `text`: UTF-8 bytes; `bool`: 1 byte (`0x01`/`0x00`); `bytes`: unchanged. | — |
| `to_int(b: bytes) -> int` | Unsigned big-endian integer from at most 15 bytes. | — |
| `slice(b: bytes, start, end) -> bytes` | Bytes `start` to `end - 1`. Fails if out of range. | — |
| `sha256(x) -> bytes` | SHA-256 of `bytes` or `text` (32 bytes). | 60 + 20 per 64 bytes |
| `blake3(x) -> bytes` | BLAKE3 of `bytes` or `text` (32 bytes). | 60 + 20 per 64 bytes |
| `verify_ed25519(pk, msg, sig) -> bool` | Checks an Ed25519 signature (32-byte key, 64-byte signature, strict verification). Returns `false` for malformed input. | 3 500 + 1 per 64 bytes |
| `ring_verify(ring, msg, sig, key_image) -> bool` | Checks a linkable ring signature: `ring` is a `list[bytes]` of 32-byte public keys (at most 64). See Section 8.9. | 5 000 + 10 000 per key |

| Function | Description | Extra fuel |
|---|---|---|
| `address_of(pk: bytes) -> address` | The address that corresponds to a 32-byte Ed25519 public key. | — |
| `zero_address() -> address` | The all-zero address. | — |
| `address("tc1…") -> address` | Compile-time address literal. | — |
| `range(start, end)` | Only in `for i in range(start, end):`. | 1 per iteration |

```
let msg: bytes = to_bytes(self) + to_bytes(caller) + to_bytes(amount)
require verify_ed25519(pk, blake3(msg), sig), "bad signature"
let owner_of_key: address = address_of(pk)
let short: bytes = slice(blake3(msg), 0, 8)
let n: int = to_int(short)
```

## 4.14 Events

Events are declared at the top level with typed fields and emitted from actions, `init` or helpers called by them. Field types can be any type except maps.

```
event Transfer(from: address, to: address, amount: int)
event Listed(id: int, tags: list[text])
```

Wallets show events from simulations before you send a transaction, and the receipt of every confirmed call lists them (Section 7.6). Design events for the people who will build interfaces and indexers on top of your contract: emit one for every change of ownership, balance or status.

## 4.15 Patterns for structured data

TCCL has no structs or classes. Two patterns cover almost every need.

**Parallel maps** store the fields of a record in separate maps that share a key:

```
state order_buyer: map[int, address]
state order_amount: map[int, int]
state order_paid: map[int, bool]
state order_count: int
```

**Composite keys** combine several values into one `bytes` key with `to_bytes`. Because every `address` is exactly 20 bytes and every `int` exactly 16 bytes, joining them cannot create ambiguous keys:

```
fn allowance_key(holder: address, spender: address) -> bytes:
    return to_bytes(holder) + to_bytes(spender)
```

> **SECURITY TIP**
> Joining variable-length values such as two `text` values **can** be ambiguous: `"ab" + "c"` and `"a" + "bc"` give the same key. Put a length or a separator that cannot appear in the data between them, or hash fixed-size parts.

CHAPTER 5

# State, storage and the deposit

Contract storage is kept by every node forever, so it is not free. The Coin asks the people who make a contract's state grow to lock a small, **refundable** deposit, and gives it back to whoever makes the state shrink. This chapter explains exactly how storage is measured and how the deposit moves.

## 5.1 How state is stored

The compiled program and every storage entry are separate records in the node's database. A contract's size, `state_bytes`, is the size of its compiled code plus the size of every entry (key and value). Entries are created as follows:

| What | Entries | Size of each entry |
|---|---|---|
| scalar state variable | 1, only if not default | 3-byte key + encoded value |
| map item | 1 per stored key | 3 bytes + encoded key + encoded value |
| state list | 1 for the length + 1 per item | length: 3 + 8 bytes; item: 11 bytes + encoded value |

Values are encoded compactly: one type byte, then 16 bytes for an `int`, 1 for a `bool`, 20 for an `address`, or 4 bytes of length plus the content for `text`, `bytes` and lists. For example, after one call to `increment`, the counter of chapter 3 holds two entries: `count` (3 + 17 = 20 bytes) and `last_caller` (3 + 21 = 24 bytes). Its 323 bytes of code plus 44 bytes of entries make 367 bytes, the `state_bytes` reported by the node in Section 7.8.

Remember the rule from chapter 4: **a default value is not stored**. Setting an `int` to `0`, a `bool` to `false` or removing a map key deletes the entry and makes the contract smaller.

## 5.2 The storage deposit

The deposit a contract must hold is

$$\textbf{required deposit} = \lceil \texttt{state\_bytes} \div 1\,000 \rceil \times \texttt{storage\_deposit\_per\_kb}$$

`storage_deposit_per_kb` is a governance parameter. Its default values are:

| Network | `storage_deposit_per_kb` | In TCN |
|---|---|---|
| mainnet | 100 000 motes | 0.001 TCN per started kB |
| testnet | 100 000 motes | 0.001 TCN per started kB |
| regtest | 10 000 motes | 0.0001 TCN per started kB |

The rules, applied after every successful deploy or action:

1. **Deploy.** The deployer pays the deposit for the compiled code and whatever `init` stored.
2. **Growth.** If a call makes `state_bytes` larger, the caller pays `required − current deposit` (if positive).
3. **Shrinking.** If a call makes `state_bytes` smaller, the caller receives a proportional refund: `deposit × (old_bytes − new_bytes) ÷ old_bytes`.
4. **Destroy.** `destroy(to)` pays the contract's balance **and its whole deposit** to `to`.

Every deploy and invoke transaction carries a `max_deposit`: the most the sender accepts to lock. If the call would need more, it fails with `storage deposit of N motes exceeds max_deposit M (contract state: B bytes)`. The wallet sets `max_deposit` to 1 TCN by default; change it with `--max-deposit <TCN>`.

> **NOTE**

> The refund goes to **whoever frees the storage**, not necessarily to whoever paid for it. In practice this is fair: the user who withdraws their balance from a token or a savings contract is usually the one who created that entry. It also rewards users for cleaning up.

## 5.3 The deposit in action

The time-locked savings contract (Section 8.6) creates three storage entries when someone deposits and deletes all of them on withdrawal. Here is the deposit moving on a regtest network (0.0001 TCN per kB; transcripts trimmed to the relevant lines). Right after deployment, the contract holds only its 854 bytes of compiled code:

```
$ thecoin-wallet contract program tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Contract:  Savings at tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Creator:   tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq (block 77, tx
699c87fe6490e0fb622ae1e6cb7cbebd0b6245ae3174b65728313b2c36e31c2b)
Balance:   0 TCN
Storage:   854 bytes in 0 entries · deposit 0.0001 TCN
```

After a deposit of 5 TCN, 3 entries (102 bytes) were added. The contract is still under 1 kB, so the required deposit did not change:

```
$ thecoin-wallet contract program tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Contract:  Savings at tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Creator:   tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq (block 77, tx
699c87fe6490e0fb622ae1e6cb7cbebd0b6245ae3174b65728313b2c36e31c2b)
Balance:   5 TCN
Storage:   956 bytes in 3 entries · deposit 0.0001 TCN
```

After `withdraw()` the entries are gone, and the saver received 0.0001 × 102 ÷ 956 ≈ 0.00001066 TCN of deposit back:

```
$ thecoin-wallet contract program tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Contract:  Savings at tcr1qfqaq5tuuyyspm59kghzqgk28lzp76zle3lzj2
Creator:   tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq (block 77, tx
699c87fe6490e0fb622ae1e6cb7cbebd0b6245ae3174b65728313b2c36e31c2b)
Balance:   0 TCN
Storage:   854 bytes in 0 entries · deposit 0.00008934 TCN
```

## 5.4 Destroying a contract

`destroy(to)` is the only way to recover the full deposit. It requires the contract's storage to be empty apart from scalar state variables, which are cleared automatically. The escrow recipe (Section 8.4) uses it: once the escrow is settled, the buyer calls `close()` and receives the deposit back.

```
action shutdown():
    require caller == owner, "only the owner"
    while len(members) > 0:
        let m: address = members.pop()
        shares.remove(m)
    destroy(owner)
```

A contract that keeps unbounded maps can usually never be destroyed — and that is fine. Only design for `destroy` when the contract has a natural end.

## 5.5 Designing for small state

> **FUEL TIP**
> Storage is also the most expensive thing in fuel: every write costs 400 fuel plus 4 per byte, every read 250. Small state is cheap state.

- Prefer **constants** for values that never change: they cost neither storage nor reads.

- Store **amounts and flags**, not history. Events are the right place for history: they are kept in receipts and cost no storage.
- **Delete what you no longer need** (`remove`, set to default, `pop`). The caller gets part of the deposit back, and the next caller pays less fuel.
- Use fixed-size keys (`address`, `int`, 32-byte hashes) instead of long texts.
- Keep lists bounded. A list that only grows makes the contract bigger forever and any loop over it slower forever.

CHAPTER 6

# Fuel and fees

Every node runs every contract call, so every call must pay for the work it causes. The Coin measures that work in **fuel**, and the transaction fee is computed from the transaction size and the fuel it reserves.

## 6.1 The fuel schedule

The fuel schedule is part of the consensus rules. These are the prices in language version 1:

Table 5: Fuel schedule, language version 1.

| Operation | Fuel |
|---|---|
| Each statement executed | 2 |
| Each expression evaluated | 1 |
| Reading a constant or local value, per 32 bytes of its size | 1 |
| Binary operator, per 32 bytes of both operands | 1 |
| Each loop iteration (`range` and local lists) | 1 |
| Appending to a local list | 1 + 1 per 32 bytes |
| Calling a function (entry point or helper) | 20 |
| Storage read (state variable, map item, list item or length) | 250 |
| Storage write or delete | 400 + 4 per byte of key and value |
| `sha256`, `blake3` | 60 + 20 per 64 bytes of input |
| `verify_ed25519` | 3 500 + 1 per 64 bytes of message |
| `ring_verify` | 5 000 + 10 000 per ring member |
| `send` | 300 |
| `emit` | 100 + 1 per byte of the field values |
| `destroy` | 1 000 |
| Deploy: compiling the source code | 5 per byte of source |
| Invoke: loading the contract | 100 + 1 per 100 bytes of compiled code |

> **FUEL TIP**
> **Why these prices?** The schedule was calibrated with benchmarks (`crates/tccl/tests/fuel_bench.rs` and `crates/node/tests/fuel_storage_bench.rs`) so that every operation costs roughly 20 nanoseconds of CPU per unit of fuel on a 2-vCPU server. A block completely filled with 50 000 000 fuel executes in about 1.2 seconds in the worst case, even on the modest machines that validate The Coin. That is why storage reads and cryptography are priced much higher than arithmetic.

## 6.2 Where the fuel goes

The counter's `increment(5)` used 1 674 fuel in the simulator. Following the schedule statement by statement:

| Part of the call | Fuel |
|---|---|
| Calling `increment` | 20 |
| Two `require` statements (statement + comparison) | 12 |
| `count += amount`: read `count` (250), write 20 bytes (480), expressions | 736 |
| `last_caller = caller`: write 24 bytes (496), expressions | 499 |

| Part of the call | Fuel |
|---|---|
| `emit Increased(…)`: read `count` again (250), event of 52 bytes (152), expressions | 407 |
| **Total in the simulator** | **1 674** |
| Loading the 323-byte program on the network | 103 |
| **Total on the network** (matches the wallet's measurement in Section 7.4) | **1 777** |

Storage accounts for 1 476 of the 1 674 units. That is typical: **reads and writes dominate**, while arithmetic and control flow are almost free.

> **FUEL TIP**
> `emit Increased(caller, amount, count)` reads `count` from storage a second time. In a function that uses a state variable several times, copy it once into a local variable (`let total: int = count + amount`), write it once and use the local afterwards. Each avoided read saves 250 fuel.

Deploying costs 5 fuel per byte of source code plus whatever `init` does. The 2 984-byte escrow recipe uses 14 920 fuel to compile and 2 950 to run `init`: 17 870 in total, exactly what the wallet measured when deploying it (Section 7.7). Comments count as source code.

## 6.3 The fee formula

The minimum fee of a transaction is

**fee** = ( `base_fee` + ⌈ size × `fee_per_kb` ÷ 1 000 ⌉ + ⌈ `max_fuel` × `fee_per_kfuel` ÷ 1 000 ⌉ ) × congestion

where **size** is the serialized transaction size in bytes (it includes the source code for a deploy and the arguments for an invoke) and **max_fuel** is the fuel limit the transaction **reserves**. All three prices are governance parameters:

| Parameter | Mainnet & testnet | Regtest | Meaning |
|---|---|---|---|
| `base_fee` | 1 000 motes | 100 motes | per transaction |
| `fee_per_kb` | 10 000 motes | 1 000 motes | per 1 000 bytes of transaction |
| `fee_per_kfuel` | 1 000 motes | 100 motes | per 1 000 units of reserved fuel |

> **FUEL TIP**
> **You pay for the fuel you reserve, not for the fuel you use.** An unused part of `max_fuel` is not refunded, and a large `max_fuel` also lowers your transaction's priority in the mempool, which orders transactions by fee per weight unit (`bytes + max_fuel / 100`). Let the wallet measure the fuel for you.

### 6.3.1 A worked example

The `increment(5)` transaction of Section 7.4 was 205 bytes with `max_fuel` 7 310. On regtest:

- 100 + ⌈205 × 1 000 ÷ 1 000⌉ + ⌈7 310 × 100 ÷ 1 000⌉ = 100 + 205 + 731 = **1 036 motes** minimum (the node's simulation endpoint reports `"required_fee": 1036` in Section 7.8);
- the wallet's default **normal** priority pays 1.25 × the minimum: ⌈1 036 × 1.25⌉ = **1 295 motes**, the fee shown in the transcript.

The same transaction on mainnet costs 1 000 + 2 050 + 7 310 = 10 360 motes minimum, or 12 950 motes (0.0001295 TCN) at normal priority.

## 6.4 Congestion and priority

The **congestion multiplier** starts at 1× and follows demand: when blocks are more than half full it rises by up to 12.5% per block, and when they are emptier it falls back, never below 1× (and never above 1 000×). The part of the fee caused by congestion is **burned**, so miners gain nothing by stuffing blocks. Anything paid above the minimum is a priority tip for the miner.

The wallet offers four priorities with `--priority`:

| Priority | Fee paid |
|---|---|
| `low` | the minimum at the current congestion |
| `normal` | minimum × 1.25 (default) — still valid if congestion rises for a block |
| `high` | enough to be ahead of the waiting transactions, at least 2× the minimum |
| `urgent` | at least 2× `high`: next block with very high probability |

`thecoin-wallet fees` shows the current congestion, the fee parameters and the fee of a typical transfer at each priority.

## 6.5 How the wallet measures fuel

When you run `contract deploy` or `contract invoke` without `--max-fuel`, the wallet:

1. builds the transaction with a provisional limit of 2 000 000 fuel and asks the node to **simulate** it on the current state (`POST /api/v1/tx/simulate`);
2. stops with `contract call would fail: <error> (nothing was sent)` if the simulation fails — you pay nothing;
3. otherwise sets `max_fuel` to **fuel used × 1.3 + 5 000** (at most 10 000 000), shows the measured fuel, the return value and the events of the simulation, and signs the final transaction.

The 30% margin plus 5 000 absorbs small differences between the simulation and the moment the transaction is included in a block — for example another user's call that makes a list longer. If your action's cost depends heavily on state that may change, or needs more than 2 000 000 fuel to simulate, pass `--max-fuel` yourself.

## 6.6 When a call fails

The node simulates every contract transaction before accepting it into its mempool and **rejects calls that would fail**, so a failing call normally never reaches a block. A call can still fail after it was accepted, when the state changes before it is mined: another transaction took the last item, the deadline passed, the reserved fuel is no longer enough.

In that case the transaction is still included and:

- **the fee is paid** — the network did the work;
- **every effect of the call is reverted**: storage writes, events, `send`s and the TCN sent with the call (the `value` goes back to the sender);
- no storage deposit is charged;
- the receipt has `"success": false`, the error message and the fuel used (all of `max_fuel` if the call ran out of fuel).

## 6.7 Limits

Table 6: Network limits relevant to contracts. Language limits are listed in Appendix A.6.

| Limit | Value |
|---|---|
| Fuel per transaction (`max_fuel`) | 1 to 10 000 000 |
| Fuel per block (mainnet default, governance) | 50 000 000 |
| Fuel for a view through the node API | 2 000 000 |
| Fuel per call in the `tccl` simulator | 5 000 000 |
| Deploy transaction size | 64 000 bytes |
| Other transaction size | 16 384 bytes |
| Source code | 48 000 bytes |
| Compiled program | 262 144 bytes |
| Arguments per call | 32 |
| Function name in an invoke | 1 to 64 bytes |
| Events per call | 64 |

CHAPTER 7

# Deploying to the network

Your contract compiles and its tests pass. Time to put it on a real network.

## 7.1 Before you start

You need:

- **A node API.** The wallet talks to `http://127.0.0.1:7334` by default (mainnet; testnet uses port 17334). Use `--node <url>` or the `THECOIN_NODE` environment variable to point it at another node.
- **A wallet with some TCN.** Create one with `thecoin-wallet create` (or use the one the installer created) and show its address with `thecoin-wallet address`.
- **A network choice.** The commands in this chapter are written for **mainnet**. While you are learning, add `--network testnet` to every `thecoin-wallet` command (or set `THECOIN_NETWORK=testnet`) and use a testnet node: testnet coins have no value, so mistakes cost nothing.

> **NOTE**
> The transcripts in this chapter were recorded on a private **regtest** network with these environment variables set, so the commands look exactly like mainnet commands: `THECOIN_NETWORK=regtest`, `THECOIN_NODE=http://127.0.0.1:47334`, `THECOIN_WALLET=alice.json` and `THECOIN_WALLET_PASSWORD` (so the wallet does not ask for the password). The `-y` flag skips the "Broadcast this transaction? [y/N]" confirmation; leave it out when you work by hand. Regtest addresses start with `tcr1` and its fees are ten times lower than mainnet's (chapter 6), but fuel is identical.

Global wallet options that are useful with contracts:

| Option | Effect |
|---|---|
| `--network mainnet\|testnet\|regtest` | Network (and address prefix) to use. |
| `--node <url>` | Node API to talk to. |
| `-w, --wallet <file>` | Wallet file (default `~/.thecoin/wallet-<network>.json`). |
| `--from <n>` | Which address of the wallet signs (default 0). |
| `--priority low\|normal\|high\|urgent` | Fee priority (chapter 6). |
| `-y, --yes` | Do not ask for confirmation. |
| `--dry-run` | Build and print the signed transaction without broadcasting it. |

## 7.2 Deploy

```
$ thecoin-wallet -y contract deploy counter.tccl
Fuel:     2420 measured, limit 8146
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   deploy contract Counter (484 bytes of TCCL)
Fee:      0.00001948 TCN (643 bytes, priority Normal)
Broadcast OK. txid: 28e99a814f77e26f776274cef2e2e80db1703f15b360690b26d3aef0a03fdd60
Contract address: tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
```

The wallet compiled the file locally first (with the network's address prefix), simulated the deployment (2 420 fuel = 484 bytes × 5 to compile; there is no `init`), set the fuel limit, and broadcast the transaction. The **contract address** is known before the transaction is mined: it is derived from the deployer's address and the transaction nonce, so the same wallet never gets the same contract address twice.

The full syntax is:

```
thecoin-wallet contract deploy <file.tccl> [init args...] [--value <TCN>] [--max-fuel <n>] [--max-deposit <TCN>]
```

Arguments are given in the order of the `init` parameters and parsed with their declared types:

| Type | How to write the argument |
|---|---|
| `int` | `42`, `-5`, `1_000`, or an amount with a unit: `2.5tcn` = 250 000 000 |
| `bool` | `true` or `false` |
| `text` | anything; surrounding double quotes are removed: `"hello world"` |
| `bytes` | hex with `0x`: `0xdeadbeef` |
| `address` | `tc1…` (mainnet), `tct1…` (testnet), `tcr1…` (regtest) |
| `list[T]` | `[a, b, c]` — quote the whole list in your shell: `"[1, 2, 3]"` |

Note that the wallet's `--value` is in **TCN** (`--value 25` sends 25 TCN), while inside arguments of type `int` a plain number is in motes unless you add `tcn`.

## 7.3 Inspect the contract

```
$ thecoin-wallet contract program tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
Contract:  Counter at tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
Creator:   tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq (block 64, tx
28e99a814f77e26f776274cef2e2e80db1703f15b360690b26d3aef0a03fdd60)
Balance:   0 TCN
Storage:   323 bytes in 0 entries · deposit 0.0001 TCN
  action increment(amount: int)
  view get() -> int
  view last() -> address
```

## 7.4 Invoke an action

```
$ thecoin-wallet -y contract invoke tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz increment 5
Fuel:     1777 measured, limit 7310
Preview:  Increased(by: tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq, amount: 5, total: 5) (simulated on
the current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   increment(5) on tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
Fee:      0.00001295 TCN (205 bytes, priority Normal)
Broadcast OK. txid: 04a24a2dd4268cc4fa0ff1d3c54aba9f953f03abc8aad56a9e451a84da4c7e5b
```

The syntax is `contract invoke <address> <function> [args...] [--value <TCN>] [--max-fuel <n>] [--max-deposit <TCN>]`. The wallet reads the function's parameter types from the node, so arguments are parsed exactly like for `deploy`. It refuses `--value` for a function that is not payable.

A call that would fail is stopped before anything is sent:

```
$ thecoin-wallet -y contract invoke tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz increment 500
error: contract call would fail: requirement failed: at most 100 per call (nothing was sent)
```

## 7.5 Query a view

Views are free and need no wallet password:

```
$ thecoin-wallet contract view tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz get
5
$ thecoin-wallet contract view tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz last
tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
```

## 7.6 Read the receipt

`thecoin-wallet tx <txid>` shows a transaction with its execution receipt (some fields omitted and short arrays joined on one line):

```
$ thecoin-wallet tx 04a24a2dd4268cc4fa0ff1d3c54aba9f953f03abc8aad56a9e451a84da4c7e5b
{
  "txid": "04a24a2dd4268cc4fa0ff1d3c54aba9f953f03abc8aad56a9e451a84da4c7e5b",
  "sender": "tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq",
  "fee": 1295,
  "size": 205,
  "action": {
    "type": "invoke",
    "contract": "tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz",
    "function": "increment",
    "args": [ "5" ],
    "value": 0,
    "max_fuel": 7310,
    "max_deposit": 100000000
  },
  "block_height": 66,
  "confirmations": 4,
  "success": true,
  "error": null,
  "fuel_used": 1777,
  "burned": 0,
  "logs": [
    {
      "contract": "tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz",
      "event": "Increased",
      "fields": [
        [ "by", "tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq" ],
        [ "amount", "5" ],
        [ "total", "5" ]
      ]
    }
  ],
  "program": null,
  "return_value": null
}
```

| Field | Meaning |
|---|---|
| `success`, `error` | Whether the contract code succeeded, and the error message if not. |
| `fuel_used` | Fuel consumed, including loading (or compiling) the contract. |
| `logs` | Events emitted, with fields rendered as text (integers as decimal strings). |
| `return_value` | The value returned by the action, if any. |
| `program` | For a deploy: the address of the new contract. |
| `burned` | Part of the fee burned by congestion. |

## 7.7 Deploying with arguments and TCN

The escrow recipe (Section 8.4) takes the seller, the arbiter and a duration as `init` arguments, and the payment itself as `--value`:

```
$ thecoin-wallet -y contract deploy escrow.tccl tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w
tcr1evz9j6snsj25wfan6uym3r4la3zjl5qzp896zj 1440 --value 25
Fuel:     17870 measured, limit 28231
Preview:  Funded(buyer: tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq, seller:
tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w, amount: 2500000000, deadline: 1513) (simulated on the
current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   deploy contract Escrow (2984 bytes of TCCL)
Fee:      0.00007658 TCN (3202 bytes, priority Normal)
Broadcast OK. txid: 3fc7b79a5b5da5edf6b644bab1a069a2492a357ef8b38cd700ccda198349a2da
Contract address: tcr1kgmvym7tmtqxnj8knjqux8el8nd74lytkqyrk6
$ thecoin-wallet contract view tcr1kgmvym7tmtqxnj8knjqux8el8nd74lytkqyrk6 status
"open"
$ thecoin-wallet contract view tcr1kgmvym7tmtqxnj8knjqux8el8nd74lytkqyrk6 locked
2500000000
```

Forgetting the payment is caught by the contract's own `require`, during the wallet's simulation:

```
$ thecoin-wallet -y contract deploy escrow.tccl tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w
tcr1evz9j6snsj25wfan6uym3r4la3zjl5qzp896zj 1440
error: contract call would fail: requirement failed: attach the payment with --value (nothing was
sent)
```

## 7.8 The HTTP API

Everything the wallet does goes through the node's REST API, which you can use from any language. Integers in JSON results are strings, to keep their full 128-bit precision.

| Endpoint | Purpose |
|---|---|
| `GET /api/v1/program/{address}` | Contract metadata: name, creator, deploy transaction, source hash, balance, storage, deposit and interface. |
| `POST /api/v1/program/{address}/view` | Call a view. Body: `{"function": "get", "args": []}` with arguments as strings. |
| `POST /api/v1/tx/simulate` | Simulate a signed transaction (hex) on the current state without broadcasting it. |
| `POST /api/v1/tx` | Broadcast a signed transaction. |
| `GET /api/v1/tx/{txid}` | Transaction with receipt. |

The contract metadata below is shown with one field and one function per line; the JSON itself is a single line:

```
$ curl -s http://127.0.0.1:47334/api/v1/program/tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
{
  "address": "tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz",
  "name": "Counter",
  "creator": "tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq",
  "created_height": 64,
  "deploy_txid": "28e99a814f77e26f776274cef2e2e80db1703f15b360690b26d3aef0a03fdd60",
  "source_hash": "678565762aeb45005ff68065af3d03ffbcf6c99c9effafef5f374672a4227036",
  "balance": 0,
  "state_bytes": 367,
  "storage_items": 2,
  "deposit": 10000,
  "functions": [
    {"name": "increment", "kind": "action", "payable": false, "params": [["amount", "int"]],
"returns": "nothing"},
    {"name": "get", "kind": "view", "payable": false, "params": [], "returns": "int"},
    {"name": "last", "kind": "view", "payable": false, "params": [], "returns": "address"}
  ]
}
$ curl -s -X POST -H 'content-type: application/json' -d '{"function":"get","args":[]}'
http://127.0.0.1:47334/api/v1/program/tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz/view
{"result":"5","error":null,"fuel_used":273}
```

To preview an action without sending it, sign it with `--dry-run` and post the hex to the simulation endpoint (the long hex string is shortened here):

```
$ thecoin-wallet --dry-run contract invoke tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz increment 3
Fuel:     1777 measured, limit 7310
Preview:  Increased(by: tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq, amount: 3, total: 8) (simulated on
the current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   increment(3) on tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz
Fee:      0.00001295 TCN (205 bytes, priority Normal)
Signed transaction (not broadcast):
0103004354000200000000000000f05…
txid: 9529fc95bd70d68deafc298b59b187db5ec7f1161add63d9de6b9c1b6ca5cd49
$ curl -s -X POST -H 'content-type: application/json' -d '{"tx":"0103004354000200000000000000f05…"}'
http://127.0.0.1:47334/api/v1/tx/simulate
{
  "valid": true,
  "invalid_reason": null,
  "success": true,
  "error": null,
  "fuel_used": 1777,
  "required_fee": 1036,
  "logs": [
    {
      "contract": "tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz",
      "event": "Increased",
      "fields": [
        [ "by", "tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq" ],
        [ "amount", "3" ],
        [ "total", "8" ]
      ]
    }
  ],
  "return_value": null,
  "program": null
}
```

## 7.9 Verifying a contract's source

The deploy transaction contains the complete source code. `thecoin-wallet tx <deploy txid>` shows it in `action.source` (shortened below), together with `action.source_hash`; the same hash is reported by `/api/v1/program/{address}`. Anyone can therefore read the exact code of a contract, compile it with `tccl check` and compare the interface before trusting it with money.

```
$ thecoin-wallet tx 28e99a814f77e26f776274cef2e2e80db1703f15b360690b26d3aef0a03fdd60
{
  "txid": "28e99a814f77e26f776274cef2e2e80db1703f15b360690b26d3aef0a03fdd60",
  "sender": "tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq",
  "nonce": 0,
  "fee": 1948,
  "size": 643,
  "action": {
    "type": "deploy",
    "source": "# The smallest useful contract: a counter anyone can increase.\ncontract
Counter\n\nstate count: int\nstate last_c…",
    "source_hash": "678565762aeb45005ff68065af3d03ffbcf6c99c9effafef5f374672a4227036",
    "init_args": [],
    "value": 0,
    "max_fuel": 8146,
    "max_deposit": 100000000
  },
  "block_height": 64,
  "confirmations": 7,
  "success": true,
  "fuel_used": 2420,
  "program": "tcr17kp5hrqhpdkwmukkrsfp6xft6y9l8j02rhxmvz"
}
```

## 7.10 There are no upgrades

A deployed contract's code never changes. This is a feature — users can trust that the rules will not change under them — but it means you must plan for mistakes:

- test thoroughly on the simulator and on testnet before mainnet;
- keep the amount at risk small at first, with limits enforced by the contract;
- if a new version is needed, deploy a new contract and let users move to it (for example with a `withdraw` action on the old one that always keeps working);
- if the contract has a natural end, give it a `destroy` path to recover the storage deposit.

CHAPTER 8

# Recipes

Each recipe in this chapter is a complete contract you can deploy as it is, followed by the decisions behind it and a session in the simulator. They are ordered from simple to advanced. All of them live in `crates/tccl/examples/`, and `crates/tccl/tests/examples.rs` tests their normal use and their failure paths.

Read the code first, then the explanation. The recipes are also meant to be **modified**: every one ends with ideas for variations.

## 8.1 Tip jar

| | |
|---|---|
| **File** | `crates/tccl/examples/tip_jar.tccl` |
| **Teaches** | init · payable · owner permissions · balance · events |
| **Entry points** | `tip(message)` payable · `withdraw(amount)` · `stats()` |

Anyone can send TCN with a short public message; only the owner can take the money out.

```
# A tip jar: anyone can send TCN with a message, only the owner withdraws.
contract TipJar

state owner: address
state total_received: int
state tips: int

event Tip(from: address, amount: int, message: text)
event Withdrawn(to: address, amount: int)

init():
    owner = caller

action tip(message: text) payable:
    require value >= TCN / 100, "minimum tip is 0.01 TCN"
    require len(message) <= 140, "message too long"
    total_received += value
    tips += 1
    emit Tip(caller, value, message)

action withdraw(amount: int):
    require caller == owner, "only the owner can withdraw"
    require amount > 0 and amount <= balance, "invalid amount"
    send(owner, amount)
    emit Withdrawn(owner, amount)

view stats() -> list[int]:
    return [total_received, tips, balance]
```

**How it works.** `init` runs once, at deployment, so `owner = caller` stores the deployer's address. `tip` is `payable`: the TCN attached to the transaction is already in the contract's `balance` when the first line runs, and `value` tells how much it was. The minimum is written as `TCN / 100` rather than a magic number. `withdraw` checks the caller **before** doing anything, and bounds `amount` by `balance` so the error message is clear (without that line `send` would fail with `insufficient contract balance`). `stats` returns three numbers at once as a `list[int]`.

```
$ tccl run tip_jar.tccl --from owner deploy
deployed TipJar at c871ebffa3b42e0f5d66c9edb86cd18e25256b87
ok · fuel used: 4449 · height: 2 · owner balance: 100000000000000 motes
```

```
$ tccl run tip_jar.tccl --from fan --value 3tcn call tip "great work"
event Tip(from: tcr1c9tjxeq7dvrplrhwfm8gthth80wqa2mrjev34w, amount: 300000000, message: "great work")
ok · fuel used: 1659 · height: 3 · fan balance: 99999700000000 motes
$ tccl run tip_jar.tccl --from fan --value 0.001tcn call tip "tiny"
FAILED: requirement failed: minimum tip is 0.01 TCN · fuel used: 30 (all changes reverted)
$ tccl run tip_jar.tccl --from fan call withdraw 1tcn
FAILED: requirement failed: only the owner can withdraw · fuel used: 277 (all changes reverted)
$ tccl run tip_jar.tccl --from owner call withdraw 2tcn
event Withdrawn(to: tcr1pzr4w70h457vcer4xm4q7s85r97f7upm8rr97t, amount: 200000000)
ok · fuel used: 1231 · height: 4 · owner balance: 100000200000000 motes
$ tccl run tip_jar.tccl view stats
result: [300000000, 1, 100000000]
ok · fuel used: 526 · height: 4 · alice balance: 100000000000000 motes
```

> **SECURITY TIP**
> `withdraw` pays `owner`, not `caller`. Even though they are equal after the `require`, paying the stored address makes the intent obvious and survives future edits that relax the check.

**Variations.** Add `set_owner(new_owner: address)` so the owner can hand the jar over. Keep a `map[address, int]` of the total tipped by each fan and a `top_fan` view.

## 8.2 Token with allowances

| | |
|---|---|
| **File** | `crates/tccl/examples/token.tccl` |
| **Teaches** | maps · helper functions · composite keys · supply cap · allowances |
| **Entry points** | `mint` · `transfer` · `approve` · `transfer_from` · `balance_of` · `allowance` |

A fungible token: an owner mints up to a fixed maximum supply, holders transfer, and holders can **approve** another address (a shop, an exchange contract operator) to spend part of their balance.

```
# A simple fungible token with a fixed maximum supply.
contract SimpleToken

const MAX_SUPPLY: int = 21_000_000
state name: text = "Cloud Token"
state owner: address
state supply: int
state balances: map[address, int]
state allowances: map[bytes, int]

event Transfer(from: address, to: address, amount: int)
event Approval(holder: address, spender: address, amount: int)

init():
    owner = caller

action mint(to: address, amount: int):
    require caller == owner, "only the owner can mint"
    require amount > 0, "amount must be positive"
    require supply + amount <= MAX_SUPPLY, "max supply reached"
    supply += amount
    balances[to] += amount
    emit Transfer(zero_address(), to, amount)

action transfer(to: address, amount: int):
    move(caller, to, amount)

action approve(spender: address, amount: int):
    require amount >= 0, "amount cannot be negative"
    allowances[pair(caller, spender)] = amount
    emit Approval(caller, spender, amount)

action transfer_from(holder: address, to: address, amount: int):
```

```
    let key: bytes = pair(holder, caller)
    require allowances[key] >= amount, "allowance too low"
    allowances[key] -= amount
    move(holder, to, amount)

view balance_of(who: address) -> int:
    return balances[who]

view allowance(holder: address, spender: address) -> int:
    return allowances[pair(holder, spender)]

fn move(from: address, to: address, amount: int):
    require amount > 0, "amount must be positive"
    require balances[from] >= amount, "insufficient token balance"
    balances[from] -= amount
    balances[to] += amount
    emit Transfer(from, to, amount)

fn pair(a: address, b: address) -> bytes:
    return to_bytes(a) + to_bytes(b)
```

**How it works.** Balances are a `map[address, int]`: addresses that never received tokens read as 0. All movements go through one helper, `move`, so the balance checks exist in exactly one place. Allowances need a key made of **two** addresses; `pair` concatenates their 20 bytes each, which can never be ambiguous (Section 4.15). `transfer_from` reduces the allowance **before** moving tokens, and if `move` fails the whole call — including the allowance change — is reverted. Token amounts here are whole units; nothing forces them to be motes, because this token is not TCN.

```
$ tccl run token.tccl --from issuer deploy
deployed SimpleToken at 639990f8951b80368b4433700306a34e9af452fe
ok · fuel used: 9500 · height: 2 · issuer balance: 100000000000000 motes
$ tccl run token.tccl --from issuer call mint @alice 1000
event Transfer(from: tcr1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqwe28w, to:
tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, amount: 1000)
ok · fuel used: 2260 · height: 3 · issuer balance: 100000000000000 motes
$ tccl run token.tccl --from alice call transfer @bob 300
event Transfer(from: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, to:
tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 300)
ok · fuel used: 2114 · height: 4 · alice balance: 100000000000000 motes
$ tccl run token.tccl --from alice call approve @shop 100
event Approval(holder: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, spender:
tcr1j6wkxaxx7zr5mk628agz6hx3xnmf5yxqrfuxwc, amount: 100)
ok · fuel used: 881 · height: 5 · alice balance: 100000000000000 motes
$ tccl run token.tccl --from shop call transfer_from @alice @carol 60
event Transfer(from: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, to:
tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e, amount: 60)
ok · fuel used: 3325 · height: 6 · shop balance: 100000000000000 motes
$ tccl run token.tccl --from shop call transfer_from @alice @carol 41
FAILED: requirement failed: allowance too low · fuel used: 312 (all changes reverted)
$ tccl run token.tccl view balance_of @alice
result: 640
ok · fuel used: 274 · height: 6 · alice balance: 100000000000000 motes
$ tccl run token.tccl view allowance @alice @shop
result: 40
ok · fuel used: 304 · height: 6 · alice balance: 100000000000000 motes
```

**SECURITY TIP**

`approve` **overwrites** the allowance. If Alice lowers an allowance from 100 to 50 while the spender watches the mempool, the spender can use the 100 first and then the new 50. Users should set an allowance to 0 and wait for confirmation before setting a new non-zero value, or you can add `increase_allowance`/`decrease_allowance` actions.

**FUEL TIP**

A transfer that empties a sender's balance **deletes** that map entry, which makes the storage smaller and refunds part of the deposit to the sender (chapter 5).

**Variations.** Add `burn(amount)`. Add a `decimals` constant for wallets. Replace the owner with a fixed minting schedule based on `height`.

## 8.3 Crowdfunding with a deadline

| | |
|---|---|
| **File** | `crates/tccl/examples/crowdfund.tccl` |
| **Teaches** | init arguments · block height as time · refunds · remove · if/elif/else views |
| **Entry points** | `init(goal_tcn, duration_blocks)` · `pledge()` payable · `collect()` · `refund()` · `status()` |

A campaign collects pledges until a deadline. If the goal is reached the creator collects everything; if not, every backer can take their pledge back.

```
# Crowdfunding with a goal and a deadline (in blocks).
# If the goal is reached the creator collects; otherwise backers get refunds.
contract Crowdfund

state creator: address
state goal: int
state deadline: int
state raised: int
state collected: bool
state pledges: map[address, int]

event Pledged(backer: address, amount: int)
event Refunded(backer: address, amount: int)

init(goal_tcn: int, duration_blocks: int):
    require goal_tcn > 0, "goal must be positive"
    require duration_blocks >= 10, "campaign too short"
    creator = caller
    goal = goal_tcn * TCN
    deadline = height + duration_blocks

action pledge() payable:
    require height <= deadline, "campaign ended"
    require value > 0, "send some TCN"
    pledges[caller] += value
    raised += value
    emit Pledged(caller, value)

action collect():
    require caller == creator, "only the creator"
    require height > deadline, "campaign still running"
    require raised >= goal, "goal not reached"
    require not collected, "already collected"
    collected = true
    send(creator, raised)

action refund():
    require height > deadline, "campaign still running"
    require raised < goal, "goal was reached, no refunds"
    let amount: int = pledges[caller]
    require amount > 0, "nothing to refund"
    pledges.remove(caller)
    send(caller, amount)
    emit Refunded(caller, amount)

view status() -> text:
    if height <= deadline:
        return "running"
    elif raised >= goal:
        return "successful"
    else:
        return "failed"
```

**How it works.** `init` receives the goal in whole TCN and the duration in blocks, validates both and converts the goal to motes once. Time is the block `height`: pledges are accepted while `height <= deadline`, collecting and refunds only after. `collected` prevents the creator from collecting twice. In `refund` the amount is read into a local, the entry is removed **before** the `send` (so a second refund finds nothing), and the event is emitted last.

The simulator's `--height` option moves time forward:

```
$ tccl run crowdfund.tccl --from maker deploy 5 10
deployed Crowdfund at 5587c5870fadde50689b36eba05a3a24ec58a5d2
ok · fuel used: 9183 · height: 2 · maker balance: 100000000000000 motes
$ tccl run crowdfund.tccl --from bob --value 3tcn call pledge
event Pledged(backer: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 300000000)
ok · fuel used: 1980 · height: 3 · bob balance: 99999700000000 motes
$ tccl run crowdfund.tccl --from carol --value 2tcn call pledge
event Pledged(backer: tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e, amount: 200000000)
ok · fuel used: 1980 · height: 4 · carol balance: 99999800000000 motes
$ tccl run crowdfund.tccl --from maker call collect
FAILED: requirement failed: campaign still running · fuel used: 533 (all changes reverted)
$ tccl run crowdfund.tccl --height 20 view status
result: "successful"
ok · fuel used: 785 · height: 20 · alice balance: 100000000000000 motes
$ tccl run crowdfund.tccl --from maker call collect
ok · fuel used: 2519 · height: 21 · maker balance: 100000500000000 motes
$ tccl run crowdfund.tccl --from bob call refund
FAILED: requirement failed: goal was reached, no refunds · fuel used: 783 (all changes reverted)
```

> **SECURITY TIP**
> The creator's `collect` sends `raised`, not `balance`. Anyone can send TCN to a contract address with a normal transfer, which increases `balance` without any code running; accounting based on your own counters cannot be disturbed that way.

**Variations.** Let backers cancel a pledge while the campaign runs. Pay the creator in milestones. Add a `stretch_goal`.

## 8.4 Escrow with an arbiter

| | |
|---|---|
| **File** | `crates/tccl/examples/escrow.tccl` |
| **Teaches** | payable init · roles · state machine · shared settlement helper · destroy |
| **Entry points** | `init(seller, arbiter, duration)` payable · `release` · `cancel` · `dispute` · `resolve(pay_seller)` · `reclaim` · `close` · `status` · `locked` |

A buyer and a seller who do not trust each other agree on a third party. The buyer locks the payment in the contract when deploying it.

```
# Escrow with an arbiter.
#
# The buyer deploys the contract with the payment attached. The buyer releases
# the money when the goods arrive; if buyer and seller disagree, either can
# open a dispute and the arbiter decides. If nobody acts before the deadline,
# the buyer can take the money back.
contract Escrow

const MIN_DURATION: int = 60          # about 1 hour (1 block = 1 minute)
const MAX_DURATION: int = 525_600     # about 1 year

state buyer: address
state seller: address
state arbiter: address
state amount: int
state deadline: int
state disputed: bool
state settled: bool

event Funded(buyer: address, seller: address, amount: int, deadline: int)
event Disputed(by: address)
```

```
event Settled(to: address, amount: int)

init(seller_address: address, arbiter_address: address, duration_blocks: int) payable:
    require value > 0, "attach the payment with --value"
    require seller_address != caller, "buyer and seller must be different"
    require arbiter_address != caller, "the arbiter must be a third party"
    require arbiter_address != seller_address, "the arbiter must be a third party"
    require duration_blocks >= MIN_DURATION, "duration too short (min 60 blocks)"
    require duration_blocks <= MAX_DURATION, "duration too long (max 525600 blocks)"
    buyer = caller
    seller = seller_address
    arbiter = arbiter_address
    amount = value
    deadline = height + duration_blocks
    emit Funded(caller, seller_address, value, deadline)

# The buyer is happy: pay the seller.
action release():
    require caller == buyer, "only the buyer can release"
    pay(seller)

# The seller cannot deliver: give the money back.
action cancel():
    require caller == seller, "only the seller can cancel"
    pay(buyer)

action dispute():
    require caller == buyer or caller == seller, "only the buyer or the seller"
    require not settled, "already settled"
    require not disputed, "already disputed"
    disputed = true
    emit Disputed(caller)

action resolve(pay_seller: bool):
    require caller == arbiter, "only the arbiter"
    require disputed, "there is no dispute"
    if pay_seller:
        pay(seller)
    else:
        pay(buyer)

action reclaim():
    require caller == buyer, "only the buyer"
    require height > deadline, "the deadline has not passed"
    require not disputed, "a dispute is open: the arbiter decides"
    pay(buyer)

# After settlement the buyer removes the contract and recovers its storage deposit.
action close():
    require caller == buyer, "only the buyer"
    require settled, "settle the escrow first"
    destroy(buyer)

view status() -> text:
    if settled:
        return "settled"
    elif disputed:
        return "disputed"
    elif height > deadline:
        return "expired"
    return "open"

view locked() -> int:
    if settled:
        return 0
    return amount

fn pay(to: address):
```

```
    require not settled, "already settled"
    settled = true
    send(to, amount)
    emit Settled(to, amount)
```

**How it works.** The contract is a small **state machine**: `open` → (`disputed`) → `settled`, plus `expired` when the deadline passes. Each role has exactly the actions it needs:

| Role | Can |
|---|---|
| buyer | `release` (pay the seller), `dispute`, `reclaim` after the deadline if there is no dispute, `close` after settlement |
| seller | `cancel` (refund the buyer), `dispute` |
| arbiter | `resolve` a dispute in favour of either party |

Every payment goes through the single helper `pay`, whose first line makes a second settlement impossible, whoever calls it and however. `init` rejects configurations that would make the escrow pointless — a buyer who is also the seller or the arbiter — and bounds the duration.

```
$ tccl run escrow.tccl --from buyer --value 25tcn deploy @seller @arbiter 1440
deployed Escrow at 1d1b3187b4306c557732ecadbdf1052e801076da
event Funded(buyer: tcr1y72h6jxke4r0j5uajhe2vm64hyt37sexdsjs6r, seller:
tcr139nnf4gvgwq5c0qzms5vcs0lphj3jtwyjznc0e, amount: 2500000000, deadline: 1441)
ok · fuel used: 17870 · height: 2 · buyer balance: 99997500000000 motes
$ tccl run escrow.tccl --from seller call release
FAILED: requirement failed: only the buyer can release · fuel used: 277 (all changes reverted)
$ tccl run escrow.tccl --from seller call dispute
event Disputed(by: tcr139nnf4gvgwq5c0qzms5vcs0lphj3jtwyjznc0e)
ok · fuel used: 1585 · height: 3 · seller balance: 100000000000000 motes
$ tccl run escrow.tccl --from arbiter call resolve true
event Settled(to: tcr139nnf4gvgwq5c0qzms5vcs0lphj3jtwyjznc0e, amount: 2500000000)
ok · fuel used: 2427 · height: 4 · arbiter balance: 100000000000000 motes
$ tccl run escrow.tccl --from arbiter call resolve false
FAILED: requirement failed: already settled · fuel used: 1061 (all changes reverted)
$ tccl run escrow.tccl view status
result: "settled"
ok · fuel used: 276 · height: 4 · alice balance: 100000000000000 motes
$ tccl run escrow.tccl --from buyer call close
ok · fuel used: 4666 · height: 5 · buyer balance: 99997500000000 motes
$ tccl run escrow.tccl view status
error: no contract at 1d1b3187b4306c557732ecadbdf1052e801076da
```

After `close` the contract no longer exists: its storage deposit went back to the buyer, and later calls fail with `no contract at …`.

> **SECURITY TIP**
> Think about **every** path that moves money and write down who may trigger it and when. A common escrow bug is a refund path that stays open after a dispute starts; here `reclaim` explicitly requires `not disputed`.

**Variations.** Pay the arbiter a fee from the escrow. Split a resolution (e.g. 70/30) with a `seller_percent: int` argument. Allow partial releases for milestone payments.

## 8.5 Poll with registered voters

| | |
|---|---|
| **File** | `crates/tccl/examples/poll.tccl` |
| **Teaches** | list arguments · state lists · for loops · bounded loops · ties |
| **Entry points** | `init(question, choices, duration)` · `add_voters(list)` · `vote(option)` · `results` · `turnout` · `winner` · `option_name` |

A creator asks a question with 2 to 16 options and registers who may vote. Each registered address votes once; results are public while the vote runs.

```
# A poll with a fixed list of options and registered voters.
# One registered address = one vote. Results are public at any time.
contract Poll

const MAX_OPTIONS: int = 16
const MAX_VOTERS_PER_CALL: int = 50

state creator: address
state question: text
state options: list[text]
state tally: list[int]
state registered: map[address, bool]
state voted: map[address, bool]
state voters: int
state votes_cast: int
state closes_at: int

event VoterAdded(voter: address)
event Voted(voter: address, option: int)

init(poll_question: text, choices: list[text], duration_blocks: int):
    let size: int = len(poll_question)
    require size >= 1 and size <= 200, "question must have 1 to 200 bytes"
    require len(choices) >= 2 and len(choices) <= MAX_OPTIONS, "a poll needs 2 to 16 options"
    require duration_blocks >= 1, "duration must be at least 1 block"
    creator = caller
    question = poll_question
    closes_at = height + duration_blocks
    for choice in choices:
        require len(choice) >= 1 and len(choice) <= 64, "each option must have 1 to 64 bytes"
        options.push(choice)
        tally.push(0)

action add_voters(who: list[address]):
    require caller == creator, "only the creator registers voters"
    require height <= closes_at, "voting is closed"
    require len(who) <= MAX_VOTERS_PER_CALL, "at most 50 voters per call"
    for v in who:
        if not registered.has(v):
            registered[v] = true
            voters += 1
            emit VoterAdded(v)

action vote(option: int):
    require height <= closes_at, "voting is closed"
    require registered.has(caller), "you are not a registered voter"
    require not voted.has(caller), "you already voted"
    require option >= 0 and option < len(options), "unknown option"
    voted[caller] = true
    tally[option] += 1
    votes_cast += 1
    emit Voted(caller, option)

view results() -> list[int]:
    let out: list[int] = []
    for count in tally:
        out.push(count)
    return out

view option_name(index: int) -> text:
    return options[index]

view turnout() -> list[int]:
    return [votes_cast, voters]

view winner() -> text:
    require height > closes_at, "voting is still open"
    let best: int = 0
```

```
    let tie: bool = false
    for i in range(1, len(tally)):
        if tally[i] > tally[best]:
            best = i
            tie = false
        elif tally[i] == tally[best]:
            tie = true
    if tie:
        return "tie"
    return options[best]
```

**How it works.** `init` takes a `list[text]` and copies it into two state lists: the option names and a tally per option, so `tally[i]` counts votes for `options[i]`. Registration and voting use `map[address, bool]` sets (remember: only `true` can be stored). `add_voters` silently skips addresses that are already registered, so the creator can resend a list safely, and it is limited to 50 addresses per call so its fuel stays bounded. `winner` walks the tally once and detects ties.

List arguments are written in brackets. The simulator's `@name` shortcut only works for plain `address` parameters, so the voter list uses the addresses of the test accounts `alice`, `bob` and `carol` (Section 10.2 shows how to find them):

```
$ tccl run poll.tccl --from chair deploy "Lunch?" "[pizza, sushi, salad]" 100
deployed Poll at 08a57a712e06d77b85985bf0894b0cb1606dcac1
ok · fuel used: 21073 · height: 2 · chair balance: 100000000000000 motes
$ tccl run poll.tccl --from chair call add_voters "[tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa,
tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e,
tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa]"
event VoterAdded(voter: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa)
event VoterAdded(voter: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452)
event VoterAdded(voter: tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e)
ok · fuel used: 5671 · height: 3 · chair balance: 100000000000000 motes
$ tccl run poll.tccl --from bob call vote 1
event Voted(voter: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, option: 1)
ok · fuel used: 3700 · height: 4 · bob balance: 100000000000000 motes
$ tccl run poll.tccl --from bob call vote 2
FAILED: requirement failed: you already voted · fuel used: 786 (all changes reverted)
$ tccl run poll.tccl --from dave call vote 1
FAILED: requirement failed: you are not a registered voter · fuel used: 531 (all changes reverted)
$ tccl run poll.tccl --from carol call vote 1
event Voted(voter: tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e, option: 1)
ok · fuel used: 3700 · height: 5 · carol balance: 100000000000000 motes
$ tccl run poll.tccl view winner
FAILED: requirement failed: voting is still open · fuel used: 277 (all changes reverted)
$ tccl run poll.tccl --height 500 view winner
result: "sushi"
ok · fuel used: 4075 · height: 500 · alice balance: 100000000000000 motes
$ tccl run poll.tccl view results
result: [0, 2, 0]
ok · fuel used: 1041 · height: 500 · alice balance: 100000000000000 motes
```

**SECURITY TIP**
"One address, one vote" is not "one person, one vote": creating addresses is free. That is why this poll has a registration step. For token-weighted voting, read balances from your own token contract logic and snapshot them, or lock the voting tokens until the vote ends.

**FUEL TIP**
`results` and `winner` loop over at most `MAX_OPTIONS` items, and `add_voters` over at most 50. Every loop in this contract has a bound that is written in the code.

**Variations.** Let voters change their vote before the deadline (subtract from the old option). Close the poll early when `votes_cast == voters`. Require a quorum in `winner`.

## 8.6 Time-locked savings

| File | `crates/tccl/examples/savings.tccl` |
|---|---|

| | |
|---|---|
| **Teaches** | per-user maps · height locks · checks-effects-interactions · storage refunds · max |
| **Entry points** | `deposit(unlock_height)` payable · `withdraw()` · `balance_of` · `unlock_height_of` · `blocks_left` · `total` |

Each saver locks TCN until a block height they choose, up to about five years ahead. Nobody — including the saver — can withdraw before that height.

```
# Time-locked savings: lock TCN until a block height you choose.
# Nobody can withdraw early - not even the saver. Useful for self-discipline,
# long-term goals or proving that funds are committed.
contract Savings

const MAX_LOCK: int = 2_628_000        # about 5 years (1 block = 1 minute)

state balances: map[address, int]
state unlock_at: map[address, int]
state total_locked: int

event Locked(saver: address, amount: int, unlock_height: int)
event Withdrawn(saver: address, amount: int)

action deposit(unlock_height: int) payable:
    require value > 0, "attach TCN with --value"
    require unlock_height > height, "the unlock height must be in the future"
    require unlock_height <= height + MAX_LOCK, "locks are limited to about 5 years"
    require unlock_height >= unlock_at[caller], "you cannot shorten an existing lock"
    balances[caller] += value
    unlock_at[caller] = unlock_height
    total_locked += value
    emit Locked(caller, value, unlock_height)

action withdraw():
    let amount: int = balances[caller]
    require amount > 0, "you have no savings here"
    require height >= unlock_at[caller], "your savings are still locked"
    # Effects first ...
    balances.remove(caller)
    unlock_at.remove(caller)
    total_locked -= amount
    # ... interaction last.
    send(caller, amount)
    emit Withdrawn(caller, amount)

view balance_of(saver: address) -> int:
    return balances[saver]

view unlock_height_of(saver: address) -> int:
    return unlock_at[saver]

view blocks_left(saver: address) -> int:
    return max(0, unlock_at[saver] - height)

view total() -> int:
    return total_locked
```

**How it works.** One contract serves any number of savers, with two maps keyed by address. A saver can add to their savings, but only with an unlock height at least as late as the current one: otherwise a second tiny deposit could shorten the lock of the first. `withdraw` follows the **checks–effects–interactions** order: first all `require`s, then every state change, and the `send` last. Removing both map entries frees storage, so the saver also gets back part of the storage deposit (the regtest session in chapter 5 shows exactly that).

```
$ tccl run savings.tccl deploy
deployed Savings at daa436158c1dcdc0242085b6dd9451e33496cbee
ok · fuel used: 8160 · height: 2 · alice balance: 100000000000000 motes
$ tccl run savings.tccl --from bob --value 7tcn call deposit 1000
```

```
event Locked(saver: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 700000000, unlock_height:
1000)
ok · fuel used: 2581 · height: 3 · bob balance: 99999300000000 motes
$ tccl run savings.tccl --from bob --value 1tcn call deposit 900
FAILED: requirement failed: you cannot shorten an existing lock · fuel used: 300 (all changes
reverted)
$ tccl run savings.tccl --from bob call withdraw
FAILED: requirement failed: your savings are still locked · fuel used: 538 (all changes reverted)
$ tccl run savings.tccl view blocks_left @bob
result: 997
ok · fuel used: 279 · height: 3 · alice balance: 100000000000000 motes
$ tccl run savings.tccl --height 1000 --from bob call withdraw
event Withdrawn(saver: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 700000000)
ok · fuel used: 2647 · height: 1001 · bob balance: 100000000000000 motes
$ tccl run savings.tccl view total
result: 0
ok · fuel used: 273 · height: 1001 · alice balance: 100000000000000 motes
```

> **NOTE**
> `blocks_left` uses `max(0, …)` so that a lock in the past reads as 0 instead of a negative number. Because a view cannot see `caller` (it is the zero address in views), views that concern a user take the user's address as a parameter.

**Variations.** Add a beneficiary who can withdraw if the saver does not within a year after unlock. Allow early withdrawal with a penalty that is sent to a charity address constant.

## 8.7 Multi-owner treasury

| | |
|---|---|
| **File** | `crates/tccl/examples/treasury.tccl` |
| **Teaches** | M-of-N approvals · parallel maps · actions that return values · to_text |
| **Entry points** | `init(owners, required)` · `deposit` payable · `propose(to, amount, memo)` · `approve(id)` · `execute(id)` · `proposal` · `has_approved` · `owner_count` |

A group of 2 to 10 owners shares a treasury. Any owner proposes a payment; once `threshold` owners approved it, any owner executes it.

```
# Multi-owner treasury (M-of-N).
# Any owner proposes a payment; it can be executed once `threshold` different
# owners approved it. TCCL has no structs, so each proposal is stored as a set
# of maps that share the same id ("parallel maps").
contract Treasury

const MAX_OWNERS: int = 10

state owners: list[address]
state is_owner: map[address, bool]
state threshold: int
state proposal_count: int
state p_to: map[int, address]
state p_amount: map[int, int]
state p_memo: map[int, text]
state p_approvals: map[int, int]
state p_executed: map[int, bool]
state approved: map[bytes, bool]

event Deposited(from: address, amount: int)
event Proposed(id: int, by: address, to: address, amount: int, memo: text)
event Approved(id: int, by: address, approvals: int)
event Executed(id: int, to: address, amount: int)

init(owner_list: list[address], required: int):
    let n: int = len(owner_list)
    require n >= 2 and n <= MAX_OWNERS, "a treasury needs 2 to 10 owners"
    require required >= 1 and required <= n, "threshold must be 1 to the number of owners"
    for o in owner_list:
```

```
        require not is_owner.has(o), "duplicate owner"
        is_owner[o] = true
        owners.push(o)
    threshold = required

action deposit() payable:
    require value > 0, "attach TCN with --value"
    emit Deposited(caller, value)

action propose(to: address, amount: int, memo: text) -> int:
    only_owner()
    require amount > 0, "amount must be positive"
    require len(memo) <= 140, "memo too long (max 140 bytes)"
    let id: int = proposal_count
    proposal_count += 1
    p_to[id] = to
    p_amount[id] = amount
    p_memo[id] = memo
    emit Proposed(id, caller, to, amount, memo)
    record_approval(id)
    return id

action approve(id: int):
    only_owner()
    require id >= 0 and id < proposal_count, "unknown proposal"
    record_approval(id)

action execute(id: int):
    only_owner()
    require id >= 0 and id < proposal_count, "unknown proposal"
    require not p_executed[id], "already executed"
    require p_approvals[id] >= threshold, "not enough approvals"
    require p_amount[id] <= balance, "treasury balance too low"
    p_executed[id] = true
    send(p_to[id], p_amount[id])
    emit Executed(id, p_to[id], p_amount[id])

view proposal(id: int) -> text:
    require id >= 0 and id < proposal_count, "unknown proposal"
    let state_name: text = "pending"
    if p_executed[id]:
        state_name = "executed"
    elif p_approvals[id] >= threshold:
        state_name = "ready"
    let amount: text = to_text(p_amount[id]) + " motes, "
    let approvals: text = to_text(p_approvals[id]) + " approval(s), "
    return state_name + ": " + amount + approvals + "memo: " + p_memo[id]

view has_approved(id: int, owner: address) -> bool:
    return approved.has(approval_key(id, owner))

view owner_count() -> int:
    return len(owners)

fn record_approval(id: int):
    require not p_executed[id], "already executed"
    let key: bytes = approval_key(id, caller)
    require not approved.has(key), "you already approved this proposal"
    approved[key] = true
    p_approvals[id] += 1
    emit Approved(id, caller, p_approvals[id])

fn only_owner():
    require is_owner.has(caller), "only an owner can do this"

fn approval_key(id: int, owner: address) -> bytes:
    return to_bytes(id) + to_bytes(owner)
```

**How it works.** TCCL has no structs, so a proposal is a set of **parallel maps** sharing an integer id (Section 4.15). Each approval is stored under a composite key `id || owner`, which makes approving twice impossible. `propose` records the proposer's own approval and **returns** the new id: the value appears in the simulator output and in the transaction receipt (`return_value`). Execution is a separate step so that approvals can be collected while the treasury is still being funded: if executing happened automatically inside `approve`, a low balance would make the last approval fail and nobody could record it.

```
$ tccl run treasury.tccl deploy "[tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa,
tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e]" 2
deployed Treasury at daa436158c1dcdc0242085b6dd9451e33496cbee
ok · fuel used: 23138 · height: 2 · alice balance: 100000000000000 motes
$ tccl run treasury.tccl --from dave --value 50tcn call deposit
event Deposited(from: tcr1ugp7r6ctz099mtlqvj6uj9ms9es23wkdrnn3kw, amount: 5000000000)
ok · fuel used: 166 · height: 3 · dave balance: 99995000000000 motes
$ tccl run treasury.tccl --from alice call propose @supplier 20tcn "servers"
event Proposed(id: 0, by: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, to:
tcr1u05f2vn7duwdv8m6e29qxwsfnvdnzyulga8d4y, amount: 2000000000, memo: "servers")
event Approved(id: 0, by: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, approvals: 1)
result: 0
ok · fuel used: 5511 · height: 4 · alice balance: 100000000000000 motes
$ tccl run treasury.tccl --from bob call execute 0
FAILED: requirement failed: not enough approvals · fuel used: 1321 (all changes reverted)
$ tccl run treasury.tccl --from carol call approve 0
event Approved(id: 0, by: tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e, approvals: 2)
ok · fuel used: 2929 · height: 5 · carol balance: 100000000000000 motes
$ tccl run treasury.tccl --from bob call execute 0
event Executed(id: 0, to: tcr1u05f2vn7duwdv8m6e29qxwsfnvdnzyulga8d4y, amount: 2000000000)
ok · fuel used: 3534 · height: 6 · bob balance: 100000000000000 motes
$ tccl run treasury.tccl view proposal 0
result: "executed: 2000000000 motes, 2 approval(s), memo: servers"
ok · fuel used: 1322 · height: 6 · alice balance: 100000000000000 motes
```

**SECURITY TIP**

Owners and threshold are fixed at deployment. That is simple and safe, but a lost key reduces the number of available owners forever — choose `required` so that the group still works if one or two keys are lost, and consider adding owner rotation (itself approved by `threshold` owners).

**Variations.** Add an expiry height to proposals. Let an owner revoke an approval before execution. Add a daily spending limit that a single owner may use without approvals.

## 8.8 Name service

| | |
|---|---|
| **File** | `crates/tccl/examples/names.tccl` |
| **Teaches** | text keys · input validation · byte-level text checks · expiry · rent |
| **Entry points** | `register(name, years)` payable · `renew` payable · `transfer` · `point_to` · `collect_fees` · `resolve` · `whois` · `expires` · `available` |

People rent readable names such as `my-shop` that point to an address — for payments, profiles or contract discovery. Names cost 0.1 TCN per year and become available again when they expire.

```
# A name service: register human-readable names that point to addresses.
# Names are rented by the year; an expired name can be registered again.
contract NameService

const PRICE_PER_YEAR: int = TCN / 10   # 0.1 TCN
const YEAR: int = 525_600              # blocks (1 block = 1 minute)
const MAX_YEARS: int = 10

state admin: address
state owner_of: map[text, address]
state target_of: map[text, address]
state expires_at: map[text, int]
```

```
event Registered(name: text, owner: address, expires: int)
event Renewed(name: text, expires: int)
event Transferred(name: text, from: address, to: address)
event Pointed(name: text, target: address)

init():
    admin = caller

action register(name: text, years: int) payable:
    require valid_name(name), "names have 3 to 32 characters: a-z, 0-9 and '-'"
    require years >= 1 and years <= MAX_YEARS, "register for 1 to 10 years"
    require value == PRICE_PER_YEAR * years, "send exactly 0.1 TCN per year"
    require not is_active(name), "this name is taken"
    owner_of[name] = caller
    target_of[name] = caller
    expires_at[name] = height + YEAR * years
    emit Registered(name, caller, expires_at[name])

# Anyone may pay to renew a name (for example as a gift), but it stays with its owner.
action renew(name: text, years: int) payable:
    require is_active(name), "name not registered or expired"
    require years >= 1 and years <= MAX_YEARS, "renew for 1 to 10 years"
    require value == PRICE_PER_YEAR * years, "send exactly 0.1 TCN per year"
    let new_expiry: int = expires_at[name] + YEAR * years
    require new_expiry <= height + YEAR * MAX_YEARS, "at most 10 years ahead"
    expires_at[name] = new_expiry
    emit Renewed(name, new_expiry)

action transfer(name: text, to: address):
    only_name_owner(name)
    owner_of[name] = to
    emit Transferred(name, caller, to)

action point_to(name: text, target: address):
    only_name_owner(name)
    target_of[name] = target
    emit Pointed(name, target)

action collect_fees():
    require caller == admin, "only the admin"
    require balance > 0, "nothing to collect"
    send(admin, balance)

view resolve(name: text) -> address:
    require is_active(name), "name not registered or expired"
    return target_of[name]

view whois(name: text) -> address:
    require is_active(name), "name not registered or expired"
    return owner_of[name]

view expires(name: text) -> int:
    return expires_at[name]

view available(name: text) -> bool:
    return valid_name(name) and not is_active(name)

fn only_name_owner(name: text):
    require is_active(name), "name not registered or expired"
    require owner_of[name] == caller, "you do not own this name"

fn is_active(name: text) -> bool:
    return owner_of.has(name) and expires_at[name] >= height

fn valid_name(name: text) -> bool:
    let n: int = len(name)
    if n < 3 or n > 32:
        return false
```

```
    let b: bytes = to_bytes(name)
    for i in range(0, n):
        let c: int = b[i]
        let letter: bool = c >= 97 and c <= 122
        let digit: bool = c >= 48 and c <= 57
        if not (letter or digit or c == 45):
            return false
    return true
```

**How it works.** Three maps with `text` keys hold the owner, the target address and the expiry height of each name. `valid_name` converts the name to bytes and checks every byte: only `a–z` (97–122), `0–9` (48–57) and `-` (45) are accepted. `is_active` combines existence and expiry, and every action and view uses it, so an expired name behaves exactly like a free one. Renewals are capped to ten years ahead of the current height, whoever pays for them.

```
$ tccl run names.tccl --from admin deploy
deployed NameService at 3c4598ab69cfc1e089cb3dee7ae82345ed3ee0ca
ok · fuel used: 16254 · height: 2 · admin balance: 100000000000000 motes
$ tccl run names.tccl --from alice --value 0.1tcn call register my-shop 1
event Registered(name: "my-shop", owner: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, expires: 525602)
ok · fuel used: 2638 · height: 3 · alice balance: 99999990000000 motes
$ tccl run names.tccl --from bob --value 0.1tcn call register my-shop 1
FAILED: requirement failed: this name is taken · fuel used: 861 (all changes reverted)
$ tccl run names.tccl --from bob --value 0.1tcn call register My-Shop 1
FAILED: requirement failed: names have 3 to 32 characters: a-z, 0-9 and '-' · fuel used: 107 (all
changes reverted)
$ tccl run names.tccl view resolve my-shop
result: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa
ok · fuel used: 808 · height: 3 · alice balance: 99999990000000 motes
$ tccl run names.tccl --from alice call point_to my-shop @alice-cold
event Pointed(name: "my-shop", target: tcr10eurpm7ulrn0ssxv6q4cy6ctlhv3ndxjxrn5nu)
ok · fuel used: 1514 · height: 4 · alice balance: 99999990000000 motes
$ tccl run names.tccl view resolve my-shop
result: tcr10eurpm7ulrn0ssxv6q4cy6ctlhv3ndxjxrn5nu
ok · fuel used: 808 · height: 4 · alice balance: 99999990000000 motes
$ tccl run names.tccl view available my-shop
result: false
ok · fuel used: 839 · height: 4 · alice balance: 99999990000000 motes
```

> **SECURITY TIP**
> Validate text that becomes a key or is shown to users. Without the character check, `My-Shop`, `my-shop` and names with look-alike Unicode letters would all be different registrations — an invitation to impersonation.

**Variations.** Add reverse resolution (`map[address, text]` of a primary name). Charge more for short names. Send part of the fees to a treasury contract address.

## 8.9 Private payments pool

| | |
|---|---|
| **File** | `crates/tccl/examples/private_pool.tccl` |
| **Teaches** | ring_verify · key images · message binding · relayers · TCCL-PRIV-1 interface |
| **Entry points** | `deposit(public_key)` payable · `withdraw(to, relayer, fee, members, signature, key_image)` · `denomination` · `deposits` · `key_at` · `is_withdrawn` · `message_for` |

Payments on The Coin are public. This contract lets people **break the link** between a deposit and a withdrawal: everyone deposits the same amount together with a fresh public key, and later a withdrawal proves "I own the key of one of these deposits" without revealing which.

```
# Private payments pool (fixed amount) using linkable ring signatures.
#
# 1. deposit(public_key) with exactly DENOMINATION TCN. The public key is a
#    fresh ring key only you control (tccl ring keygen / thecoin-wallet privacy).
# 2. Later, anyone holding the matching secret key withdraws to ANY address by
#    signing with a ring made of several deposited keys. The contract learns
```

```
#    that ONE of them signed — never which one. The key image stops the same
#    deposit from being withdrawn twice.
# 3. A relayer can submit the withdrawal and earn `fee`, so the destination
#    address never needs TCN beforehand.
contract PrivatePool

const DENOMINATION: int = 10 * TCN
const MIN_RING: int = 2
const MAX_RING: int = 32

state keys: list[bytes]
state used: map[bytes, bool]
state withdrawn: int

event Deposited(index: int)
event Withdrawn(ring_size: int)

action deposit(public_key: bytes) payable:
    require value == DENOMINATION, "deposit exactly 10 TCN"
    require len(public_key) == 32, "public key must be 32 bytes"
    keys.push(public_key)
    emit Deposited(len(keys) - 1)

action withdraw(to: address, relayer: address, fee: int, members: list[int], signature: bytes,
key_image: bytes):
    require len(members) >= MIN_RING and len(members) <= MAX_RING, "ring size must be 2 to 32"
    require fee >= 0 and fee < DENOMINATION, "invalid relayer fee"
    require not used.has(key_image), "this deposit was already withdrawn"
    let ring: list[bytes] = []
    for i in members:
        require i >= 0 and i < len(keys), "unknown deposit index"
        ring.push(keys[i])
    let message: bytes = withdraw_message(to, relayer, fee)
    require ring_verify(ring, message, signature, key_image), "invalid ring signature"
    used[key_image] = true
    withdrawn += 1
    send(to, DENOMINATION - fee)
    if fee > 0:
        send(relayer, fee)
    emit Withdrawn(len(members))

# Standard privacy pool interface (TCCL-PRIV-1) used by wallets:
# denomination, deposits, key_at, is_withdrawn, message_for, deposit, withdraw.
view denomination() -> int:
    return DENOMINATION

view deposits() -> int:
    return len(keys)

view is_withdrawn(key_image: bytes) -> bool:
    return used.has(key_image)

view key_at(index: int) -> bytes:
    return keys[index]

view message_for(to: address, relayer: address, fee: int) -> bytes:
    return withdraw_message(to, relayer, fee)

fn withdraw_message(to: address, relayer: address, fee: int) -> bytes:
    return blake3(to_bytes(self) + to_bytes(to) + to_bytes(relayer) + to_bytes(fee))
```

### 8.9.1 Linkable ring signatures in one page

A **ring signature** is made with one secret key over a message and a list of public keys (the **ring**). Anyone can verify that the signature was produced by the owner of **one** of the keys, but not which one. TCCL's `ring_verify(ring, message, signature, key_image)` implements bLSAG ring signatures over the Ristretto255 group:

- each ring key is a 32-byte public key; rings have 1 to 64 members;

- a signature for a ring of **n** keys is 32 × (**n** + 1) bytes;
- the **key image** is a 32-byte value determined by the secret key alone. Signing twice with the same key — even with different rings and messages — produces the same key image, yet the key image reveals nothing about which public key it belongs to.

The pool uses these properties directly:

| Threat | Defence in `private_pool.tccl` |
|---|---|
| Someone who never deposited withdraws | Every ring member must be a deposited key (`keys[i]`), and the signature must verify. |
| A depositor withdraws twice | The key image is stored in `used` before paying; a second withdrawal with the same secret key has the same key image. |
| A relayer or miner redirects the payment | The signed message is `blake3(self ‖ to ‖ relayer ‖ fee)`: changing the destination, the relayer or the fee invalidates the signature. |
| A signature is replayed in another pool | `self` (the pool's address) is part of the message. |
| Amounts link deposits to withdrawals | Every deposit and withdrawal is exactly `DENOMINATION`. |
| The relayer takes everything | `fee` is signed by the depositor and must be below `DENOMINATION`. |

The anonymity of a withdrawal is its **ring**: the other deposits it could have been. A ring of 16 deposits means an observer's best guess is right 1 time in 16.

### 8.9.2 The TCCL-PRIV-1 interface

The reference wallet works with **any** contract that exposes these functions with these exact signatures, so developers can deploy pools with other denominations or extra rules:

| Function | Meaning |
|---|---|
| `view denomination() -> int` | Exact amount of every deposit, in motes. |
| `view deposits() -> int` | Number of deposits so far. |
| `view key_at(index: int) -> bytes` | Public key of deposit `index`. |
| `view is_withdrawn(key_image: bytes) -> bool` | Whether a key image was used. |
| `view message_for(to: address, relayer: address, fee: int) -> bytes` | The message a withdrawal must sign. |
| `action deposit(public_key: bytes) payable` | Deposit `denomination` with a 32-byte ring public key. |
| `action withdraw(to: address, relayer: address, fee: int, members: list[int], signature: bytes, key_image: bytes)` | Withdraw `denomination − fee` to `to` and `fee` to `relayer`; `members` are deposit indexes forming the ring. |

### 8.9.3 Step by step with tccl

`tccl ring keygen` creates a key pair (with `--seed` it is deterministic, which is handy for tests; without it the seed is random). `tccl ring sign` produces a signature and the key image. Here two people deposit, and the second one withdraws to a new address through a relayer who earns 0.01 TCN (1 000 000 motes). Long hex values are shortened with "…", and in the `withdraw` commands `<signature>` and `<key image>` stand for the values printed by `tccl ring sign`:

```
$ tccl ring keygen --seed 0101010101010101010101010101010101010101010101010101010101010101
secret:    2f9d69e5dbc103b0d698b1774ade550b325cd21f93f26018681604250510fa09
public:    0xd897a1359101c28c0369d1b03953b074816aabb6f233216445710bd833ee966a
key image: 0x72991a99188f2295c6198030f174be54fe9d8fcbec18b0bff00865bda8e5ca4f
$ tccl ring keygen --seed 0202020202020202020202020202020202020202020202020202020202020202
secret:    46f9c29758b815b254f8c3bbf7e2a07deb12dba28a741ab42c56a6df386a2000
public:    0x9429de79d361d109030083c4a8b132f4a391f7cb8a571448603bccc113caa133
key image: 0x7a73037b12cdaf516a414c16bab23381d70584f7111142efaff7ced8bd357157
$ tccl run private_pool.tccl deploy
deployed PrivatePool at daa436158c1dcdc0242085b6dd9451e33496cbee
ok · fuel used: 12725 · height: 2 · alice balance: 100000000000000 motes
$ tccl run private_pool.tccl --from alice --value 10tcn call deposit
```

```
0xd897a1359101c28c0369d1b03953b074816aabb6f233216445710bd833ee966a
event Deposited(index: 0)
ok · fuel used: 1696 · height: 3 · alice balance: 99999000000000 motes
$ tccl run private_pool.tccl --from bob --value 10tcn call deposit
0x9429de79d361d109030083c4a8b132f4a391f7cb8a571448603bccc113caa133
event Deposited(index: 1)
ok · fuel used: 1696 · height: 4 · bob balance: 99999000000000 motes
$ tccl run private_pool.tccl view message_for @fresh @relayer 1000000
result: 0x2bf7c3a41b05c5d2f161c77e70998ae50d78bad555ef8ce4d9d8224d2de78076
ok · fuel used: 144 · height: 4 · alice balance: 99999000000000 motes
$ tccl ring sign --secret 46f9c29758…6a2000 --ring 0xd897a13591…ee966a,0x9429de79d3…caa133 --index 1
--message 0x2bf7c3a41b…e78076
signature: 0xddfe677700…ad830a
key image: 0x7a73037b12cdaf516a414c16bab23381d70584f7111142efaff7ced8bd357157
$ tccl run private_pool.tccl --from relayer call withdraw @fresh @relayer 1000000 "[0, 1]" <signature>
<key image>
event Withdrawn(ring_size: 2)
ok · fuel used: 29030 · height: 5 · relayer balance: 100000001000000 motes
$ tccl run private_pool.tccl --from relayer call withdraw @fresh @relayer 1000000 "[0, 1]" <signature>
<key image>
FAILED: requirement failed: this deposit was already withdrawn · fuel used: 304 (all changes reverted)
$ tccl run private_pool.tccl --from relayer call withdraw @thief @relayer 1000000 "[0, 1]" <signature>
<key image of alice>
FAILED: requirement failed: invalid ring signature · fuel used: 26984 (all changes reverted)
```

The last two commands show the defences at work: the same key image is refused, and a signature made for `@fresh` does not authorise a payment to `@thief`.

### 8.9.4 With the wallet

`thecoin-wallet privacy` automates everything: it derives ring keys from your recovery phrase (index `--key`, so you never have to store ring secrets separately), finds your deposit in the pool, picks a random ring among the other deposits, asks the pool for the message, signs it and sends the withdrawal.

| Command | Effect |
|---|---|
| `privacy keygen [--key N]` | Show the ring public key number N of this wallet. |
| `privacy deposit <pool> [--key N]` | Deposit the pool's denomination with ring key N. |
| `privacy status <pool> [--key N]` | Whether ring key N has a deposit and whether it was withdrawn. |
| `privacy withdraw <pool> --to <address> [--key N] [--ring-size 16] [--relayer <address>] [--fee <TCN>]` | Withdraw privately. The ring size is limited to 2–32 and to the number of deposits. |

A session on regtest with two wallets (`alice.json` is the default wallet, Bob uses `-w bob.json`): Alice deploys a pool and makes three deposits with ring keys 0, 1 and 2, Bob makes one, then Bob withdraws to a brand-new address of his own wallet, hidden among all four deposits:

```
$ thecoin-wallet -y contract deploy private_pool.tccl
Fuel:     12725 measured, limit 21542
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   deploy contract PrivatePool (2545 bytes of TCCL)
Fee:      0.00006199 TCN (2704 bytes, priority Normal)
Broadcast OK. txid: 103b67fc2ee5beab4d1fc68bdbb3367d79a503aae2459aeae5fb96508dd6cdf9
Contract address: tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5
$ thecoin-wallet privacy keygen
0x98e822ef513b97ae64f10728aebfd3bb81a4ee8c406e26ef9e73c7b45cab3328
$ thecoin-wallet -y privacy deposit tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5
Fuel:     1808 measured, limit 7350
Preview:  Deposited(index: 0) (simulated on the current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   private deposit of 10 TCN into pool tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 (ring key #0)
Fee:      0.00001323 TCN (223 bytes, priority Normal)
Broadcast OK. txid: 57d930fa43cf83bb696928dd589778d84f03adcd2e5112eece3229a32ab34c54
Keep your recovery phrase: it is the only way to withdraw (key index #0).
$ thecoin-wallet -w bob.json -y privacy deposit tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5
Fuel:     1808 measured, limit 7350
```

```
Preview:  Deposited(index: 1) (simulated on the current state)
From:     tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w
Action:   private deposit of 10 TCN into pool tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 (ring key #0)
Fee:      0.00001323 TCN (223 bytes, priority Normal)
Broadcast OK. txid: a82360e26137844334027c0cf4d96a4e4589825c9e5679a58502aad6814bff32
Keep your recovery phrase: it is the only way to withdraw (key index #0).
$ thecoin-wallet -y privacy deposit tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 --key 1
Fuel:     1808 measured, limit 7350
Preview:  Deposited(index: 2) (simulated on the current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   private deposit of 10 TCN into pool tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 (ring key #1)
Fee:      0.00001323 TCN (223 bytes, priority Normal)
Broadcast OK. txid: 731a96b70a29322f869cfa16df3cb8d3b94b4c30aebe3a3a1f67c2a89b5b7def
Keep your recovery phrase: it is the only way to withdraw (key index #1).
$ thecoin-wallet -y privacy deposit tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 --key 2
Fuel:     1808 measured, limit 7350
Preview:  Deposited(index: 3) (simulated on the current state)
From:     tcr1a9edrkqqhylclwfdaf8sk78zgfkkc0ctvk4awq
Action:   private deposit of 10 TCN into pool tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 (ring key #2)
Fee:      0.00001323 TCN (223 bytes, priority Normal)
Broadcast OK. txid: e840213506c4a358f0bd8f33f16b5281d46c47d6bd7cbed5d0e8d5b7bbfdb285
Keep your recovery phrase: it is the only way to withdraw (key index #2).
$ thecoin-wallet -w bob.json privacy status tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5
Deposit at index 1 (4 deposits in the pool) — available to withdraw
$ thecoin-wallet -w bob.json address --new --label private
tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk
$ thecoin-wallet -w bob.json -y privacy withdraw tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 --to
tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk --ring-size 4
Fuel:     50382 measured, limit 70496
Preview:  Withdrawn(ring_size: 4) (simulated on the current state)
From:     tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w
Action:   private withdrawal to tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk hidden among 4 deposits
Fee:      0.00009589 TCN (521 bytes, priority Normal)
Broadcast OK. txid: 5ded28004d2b979505b56a4c154e172d5f32e23bdd39648c746e72c98709962b
Note: this transaction was sent from your own address tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w. For
stronger privacy, send it from an unrelated address or a relayer.
$ thecoin-wallet -w bob.json privacy status tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5
Deposit at index 1 (4 deposits in the pool) — already withdrawn
$ thecoin-wallet -w bob.json --from 1 balance
Address:   tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk
Balance:   10 TCN
Spendable: 10 TCN
$ thecoin-wallet -w bob.json -y privacy withdraw tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5 --to
tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk --ring-size 4
error: this deposit was already withdrawn
```

The withdrawal transaction reveals the destination, the relayer, the fee, the ring of four deposit indexes, the signature and the key image (both shortened here) — but not which of the four deposits was Bob's:

```
$ thecoin-wallet tx 5ded28004d2b979505b56a4c154e172d5f32e23bdd39648c746e72c98709962b
{
  "txid": "5ded28004d2b979505b56a4c154e172d5f32e23bdd39648c746e72c98709962b",
  "sender": "tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w",
  "fee": 9589,
  "size": 521,
  "action": {
    "type": "invoke",
    "contract": "tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5",
    "function": "withdraw",
    "args": [ "tcr1gaty7ty0krx22lc88ng43wdcqv2ul7a6r5cdsk",
"tcr1v48huqu3407cs77dassple03uy76x8xrz0h49w", "0", "[1, 3, 0, 2]", "0xa98ca3d2a0e3085cacf5d310…",
"0x18cd2ba66109fc3ca25c7a1b…" ],
    "value": 0,
    "max_fuel": 70496,
    "max_deposit": 100000000
  },
  "block_height": 136,
  "confirmations": 4,
  "success": true,
  "error": null,
  "fuel_used": 50382,
```

```
  "burned": 0,
  "logs": [
    {
      "contract": "tcr1sacye32es9v5lsdszu7cqehsna7rs0mn9ntwn5",
      "event": "Withdrawn",
      "fields": [
        [ "ring_size", "4" ]
      ]
    }
  ]
}
```

**SECURITY TIP**

**Privacy is a practice, not a button.**

- Wait until the pool has many deposits before withdrawing; use a large ring.
- Do not withdraw right after depositing: timing links transactions as surely as amounts do.
- The address that **sends** the withdrawal transaction pays its fee and is public. Sending it from the same address that deposited (as Bob did above, and as the wallet warns) links the two. Use a relayer — any address that submits the transaction for you in exchange for `--fee` — or an unrelated address.
- Withdraw to a fresh address and do not immediately send the funds back to your old address.
- Keep your recovery phrase: ring keys are derived from it, and without them a deposit cannot be withdrawn by anyone.

**FUEL TIP**

`ring_verify` costs 5 000 fuel plus 10 000 per ring member, and building the ring reads one storage entry per member. Ring size is a trade-off between privacy and fee; this pool caps it at 32 members.

CHAPTER 9

# Security checklist

TCCL removes many classic smart-contract bugs by design, but it cannot know what your contract is **supposed** to do. Go through this chapter before every deployment that will hold real value.

## 9.1 What the language already guarantees

| You do not need to worry about… | Because |
|---|---|
| Re-entrancy | Contracts cannot call other contracts, and `send` only credits a balance: no code of the receiver runs in the middle of your function. |
| Integer overflow and underflow | All arithmetic is checked; an overflow aborts and reverts the call. |
| Partial failure | A call is atomic: either everything it did is kept, or nothing is. |
| Uninitialised or null values | Every variable has a value; every type has a default. |
| Type confusion | No implicit conversions; arguments are type-checked against the declared parameters. |
| Views changing state | Checked at compile time, transitively through helpers, and again at run time. |
| Accidentally receiving TCN in a function | Functions without `payable` reject value. |
| Infinite loops halting the network | Fuel bounds every call; the caller pays for the fuel reserved. |
| Hidden or changed code | The source is in the deploy transaction and the code is immutable. |

## 9.2 The checklist

### 9.2.1 1. Access control

- ☐ Every action that moves money or changes configuration starts with a `require` on `caller`.
- ☐ Roles (owner, arbiter, admin…) are set in `init` from `caller` or validated arguments — never left at the zero address.
- ☐ Views take user addresses as parameters: `caller` is the zero address inside a view.
- ☐ If a role can be transferred, the new holder is validated (for example not the zero address).

### 9.2.2 2. Money flows

- ☐ For every `send`, you can say who can trigger it, when, and how often.
- ☐ Settlement paths are guarded against running twice (a `settled`/`collected` flag, or deleting the balance entry before sending).
- ☐ Order inside an action is **checks → effects → interactions**: all `require`s first, then state updates, then `send`, then `emit`. TCCL calls are atomic and have no re-entrancy, so this order is about clarity and review, but it keeps functions obviously correct as they evolve.
- ☐ Payouts use your own counters (`raised`, `balances[x]`), not `balance`. Anyone can increase a contract's balance with an ordinary transfer, so `require balance == X` can be broken by a gift.
- ☐ `payable` functions check `value` (exact amount, minimum, or `value > 0`). Do not make functions payable "just in case": TCN sent to a function that does not account for it is stuck unless some other action can move it.
- ☐ Every TCN that can enter the contract can also leave it through some action.

### 9.2.3 3. Integers

- ☐ Amounts are in motes; conversions from whole TCN multiply by `TCN` exactly once.
- ☐ Multiply before dividing (`amount * bp / 10_000`), and check the rounding goes in the contract's favour when splitting payments — the remainder of a division must go somewhere.

- ☐ Negative inputs are rejected where they make no sense (`require amount > 0`). `int` is signed!
- ☐ Products of user-controlled numbers cannot overflow in normal use (an overflow is safe, but it makes the function unusable for those inputs).

#### 9.2.4 4. Denial of service

- ☐ No loop runs over a list or range whose length other users control without a written bound. A list that grows forever eventually makes every function that iterates it cost more than 10 000 000 fuel — permanently.
- ☐ Batch operations take a bounded page (`start`, `count`) instead of processing "everything".
- ☐ One user's failure cannot block others: prefer **pull** payments (each user calls `withdraw`) over **push** loops that pay many addresses in one call.
- ☐ Nothing depends on a specific transaction ordering inside a block.

#### 9.2.5 5. Storage

- ☐ Map entries that are no longer needed are removed, so users recover deposits and state stays small.
- ☐ You are aware that storing a default value deletes an entry, and `has` depends on it.
- ☐ Composite keys cannot be ambiguous (fixed-size parts, or separators).
- ☐ If the contract should end, it can empty its lists and maps and `destroy`.

#### 9.2.6 6. Time and randomness

- ☐ Deadlines use `height`, and the text shown to users says "about" when converting to hours or days.
- ☐ Nothing important depends on being **exactly** at a height: miners choose which transactions go in which block.
- ☐ There is no "randomness" derived from `height`, `caller`, `self` or hashes of them — all of these are known or can be influenced before the transaction is included. Use a commit–reveal scheme where each participant commits `blake3(secret)` first and reveals later, with a penalty for not revealing.

#### 9.2.7 7. Front-running

- ☐ Transactions wait in a public mempool before being mined. Anything a user reveals in arguments (an answer, a secret, a price) can be copied by someone paying a higher fee. Use commit–reveal.
- ☐ Signed messages bind everything that matters: the contract (`self`), the recipient, amounts, fees, and a nonce or key image so they cannot be replayed.

#### 9.2.8 8. Cryptography

- ☐ `verify_ed25519` and `ring_verify` return `false` for malformed input instead of failing: always wrap them in `require`.
- ☐ Ring signature contracts store **used key images** before paying, and check them first.
- ☐ Every ring member is a key the contract accepted earlier (never a key passed freely by the caller).
- ☐ Relayer fees are signed by the user and bounded by the contract.
- ☐ Hash inputs have unambiguous layouts (`to_bytes` of fixed-size values).

#### 9.2.9 9. Text and user input

- ☐ Text lengths are bounded (`len` counts bytes).
- ☐ Text used as a key or shown as an identity is restricted to a safe character set.
- ☐ Error messages tell users what to do, and never promise things the contract does not enforce.

#### 9.2.10 10. Process

- ☐ Every action has tests for its success path **and** for each `require`.
- ☐ The contract ran on testnet with the wallet, including the failure cases.
- ☐ Someone who did not write the contract reviewed it against this checklist.
- ☐ The amount at risk is limited at first (caps in the code), and the plan for a new version is written down — deployed code cannot be patched.

### 9.3 Known limitations of language version 1

These are properties of the current implementation that are easy to trip over:

- `caller` is the zero address inside views, and `value` cannot be used there.
- Contracts cannot call other contracts or read their state; composition happens off-chain, in wallets and applications.
- Maps cannot be iterated, and state lists cannot be copied as a whole.
- Local lists have no `pop`.
- Events cannot be read by contracts.
- In `tccl run`, the `@name` shortcut for test accounts works only for parameters of type `address`, not inside lists.

CHAPTER 10

# Testing contracts

A contract that holds money needs tests for more than the happy path: every `require` is a promise that something **cannot** happen, and each promise deserves a test. This chapter shows three levels of testing, from quick experiments to automated test suites and a private network.

## 10.1 The simulator

`tccl run` executes contracts in an in-memory blockchain that follows the network's rules:

- a deployment runs state initial values and `init`, exactly like the network, and its fuel includes compiling the source (5 per byte);
- the `--value` of a call is moved from the caller to the contract before the code runs, and moved back if the call fails;
- a failed call reverts **everything**: storage, balances, events;
- `send` moves TCN out of the contract balance and fails if the balance is too low;
- `destroy` removes the contract and pays its balance;
- each successful deploy or action advances the height by one; views and failed calls do not;
- every call gets 5 000 000 fuel.

Two things are different from the network: the simulator does not charge fees or storage deposits, and an action's fuel does not include loading the contract (on the network: 100 + 1 per 100 bytes of compiled code, see chapter 6).

## 10.2 Options and test accounts

| Option | Effect |
|---|---|
| `--state <file>` | Simulator state file (default `tccl-state.json`). Use one file per scenario. |
| `--from <name>` | Caller account (default `alice`). Any name works; each starts with 1 000 000 TCN. |
| `--value <amount>` | TCN sent with the call: `5tcn`, `0.25tcn` or a number of motes. |
| `--height <n>` | Set the block height **before** the call. The new height is saved in the state file. |
| `--contract <hex>` | Which deployed contract to call (default: the last one deployed with this state file). |

For parameters of type `address`, `@name` stands for the address of test account `name`: `@bob`, `@shop`, `@fresh-destination`. Inside lists, write real addresses. Test account addresses never change, so you can find them once, for example with the counter:

```
$ tccl run counter.tccl --from alice call increment 1 | head -1
event Increased(by: tcr1zqygp3t5ugluq2uqf7gq60mc9z9t09lpkg42sa, amount: 1, total: 1)
$ tccl run counter.tccl --from bob call increment 1 | head -1
event Increased(by: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 1, total: 2)
$ tccl run counter.tccl --from carol call increment 1 | head -1
event Increased(by: tcr167p23ahe80yxxkzehzj7qmn5plc0ejv638vs6e, amount: 1, total: 3)
```

The state file is plain JSON, useful for inspecting storage or resetting a scenario:

```
$ head -12 tccl-state.json
{
  "height": 5,
  "balances": {
    "100880c574e23fc02b804f900d3f78288ab797e1": 100000000000000,
    "ab0107bce1cbc867a943d06794a7eadce6f32bf6": 100000000000000,
    "d782a8f6f93bc8635859b8a5e06e740ff0fcc99a": 100000000000000
  },
  "contracts": {
    "daa436158c1dcdc0242085b6dd9451e33496cbee": {
      "source": "# The smallest useful contract: a counter anyone can increase.\ncontract
Counter\n\nstate count: int\nstate last_caller: address\n\nevent Increased(by: address, amount: int,
total: int)\n\naction increment(amount: int):\n    require amount > 0, \"amount must be positive\"\n
require amount <= 100, \"at most 100 per call\"\n    count += amount\n    last_caller = caller\n
emit Increased(caller, amount, count)\n\nview get() -> int:\n    return count\n\nview last() ->
address:\n    return last_caller\n",
      "storage": {
        "000000": "000300000000000000000000000000000000",
```

## 10.3 Scripted scenarios

Put a scenario in a shell script so you can re-run it after every change. `tccl run` exits with a non-zero status when a call fails, so `set -e` stops at the first unexpected failure, and `!` marks a call that **must** fail:

```
$ cat refunds.sh
#!/bin/sh
# Crowdfund that misses its goal: backers must get their TCN back, exactly once.
set -e
rm -f tccl-state.json tccl-state.last
tccl run crowdfund.tccl --from maker deploy 100 10
tccl run crowdfund.tccl --from bob --value 4tcn call pledge
tccl run crowdfund.tccl --height 20 view status
tccl run crowdfund.tccl --from bob call refund
! tccl run crowdfund.tccl --from bob call refund    # a second refund must fail
echo "scenario passed"
```

```
$ sh refunds.sh
deployed Crowdfund at 5587c5870fadde50689b36eba05a3a24ec58a5d2
ok · fuel used: 9183 · height: 2 · maker balance: 100000000000000 motes
event Pledged(backer: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 400000000)
ok · fuel used: 1980 · height: 3 · bob balance: 99999600000000 motes
result: "failed"
ok · fuel used: 785 · height: 20 · alice balance: 100000000000000 motes
event Refunded(backer: tcr14vqs008pe0yx022r6pneffl2mnn0x2lkzh7452, amount: 400000000)
ok · fuel used: 1985 · height: 21 · bob balance: 100000000000000 motes
FAILED: requirement failed: nothing to refund · fuel used: 1043 (all changes reverted)
error: call failed
scenario passed
```

## 10.4 Automated tests in Rust

For a contract that matters, write tests with the simulator library that `tccl` itself uses. Every recipe in this book is tested this way in `crates/tccl/tests/examples.rs`; `cargo test -p tccl` runs them all in well under a second. The test for the savings recipe:

```
#[test]
fn savings_are_locked_until_the_chosen_height() {
    let mut sim = Simulator::new();
    let (c, _) = sim.deploy(&src("savings.tccl"), account("anyone"), vec![], 0).unwrap();
    let unlock = sim.height as i128 + 1_000;
    sim.call(&c, account("saver"), "deposit", vec![int(unlock)], 7 *
TCN).unwrap().result.unwrap();
    sim.call(&c, account("saver"), "deposit", vec![int(unlock + 10)], 3 *
TCN).unwrap().result.unwrap();
    fails_with(sim.call(&c, account("saver"), "deposit", vec![int(unlock)], TCN).unwrap(), "you
```

```
cannot shorten an existing lock");
    fails_with(sim.call(&c, account("saver"), "deposit", vec![int(1)], TCN).unwrap(), "the
unlock height must be in the future");
    fails_with(sim.call(&c, account("saver"), "withdraw", vec![], 0).unwrap(), "your savings are
still locked");
    fails_with(sim.call(&c, account("other"), "withdraw", vec![], 0).unwrap(), "you have no
savings here");
    assert_eq!(sim.view(&c, "balance_of", vec![addr("saver")]).unwrap().result.unwrap(), int(10
* TCN as i128));
    sim.height = unlock as u64 + 10;
    assert_eq!(sim.view(&c, "blocks_left", vec![addr("saver")]).unwrap().result.unwrap(),
int(0));
    let before = sim.balance_of(&account("saver"));
    sim.call(&c, account("saver"), "withdraw", vec![], 0).unwrap().result.unwrap();
    assert_eq!(sim.balance_of(&account("saver")), before + 10 * TCN);
    assert_eq!(sim.view(&c, "total", vec![]).unwrap().result.unwrap(), int(0));
    fails_with(sim.call(&c, account("saver"), "withdraw", vec![], 0).unwrap(), "you have no
savings here");
}
```

The API is small:

| Call | Returns |
|---|---|
| `Simulator::new()` | An empty chain at height 1. |
| `sim.deploy(source, deployer, args, value)` | `(address, CallResult)`; the contract is removed again if `init` fails. |
| `sim.call(address, caller, function, args, value)` | `CallResult` with `result`, `fuel_used` and `events`. |
| `sim.view(address, function, args)` | `CallResult` of a read-only call. |
| `sim.balance_of(address)` | Balance in motes. |
| `sim.height` | Public field: set it to move time. |
| `tccl::sim::account(name)` | The 20-byte address of a test account. |
| `tccl::ring::keypair_from_seed`, `tccl::ring::sign` | Ring keys and signatures for privacy tests. |

Arguments are `tccl::program::Values`: `Value::Int`, `Value::Bool`, `Value::Text`, `Value::Bytes`, `Value::Address` and `Value::List`. A failed `require` is `Err(VmError::Require(message))`, so tests can check the exact message.

## 10.5 A private network

Before testnet, you can run a private **regtest** network on your machine. It has trivial proof of work, mines very fast, and unlocks mining rewards after a few seconds:

```
$ thecoin-wallet --network regtest -w dev.json create
$ thecoind --network regtest --data-dir ./regtest-node --miner-address <your tcr1 address> --threads 1
$ thecoin-wallet --network regtest -w dev.json balance
```

The node's API listens on `127.0.0.1:27334` by default for regtest, which is also the wallet's default for `--network regtest`. After about half a minute the wallet has spendable coins and every command of chapter 7 works. Regtest uses lower fee and deposit parameters than mainnet (chapter 6), so fuel numbers are identical but fees are ten times smaller.

## 10.6 What to test

For every contract:

1. **Deployment:** valid arguments, and each invalid one (every `require` in `init`).
2. **Every action's happy path**, checking balances and state afterwards — not just "it did not fail".
3. **Every `require`**, with the exact message, called by the wrong person, at the wrong time, with the wrong amount.
4. **Repeated calls:** twice in a row (double withdraw, double vote, double settle).

5. **Boundaries:** zero, one, the maximum, the maximum plus one, exactly at the deadline height and one block after.
6. **Money conservation:** the sum of what went in equals what went out plus what remains.
7. **Failed calls change nothing:** after a failure, views return the same values as before.
8. **Fuel:** the most expensive call with the largest allowed input stays far below 10 000 000.

APPENDIX A

# Language reference

## A.1 Grammar

The grammar in EBNF. `NEWLINE`, `INDENT` and `DEDENT` come from indentation: a line indented more than the previous one opens a block, a line indented less closes blocks until the indentation matches an enclosing level. Line breaks inside `( )` and `[ ]` are ignored. Blank lines and comment-only lines do not affect indentation.

```
contract   = "contract" NAME NEWLINE { item } ;
item       = const | state | event | function ;
const      = "const" NAME ":" type "=" expr NEWLINE ;
state      = "state" NAME ":" type [ "=" expr ] NEWLINE ;
event      = "event" NAME "(" [ params ] ")" NEWLINE ;
function   = ( "init" | ( "action" | "view" | "fn" ) NAME )
             "(" [ params ] ")" [ "->" type ] [ "payable" ] ":" block ;
params     = NAME ":" type { "," NAME ":" type } ;
type       = "int" | "bool" | "text" | "bytes" | "address"
           | "list" "[" type "]" | "map" "[" type "," type "]" ;
block      = NEWLINE INDENT statement { statement } DEDENT ;

statement  = "let" NAME ":" type "=" expr NEWLINE
           | target ( "=" | "+=" | "-=" | "*=" ) expr NEWLINE
           | "if" expr ":" block { "elif" expr ":" block } [ "else" ":" block ]
           | "while" expr ":" block
           | "for" NAME "in" ( "range" "(" expr "," expr ")" | expr ) ":" block
           | ( "break" | "continue" | "pass" ) NEWLINE
           | "return" [ expr ] NEWLINE
           | "require" expr [ "," expr ] NEWLINE
           | "send" "(" expr "," expr ")" NEWLINE
           | "emit" NAME "(" [ args ] ")" NEWLINE
           | "destroy" "(" expr ")" NEWLINE
           | expr NEWLINE ;                      (* only calls and pop() *)
target     = NAME | NAME "[" expr "]" ;

expr       = and_expr { "or" and_expr } ;
and_expr   = not_expr { "and" not_expr } ;
not_expr   = "not" not_expr | comparison ;
comparison = sum [ ( "==" | "!=" | "<" | "<=" | ">" | ">=" ) sum ] ;
sum        = product { ( "+" | "-" ) product } ;
product    = unary { ( "*" | "/" | "%" ) unary } ;
unary      = "-" unary | postfix ;
postfix    = primary { "[" expr "]" | "." NAME "(" [ args ] ")" } ;
primary    = INT | TEXT | BYTES | "true" | "false"
           | NAME [ "(" [ args ] ")" ] | "(" expr ")" | "[" [ args ] "]" ;
args       = expr { "," expr } ;

INT        = digit { digit | "_" } ;
BYTES      = "0x" hexdigit hexdigit { hexdigit hexdigit | "_" } ;
TEXT       = '"' { character | "\n" | "\t" | '\"' | "\\" } '"' ;
NAME       = ( letter | "_" ) { letter | digit | "_" } ;
```

## A.2 Keywords and reserved names

| Group | Names |
|---|---|
| Declarations | `contract` `const` `state` `event` `init` `action` `view` `fn` `payable` |

| Group | Names |
|---|---|
| Statements | `let` `if` `elif` `else` `while` `for` `in` `break` `continue` `return` `require` `send` `emit` `destroy` `pass` |
| Operators and literals | `and` `or` `not` `true` `false` |
| Reserved built-in names | `caller` `value` `balance` `height` `self` `TCN` `int` `bool` `text` `bytes` `address` `list` `map` `len` `sha256` `blake3` `to_bytes` `to_text` `to_int` `min` `max` `abs` `slice` `verify_ed25519` `ring_verify` `address_of` `zero_address` `range` |

## A.3 Types and conversions

| Type | Default | Map key | `==` | Initial value in `state` |
|---|---|---|---|---|
| `int` | `0` | yes | yes | yes |
| `bool` | `false` | yes | yes | yes |
| `text` | `""` | yes | yes | yes |
| `bytes` | `0x` (empty) | yes | yes | yes |
| `address` | zero address | yes | yes | yes |
| `list[T]` | `[]` | no | no | no |
| `map[K, V]` | empty | no | no | no; state variables only |

| From | To | How |
|---|---|---|
| `int` | `text` | `to_text(n)` |
| `int`, `address`, `text`, `bool` | `bytes` | `to_bytes(x)` |
| `bytes` (≤ 15) | `int` | `to_int(b)`, unsigned big-endian |
| `text` | `bytes` (hash) | `sha256(t)`, `blake3(t)` |
| `bytes` (32-byte key) | `address` | `address_of(pk)` |

## A.4 Methods

| Method | On | Notes |
|---|---|---|
| `.len()` | local lists, `text`, `bytes`, state lists | Same as `len(x)`. |
| `.push(v)` | local lists, state lists | Statement. |
| `.pop()` | state lists | Statement or expression; returns the removed item. |
| `.has(k)` | maps | Expression, `bool`. |
| `.remove(k)` | maps | Statement. |

## A.5 Context values and built-in functions

| Name | Type | Value |
|---|---|---|
| `caller` | `address` | Signer of the transaction (zero address in views) |
| `value` | `int` | Motes sent with the call (not in views) |
| `balance` | `int` | Contract balance in motes, including `value` |
| `height` | `int` | Block height of the call |
| `self` | `address` | This contract's address |
| `TCN` | `int` | Constant 100 000 000 |

| Signature | Fuel beyond expression cost |
|---|---|
| `len(list \| text \| bytes) -> int` | — (state lists: one storage read) |
| `min(int, int) -> int`, `max(int, int) -> int`, `abs(int) -> int` | — |

| Signature | Fuel beyond expression cost |
|---|---|
| `to_text(int) -> text` | — |
| `to_bytes(int \| address \| text \| bool \| bytes) -> bytes` | — |
| `to_int(bytes) -> int` | — |
| `slice(bytes, int, int) -> bytes` | — |
| `sha256(bytes \| text) -> bytes` | 60 + 20 per 64 bytes |
| `blake3(bytes \| text) -> bytes` | 60 + 20 per 64 bytes |
| `verify_ed25519(bytes, bytes, bytes) -> bool` | 3 500 + 1 per 64 bytes of message |
| `ring_verify(list[bytes], bytes, bytes, bytes) -> bool` | 5 000 + 10 000 per member |
| `address_of(bytes) -> address` | — |
| `zero_address() -> address` | — |
| `address(text literal) -> address` | compile time |

## A.6 Language limits

| Limit | Value |
|---|---|
| Source code size | 48 000 bytes |
| Nesting of blocks, parentheses and types | 32 levels |
| Operators of one precedence level chained in one expression | 64 |
| Depth of an expression tree | 128 |
| Chained indexes and method calls on one value (`a[i][j]…`) | 32 |
| Functions per contract | 256 |
| State variables per contract | 256 |
| Local variables (including parameters) per function | 1 024 |
| Call depth | 16 |
| Size of one `text`, `bytes` or list value | 65 536 bytes |
| Items in a local list or list literal | 4 096 |
| Ring size in `ring_verify` | 64 keys |
| List nesting in values decoded from transactions | 64 |
| `to_int` input | 15 bytes |

## A.7 Semantics in brief

- **Integers:** 128-bit signed; `+ - *` unary `-` and `abs` fail on overflow; `/` truncates toward zero; `%` has the sign of the left operand; division or remainder by zero fails.
- **Boolean operators** `and` and `or` short-circuit.
- **Text** is UTF-8; `len` and `slice`-like operations work on bytes; `+` concatenates.
- **Evaluation order** is left to right.
- **Storage:** default values are never stored; writing a default deletes the entry.
- **Atomicity:** any runtime error reverts the whole call.
- **Views** run read-only; any attempt to write fails with `state cannot be modified in a view`.
- **Deployment** writes constant initial values of state variables, then runs `init` (if present).

APPENDIX B

# Error messages

Compile errors are reported as `file:line L:C: message`. Runtime errors are the `error` of a failed call — in the simulator after `FAILED:`, on the network in the receipt and in the wallet's `contract call would fail:` message.

## B.1 Compile errors

| Message | Cause and fix |
|---|---|
| `tabs are not allowed; indent with spaces` | A tab character somewhere in the file. |
| `indentation does not match any outer block` | A line is dedented to a level that was never used. Align it with an enclosing block. |
| `unexpected indentation at top level` | A declaration is indented. |
| `empty block (use 'pass')` | A `:` line with nothing indented below it. |
| `expected … found …` | Syntax error. The message names what the parser expected, e.g. `expected ':' and a type (variables must declare their type), found assign`. |
| `invalid number literal` | Letters after digits, e.g. `5tcn`. Write `5 * TCN`. |
| `hex bytes literal must have an even number of digits` | `0xabc` — each byte needs two hex digits. |
| `unterminated text literal`, `unknown escape \x` | Missing closing `"`, or an escape other than `\n \t \" \\`. |
| `this bracket is never closed`, `unbalanced closing bracket` | Mismatched `( )` or `[ ]`. |
| `source too large (N bytes, max 48000)` | Split the contract or shorten comments. |
| `nesting too deep (max 32)` | Too many nested blocks, parentheses or type brackets. |
| `too many operators in one expression (max 64); split it with 'let'` | A very long chain like `a + b + c + …`. |
| `expression too deeply nested (max depth 128); split it with 'let'` | An expression tree that is too deep. |
| `chained comparisons are not allowed; use 'and'` | `a < b < c`. |
| `'x' is a reserved name` | A built-in name used for a declaration. |
| `'x' is already declared`, `… at contract level` | Two declarations with the same name, or a local reusing a contract-level name. |
| `variable 'x' is already declared` | A `let` or loop variable reuses a visible local or parameter. |
| `unknown name 'x'`, `unknown variable 'x'`, `unknown function 'x'`, `unknown event 'x'` | Typo or missing declaration. |
| `unknown type 'x' (types: …)` | Only `int bool text bytes address list[T] map[K, V]` exist. |
| `maps can only be used as state variables` | A map as parameter, local, list item, map value or event field. |
| `map keys must be int, bool, text, bytes or address, not …` | Lists cannot be keys. |
| `expected T, found U` | Type mismatch; convert explicitly (`to_text`, `to_bytes`…). |

| Message | Cause and fix |
|---|---|
| `cannot add A and B`, `arithmetic needs int operands…`, `ordering comparisons need int operands…` | Operators apply only to the types listed in Section 4.7. |
| `cannot compare A with B`, `cannot compare values of type list[…]` | `==` needs two values of the same scalar type. |
| `function 'f' must return a T on every path` | End the function with `return`, or an `if/else` whose branches all return. |
| `this function must return a T`, `this function does not return a value (declare '-> type')` | `return` without/with a value in the wrong kind of function. |
| `a view must declare a return type ('-> type')` | Add `-> type` to the view. |
| `a view cannot change state` / `send TCN` / `emit events` | Views are read-only; use an action. |
| `view 'f' changes state, sends TCN or emits events (directly or through a fn it calls)` | A helper called by the view mutates; split the helper. |
| `'value' is not available in a view` | Views never receive TCN. |
| `only action and init can be payable` | Remove `payable` from the view or `fn`. |
| `only one init() is allowed`, `init cannot return a value` | — |
| `'f' is an entry point; only 'fn' helpers can be called from code` | Move shared code into an `fn`. |
| `f() takes N argument(s), M given` | Wrong number of arguments. |
| `event 'E' has N fields, M given` | `emit` arguments must match the event. |
| `destroy() can only be used inside an action` | — |
| `break/continue outside of a loop` | — |
| `only int, bool, text, bytes and address state variables can have an initial value` | Lists and maps start empty. |
| `'x' is not a constant (constant values can only use literals and other constants)` | A constant uses a state variable or a later constant. |
| `'x' is a constant and cannot be changed` | — |
| `'x' is a map; use x[key] or x.has(key)`, `'x' is a state list; use x[i], len(x) or 'for x in x'` | Whole maps and state lists cannot be used as values. |
| `cannot assign a whole list[…]; change its items instead` | State lists and maps are changed item by item. |
| `only items of lists and maps can be assigned with [ ]` | e.g. `grid[0][1] = 5` or indexing a `text`. |
| `compound assignment is not defined for T` | `+=` works for `int`, `text`, `bytes`; `-=` and `*=` only for `int`. |
| `this expression (T) does nothing as a statement` | A value computed and thrown away, e.g. `x + 1` alone on a line. |
| `pop() is only available on state lists` | Local lists have no `pop`. |
| `an empty list needs a known type, e.g. 'let xs: list[int] = []'` | `[]` where the element type cannot be known. |
| `range() can only be used in 'for i in range(start, end)'`, `range needs two arguments: range(start, end)` | — |
| `address(...) takes one text literal…`, `invalid address literal`, `address prefix 'x' is not valid on this network` | Check the address and use the prefix of the target network. |

| Message | Cause and fix |
|---|---|
| `too many functions` / `state variables` / `local variables` | See Appendix A.6. |
| `a contract needs at least one init, action or view` | A contract with only helpers can never run. |

## B.2 Runtime errors

| Error | Meaning |
|---|---|
| `requirement failed: <message>` | A `require` was false. Without a message: `requirement at line N failed`. |
| `out of fuel` | The fuel limit was reached. Raise `--max-fuel` or make the code cheaper. |
| `integer overflow` | An arithmetic result outside the 128-bit range. |
| `division by zero` | `/` or `%` by 0. |
| `index I out of bounds (length N)` | List or bytes index outside `0 … N-1`, `pop` on an empty list (index −1) or a bad `slice`. |
| `value too large` | A text, bytes or list over 64 KiB, a local list over 4 096 items, or more than 64 events in one call. |
| `call depth limit reached` | More than 16 nested function calls. |
| `unknown function 'f'` | The contract has no function with that name. |
| `function 'f' cannot be called this way` | An action called as a view or the other way round, or a helper called from outside. |
| `wrong arguments: …` | Wrong number or types of arguments; ring over 64 keys; `to_int` over 15 bytes; `address_of` without 32 bytes. |
| `function does not accept TCN (not payable)` | `value` sent to a function that is not payable. |
| `state cannot be modified in a view` | Run-time guard for views. |
| `invalid amount` | `send` with zero, a negative or a too large amount. |
| `insufficient contract balance` | `send` for more than the contract holds. |
| `contract cannot be destroyed while it still has storage (N entries)` | Empty lists and maps before `destroy`. |

## B.3 Network and wallet errors

| Error | Meaning |
|---|---|
| `compile error: …` | The deploy transaction's source does not compile on the network (e.g. an address with another network's prefix). |
| `compiled program too large (N bytes, max 262144)` | — |
| `out of fuel while compiling`, `out of fuel while loading the contract` | `max_fuel` too low even to start. |
| `storage deposit of N motes exceeds max_deposit M (contract state: B bytes)` | Raise `--max-deposit`. |
| `insufficient funds for value: need N, spendable M` | The sender cannot pay the `value`. |
| `insufficient funds for storage deposit: need N, spendable M` | The sender cannot pay the deposit. |
| `no contract at …` | Wrong address, or the contract was destroyed. |
| `contract call would fail: … (nothing was sent)` | Wallet simulation failed; nothing was broadcast. |

| Error | Meaning |
|---|---|
| `transaction would be rejected: …` | The transaction is invalid (fee, nonce, balance, size…). |
| `f is not payable; remove --value` | Wallet check before simulation. |
| `init expects N argument(s): …`, `f expects N argument(s): …` | Wrong number of command-line arguments. |
| `argument 'x': …` | An argument could not be parsed with its declared type. |
| `max_fuel must be 1..=10000000` | Invalid `--max-fuel`. |

APPENDIX C

# Command reference

## C.1 tccl

| Command | Description |
|---|---|
| `tccl check <file>` | Compile and print the interface. |
| `tccl abi <file>` | Print the interface as JSON (`name`, `kind`, `payable`, `params`, `returns`). |
| `tccl run <file> [options] deploy [args…]` | Deploy in the simulator. |
| `tccl run <file> [options] call <function> [args…]` | Call an action in the simulator. |
| `tccl run <file> [options] view <function> [args…]` | Call a view in the simulator. |
| `tccl ring keygen [--seed <hex32>]` | Print a ring secret key, public key and key image. |
| `tccl ring sign --secret <hex> --ring <pk1,pk2,…> --index <i> --message <0xhex>` | Ring-sign a message with the key at position `i` of the ring. |
| `tccl --version` | Tool and language version. |

## C.2 thecoin-wallet contract and privacy commands

| Command | Description |
|---|---|
| `contract deploy <file> [args…] [--value TCN] [--max-fuel N] [--max-deposit TCN]` | Deploy a TCCL contract; prints the contract address. |
| `contract invoke <address> <function> [args…] [--value TCN] [--max-fuel N] [--max-deposit TCN]` | Call an action. |
| `contract view <address> <function> [args…]` | Query a view (free). |
| `contract program <address>` | Balance, storage, deposit and interface of a contract. |
| `tx <txid>` | Transaction and receipt as JSON. |
| `fees` | Current fee parameters, congestion and priority levels. |
| `privacy keygen [--key N]` | Ring public key N of the wallet. |
| `privacy deposit <pool> [--key N]` | Deposit into a TCCL-PRIV-1 pool. |
| `privacy status <pool> [--key N]` | State of the deposit made with key N. |
| `privacy withdraw <pool> --to <address> [--key N] [--ring-size N] [--relayer <address>] [--fee TCN]` | Private withdrawal. |

Defaults: `--max-deposit 1` (TCN); `--max-fuel` measured by simulation (fuel × 1.3 + 5 000); `--ring-size 16`; `--key 0`. The wallet's global options are listed in Section 7.1.

## C.3 Node API

| Endpoint | Description |
|---|---|
| `GET /api/v1/program/{address}` | Contract metadata and interface. |

| Endpoint | Description |
|---|---|
| `POST /api/v1/program/{address}/view` | `{"function": "…", "args": ["…"]}` → `{"result", "error", "fuel_used"}`; fuel limit 2 000 000. |
| `POST /api/v1/tx/simulate` | `{"tx": "<hex>"}` → `valid`, `invalid_reason`, `success`, `error`, `fuel_used`, `required_fee`, `logs`, `return_value`, `program`. |
| `POST /api/v1/tx` | `{"tx": "<hex>"}` → `{"txid"}`. |
| `GET /api/v1/tx/{txid}` | Transaction view with receipt: `success`, `error`, `fuel_used`, `logs`, `return_value`, `program`, `burned`. |
| `GET /api/v1/fees` | Fee parameters, congestion, storage deposit and priority levels. |