Control Structures

Control Structures

Copy page

Learn how to control code flow in Solidity.

Solidity supports many familiar control structures, but these come with additional restrictions and considerations due to the cost of gas and the necessity of setting a maximum amount of gas that can be spent in a given transaction.

Objectives

By the end of this lesson you should be able to:

- Control code flow with `if`, `else`, `while`, and `for`
- List the unique constraints for control flow in Solidity
- Utilize `require` to write a function that can only be used when a variable is set to `true`

Write a `revert` statement to abort execution of a function in a specific state

Utilize `error` to control flow more efficiently than with `require`

Control Structures

Solidity supports the basic conditional and iterative **control structures** found in other curly bracket languages, but it **does not** support more advanced statements such as `switch`, `forEach`, `in`, `of`, etc.

Solidity does support `try` / `catch`, but only for calls to other contracts.

> ⚠ **Yul** is an intermediate-level language that can be embedded in Solidity contracts and is documented within the docs for Solidity. Yul **does** contain the `switch` statement, which can confuse search results.

Conditional Control Structure Examples

The `if`, `else if`, and `else`, statements work as expected. Curly brackets may be omitted for single-line bodies, but we recommend avoiding this as it is less explicit.

```
function ConditionalExample(uint _number) external pure returns (string mem
    if(_number == 0) {
        return "The number is zero.";
    } else if(_number % 2 == 0) {
        return "The number is even and greater than zero.";
    } else {
        return "The number is odd and is greater than zero.";
    }
}
```

Iterative Control Structures

The `while`, `for`, and `do`, keywords function the same as in other languages. You can use `continue` to skip the rest of a loop and start the next iteration. `break` will terminate execution of the loop, and you can use `return` to exit the function and return a value at any point.

> You can use `console.log` by importing `import "hardhat/console.sol";`. Doing so will require you to mark otherwise `pure` contracts as `view`.

```
uint times; // Default value is 0!
for(uint i = 0; i <= times; i++) {
    console.log(i);
}

uint timesWithContinue;
for(uint i = 0; i <= timesWithContinue; i++) {
    if(i % 2 == 1) {
        continue;
    }
    console.log(i);
}

uint timesWithBreak;
for(uint i = 0; i <= timesWithBreak; i++) {
    // Always stop at 7
    if(i == 7) {
        break;
    }
    console.log(i);
}

uint stopAt = 10;
while(stopAt <= 10) {
    console.log(i);
    stopAt++;
}

uint doFor = 10;
```

```
do {
    console.log(i);
    doFor++;
} while(doFor <= 10);
```

Error Handling

Solidity contains a set of relatively unique, built-in functions and keywords to handle **errors**. They ensure certain requirements are met, and completely abort all execution of the function and revert any state changes that occurred during function execution. You can use these functions to help protect the security of your contracts and limit their execution.

The approach may seem different than in other environments. If an error occurs partly through a high-stakes transaction such as transferring millions of dollars of tokens, you **do not** want execution to carry on, partially complete, or swallow any errors.

Revert and Error

The `revert` keyword halts and reverses execution. It must be paired with a custom `error`. Revert should be used to prevent operations that are logically valid, but

should not be allowed for business reasons. It is **not** a bug if a `revert` is triggered. Examples where `revert` and `error` would be used to control operations include:

- Allowing only certain senders to access functionality
- Preventing the withdrawal of a deposit before a certain date
- Allowing inputs under certain state conditions and denying them under others

Custom `error` s can be declared without parameters, but they are much more useful if you include them:

```
error OddNumberSubmitted(uint _first, uint _second);
function onlyAddEvenNumbers(uint _first, uint _second) public pure returns
    if(_first % 2 != 0 || _second % 2 != 0) {
        revert OddNumberSubmitted(_first, _second);
    }
    return _first + _second;
}
```

When triggered, the `error` provides the values in the parameters provided. This information is very useful when debugging, and/or to transmit information to the front end to share what has happened with the user:

```
call to HelloWorld.onlyAddEvenNumbers errored: VM error: revert

revert
    The transaction has been reverted to the initial state.
Error provided by the contract:
OddNumberSubmitted
Parameters:
{
 "_first": {
  "value": "1"
 },
 "_second": {
  "value": "2"
 }
}
Debug the transaction to get more information.
```

You'll also encounter `revert` used as a function, returning a string error. This legacy pattern has been retained to maintain compatibility with older contracts:

```
function oldRevertAddEvenNumbers(uint _first, uint _second) public pure ret
    if(_first % 2 != 0 || _second % 2 != 0) {
        // Legacy use of revert, do not use
        revert("One of the numbers is odd");
    }
    return _first + _second;
}
```

The error provided is less helpful:

```
call to HelloWorld.oldRevertAddEvenNumbers errored: VM error: revert.

revert
    The transaction has been reverted to the initial state.
The reason provided by the contract: "One of the numbers is odd".
Debug the transaction to get more information.
```

Require

The `require` function is falling out of favor because it uses more gas than the pattern above. You should still become familiar with it because it is present in innumerable contracts, tutorials, and examples.

`require` takes a logical condition and a string error as arguments. It is more gas efficient to separate logical statements if they are not interdependent. In other words,

don't use `&&` or `||` in a `require` if you can avoid it.

For example:

```
function requireAddEvenNumbers(uint _first, uint _second) public pure retur
    // Legacy pattern, do not use
    require(_first % 2 == 0, "First number is not even");
    require(_second % 2 == 0, "Second number is not even");

    return _first + _second;
}
```

The output error message will be the first one that fails. If you were to submit `1` , and `3` to this function, the error will only contain the first message:

```
call to HelloWorld.requireAddEvenNumbers errored: VM error: revert.
```



```
Debug the transaction to get more information.
```


Assert and Panic

The `assert` keyword throws a `panic` error if triggered. A `panic` is the same type of error that is thrown if you try to divide by zero or access an array out-of-bounds. It is used for testing internal errors and should never be triggered by normal operations, even with flawed input. You have a bug that should be resolved if an assert throws an exception:

```
function ProcessEvenNumber(uint _validatedInput) public pure {
    // If assert triggers, input validation has failed.  This should never
    // happen!
    assert(_validatedInput % 2 == 0);
    // Do something...
}
```

The output here isn't as helpful, so you may wish to use one of the patterns above instead.

```
call to HelloWorld.ProcessEvenNumber errored: VM error: revert.

revert
    The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value
Debug the transaction to get more information.
```

Conclusion

In this lesson, you've learned how to control code flow with standard conditional and iterative operators. You've also learned about the unique keywords Solidity uses to generate errors and reset changes if one of them has been triggered. You've been exposed to both newer and legacy methods of writing errors, and learned the difference between `assert` and `require`.

‹ Require, Revert, and Error

Exercise ›