Errors

Error Triage

Copy page

Learn how to identify and resolve common errors in Solidity.

Objectives

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

> Debug common solidity errors including transaction reverted, out of gas, stack overflow, value overflow/underflow, index out of range, etc.

Compiler Errors

Compiler errors are manifold but almost always very easy to debug, since the error message usually tells you what is wrong and how to fix it.

Type Errors

You will get a compiler error if you try to assign a literal to the wrong type.

```
// Bad code example, do not use
function compilerTypeError() public pure returns (uint) {
    uint myNumber = "One";
    return myNumber;
}
```

```
from solidity:
TypeError: Type literal_string "One" is not implicitly convertible to expec
 --> contracts/ErrorTriage.sol:8:9:
  |
8 |         uint myNumber = "One";
  |         ^^^^^^^^^^^^^^^^^^^^^
```

Fix by correcting the type or value, as appropriate for your needs:

Reveal code

Conversion Errors

Conversion errors occur when you attempt to *implicitly* convert one type to another. Solidity only allows this under very narrow circumstances where there is no possibility of ambiguous interpretation of the data.

```
// Bad code example, do not use
function compilerConversionError() public pure returns (uint) {
    int8 first = 1;

    return first;
}
```

```
from solidity:
TypeError: Return argument type int8 is not implicitly convertible to expec
  --> contracts/ErrorTriage.sol:15:16:
   |
15 |         return first;
   |                ^^^^^
```

Fix by explicitly casting as necessary:

Reveal code

You'll commonly need to use multiple conversions to bridge from one type to another.

Operator Errors

You cannot use operators between types as flexibly as you may be used to.

```
// Bad code example, do not use
function compilerOperatorError() public pure returns (uint) {
    int8 first = 1;
    uint256 second = 2;

    uint sum = first + second;

    return sum;
}
```

Operator errors are often paired with a type error.

```
from solidity:
TypeError: Operator + not compatible with types int8 and uint256.
  --> contracts/ErrorTriage.sol:22:20:
   |
22 |         uint sum = first + second;
   |                    ^^^^^^^^^^^^^^

from solidity:
TypeError: Type int8 is not implicitly convertible to expected type uint256
  --> contracts/ErrorTriage.sol:22:9:
   |
22 |         uint sum = first + second;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^
```

Resolve by explicitly converting to the final type:

Reveal code

Stack Depth Limit

The **EVM stack** has 1024 slots, but only the top 16 slots are accessible. As a result, you can only have fewer than 16 variables in scope at one time.

⚠ Other items can also use up these slots. You are **not** guaranteed 15 slots, it can be lower.

```
// Bad code example, do not use
function stackDepthLimit() public pure returns (uint) {
        uint first = 1;
        uint second = 2;
        uint third = 3;
        uint fourth = 4;
        uint fifth = 5;
        uint sixth = 6;
        uint seventh = 7;
        uint eighth = 8;
        uint ninth = 9;
        uint tenth = 10;
        uint eleventh = 11;
        uint twelfth = 12;
        uint thirteenth = 13;
        uint fourteenth = 14;
        uint fifteenth = 15;
        uint sixteenth = 16;

        return first +
                second +
                third +
                fourth +
                fifth +
                sixth +
                seventh +
                eighth +
                ninth +
                tenth +
                        eleventh +
                        twelfth +
                        thirteenth +
                        fourteenth +
                        fifteenth +
                        sixteenth;
    }
```

```
from solidity:
CompilerError: Stack too deep. Try compiling with --via-ir (cli) or the equ
  --> contracts/ErrorTriage.sol:92:17:
   |
92 |                 eighth +
   |                 ^^^^^^
```

Resolve this error by breaking up large functions and separating operations into different levels of scope.

Reveal code

Logical Errors

Logical errors occur when your code is syntactically correct, but still results in a data state that is a violation of the rules of the language.

A **panic** occurs when your code tries to do an illegal operation. These return with a very basic error code, which Remix unfortunately hides. However, it makes up for that annoyance by providing a very powerful debugger.

⚠ The Remix VM doesn't behave exactly the same as true onchain operations, so note that these errors will not behave exactly the same if triggered while testing with Hardhat, or called from a front end.

caution

For each of these examples, copy them into Remix to explore with the debugger on your own.

Array Index Out-of-Bounds

A panic will be triggered if you try to access an array at an invalid index.

```
// Bad code example, do not use
function badGetLastValue() public pure returns (uint) {
    uint[4] memory arr = [uint(1), 2, 3, 4];

    return arr[arr.length];
}
```

Running this function will result in the following error in the console:

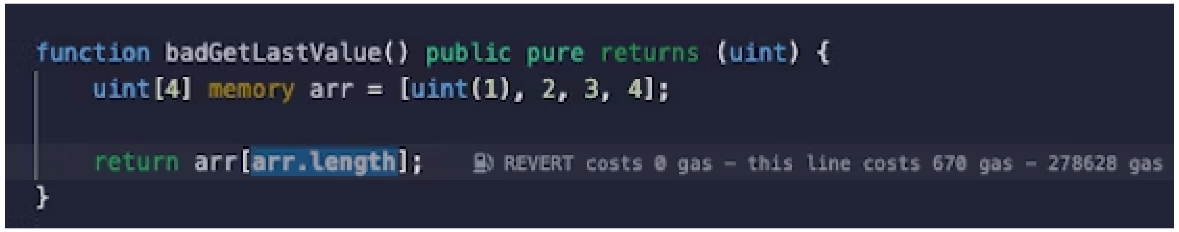
```
call to ErrorTriage.badGetLastValue errored: VM error: revert.

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.
```

Click the *Debug* button to open the debugger.

The debugger contains panels with information about variables in storage, memory, what's on the stack, and so on. You can also add breakpoints to lines of code to further help with debugging.

One of the most useful features is the link near the top instructing you to *"Click here to jump where the call reverted."*

Click that link and the debugger will jump to the point of failure, **and highlight the code that caused the panic.** Neat!

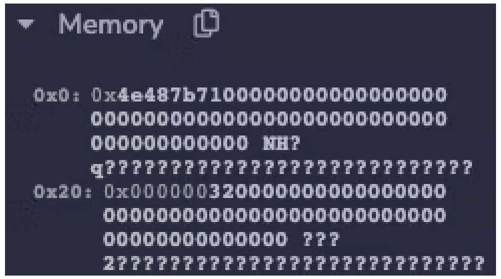


```
function badGetLastValue() public pure returns (uint) {
    uint[4] memory arr = [uint(1), 2, 3, 4];

    return arr[arr.length];    REVERT costs 0 gas - this line costs 670 gas - 278628 gas
}
```

You can find the specific error here, but it's difficult.

Look in the *Memory* panel. The first item at `0x0` has a hash starting with `0x4e487b71`. This code indicates a panic.

The second item, at `0x20` has the error code of `32` hidden in it, which is for array out-of-bounds.



```
0x0: 0x4e487b7100000000000000000
     000000000000000000000000000
     000000000000 NH?
     q????????????????????????????
0x20: 0x000000320000000000000000
      00000000000000000000000000
      00000000000000 ???
      2????????????????????????????
```

It’s sometimes better to just review the code first to see if the error is obvious.

```
function badGetLastValueFixed() public pure returns (uint) {
    uint[4] memory arr = [uint(1), 2, 3, 4];

    return arr[arr.length-1];
}
```

Out of Gas

The default settings for Remix make it difficult to trigger an out of gas error because the VM will often crash first. For this example, go to the *Deploy & Run Transactions* tab and reduce the gas limit to **300000**.

If you write code that can have an ambiguous execution time, it becomes very difficult to accurately estimate gas limits.

In this example, each loop has a 1 in 1000 chance of ending.

⚠ `block.timestamp` can be manipulated. **DO NOT** use this as a source of randomness if any value can be derived from one outcome over another!

```
// Bad code example, do not use
function badRandomLoop() public view returns (uint) {
    uint seed = 0;
    // DO NOT USE THIS METHOD FOR RANDOM NUMBERS!!! IT IS EASILY EXPLOITABL
    while(uint(keccak256(abi.encodePacked(block.timestamp, seed))) % 1000 !
        seed++;
        // ...do something
    }

    return seed;
}
```

Run this function a few times. Often, it will work just fine. Other times, an error appears:

```
call to ErrorTriage.badLoop errored: VM error: out of gas.

out of gas
    The transaction ran out of gas. Please increase the Gas Limit.

Debug the transaction to get more information.
```

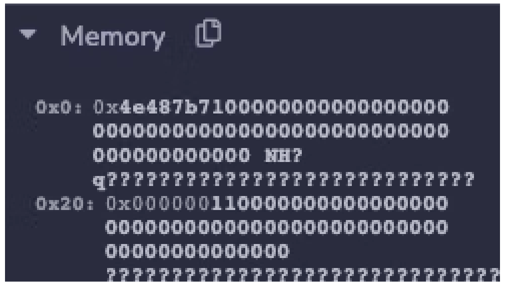
The error message here is a bit misleading. You do **not** usually want to fix this by increasing the gas limit. If you're getting a gas error because the transaction didn't estimate for enough gas, it's better to refactor for better predictability.

```
function badRandomLoopFixed() public view returns (uint) {
    // DO NOT USE THIS METHOD FOR RANDOM NUMBERS!!! IT IS EASILY EXPLOITABL
    uint times = uint(keccak256(abi.encodePacked(block.timestamp))) % 1000;

    for(uint i = 0; i <= times; i++) {
        // ...do something
    }

    return times;
}
```

The `uint` type will *panic* in the event of an overflow or underflow.

```
function badSubtraction() public pure returns (uint) {
    uint first = 1;
    uint second = 2;
    return first - second;
}
```

As before, you can see the panic code and panic type in *memory*.



```
0x0: 0x4e487b7100000000000000000
     000000000000000000000000000
     000000000000 NH?
     q?????????????????????????????
0x20: 0x0000001100000000000000000
      00000000000000000000000000
      00000000000000
      ??????????????????????????????
```

In this case, the error type is `11`, for overflow/underflow outside of an `unchecked` block.

Fix by changing your [illegible] es.

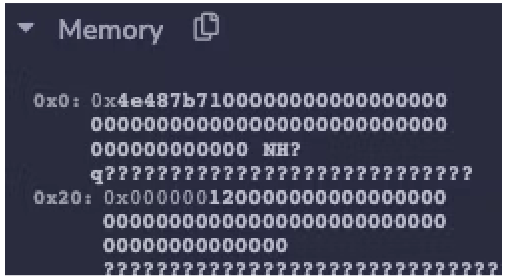
Ask a question... Ctrl+I

Reveal code

Divide by Zero

Divide by zero errors also trigger a panic, with a code of `12`.

```
function badDivision() public pure returns (uint) {
    uint first = 1;
    uint second = 0;
    return first / second;
}
```



```
0x0: 0x4e487b7100000000000000000
     000000000000000000000000000
     000000000000 NH?
     q????????????????????????????
0x20: 0x00000012000000000000000000
      00000000000000000000000000
      00000000000000
      ??????????????????????????????
```

Don’t divide by zero.

Conclusion

In this lesson, you reviewed the causes of and solutions for a number of compiler errors and logical errors that you may encounter.

Was this page helpful? Yes No Suggest edits Raise issue

‹ Error Triage Exercise ›