2.3.4.8.4.5 Special Methods

Return to Operation Data

Overview

Return to the Top

Some Object-Oriented Programming (OOP) Languages have the ability to overload built-in methods defined by the language. Some languages such as ALGOL, FORTRAN, R, and Scala allow all operators to be overwritten and even new operators to be defined. Others languages such as Ada, C#, C++, PHP, Python, Ruby, and Rust all a limited set of operators to be overwritten. C, Go, Java, JavaScript, and Visual Basic are examples that do not allow operators to be overwritten. [1)]

Method Overloading, , sometimes termed **Operator Overloading**, or **operator ad hoc polymorphism**, is a specific case of polymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by a programming language, a programmer, or both.

Operator Overloading is “syntactic sugar”, and is used because it allows programming using notation nearer to the target domain and allows user-defined types a similar level of syntactic support as types built into a language. It is common, for example, in scientific computing, where it allows computing representations of mathematical objects to be manipulated with the same syntax as on paper.

Operator overloading does not change the expressive power of a language (with functions), as it can be emulated using function calls. For example, consider variables **a**, **b**, **c**, **d** of some user-defined type, such as matrices:

Example 1:

```
matrix a, b, c, d;
d = a + b * c;
```

In a language that supports operator overloading, and with the usual assumption that the '*' operator has higher precedence than the '+' operator, this is a concise way of writing:

Example 2:

```
matrix a, b, c, d;
d = Add ( a, Multiply ( b, c ) );
```

However, the Example 1 syntax reflects common mathematical usage, but it can obfuscate the underlying fact that **a**, **b**, **c**, **d** are actually matrices that can be a problem during debugging and

performance tuning.

DIDO Specifics

Return to the Top

Ethereum's Solidity does not support **Operator Overloading** directly, but it does support being able to create libraries of reusable **`functions`** that support a way to provide operators for specific types.

Probably the most well known example of reusable library is the **SafeMath library**[2]. In essence, the purpose of the **SafeMath Library** is to provide Overflow, Underflow, and Wrap Around protection while using Solidity's **Unsigned Integer** (**`uint`**).

Classic OO Example

Return to the Top

The following classic Object-Oriented (OO) code example is written in Solidity. The Smart Contract does the following:

- A **`Counter`** smart contract is made (Line 4)
- It defines an Unsigned Integer (**`uint256`**) as a **State Variable** named **`counter`** and initializes it to zero. (Line 6)

> **Note:** In Solidity, all Unsigned Integers are initialized to zero, but it is best to be explicit.

3. The contract defines two methods that manipulate the value of **`counter`**:
   - A **`public`** function named **`increment`** by one (Line 8)
   - A **`public`** function named **`decrement`** by one (Line 13)
4. Calling the **SafeMath** function to modify the **`unit256`** state variable **`counter`** (Lines 10 and 15)

```
pragma solidity ^0.7.1;
// SPDX-License-Identifier: MIT

contract Counter
{  // Use SafeMath code
   uint256 counter = 0;
   function incrment()
     public
   { counter = SafeMath.add ( counter, 1 );
   } // End incrment function
   function decrement()
     public
   { counter = SafeMath.add ( counter, 1 );
```

```
   } // End decrement function
} // End Counter contract
```

However, this code can have problems at the range edges of the uint256 type causing Overflow, Underflow, and Wrap Around issues. The original **`Counter`** Smart contract can be modified to check for the **overflow** or **underflow** issue every time the **`counter`** is manipulated, but in-line checking can create its own issues. For example, it introduces more lines of code and each line of code has the potential for being incorrect. A very classic error occurs when the code is copied and pasted from someplace else and all the references to the variable are not changed correctly.

Classic SafeMath External Library Example

Return to the Top

A solution for these problems was introduced using an external **`library`** called **SafeMath** and the Solidity language **`using`** statement. By including this **`using SafeMath for <BaseType>`** at the beginning of the Smart Contract and replacing the traditional operators **+**, **-**, *****, **/**, **%** with the following **SafeMath** defined functions: **`add`**, **`sub`**, **`mul`**, **`div`** and **`mod`**.

The **`Counter`** Smart Contract is modified to use **SafeMath**:

- A **`using`** for **`uint256`** datatype statement is added (Line 5)
- Adding the **SafeMath** function calls to the **`unit256`** state variable **`counter`** instead of using explicitly calling the **SafeMath** functions (Lines 10 and 15)

```
pragma solidity ^0.7.1;
// SPDX-License-Identifier: MIT

contract Counter
{ using SafeMath for uint256;
  uint256 counter = 0;
     function incrment()
       public
     { counter = counter.add(1);
     } // End incrment function
    function decrement()
      public
    { counter = counter.sub(1);
    } // End incrment function
} // End Counter contract
```

Newer SafeMath Library Example

Return to the Top

Starting with Solidity 0.8 release, **SafeMath** is now integrated into the language and the import of the **OpenZeppelin SafeMath** is no longer needed. the standard mathematical operators (i.e., **+**, **-**, *****, **/**, **%**) now use the SafeMath library. For Example: **`a + b`** now automatically invokes **`revert`** on Overflow.

In the event that the Wrap Around behavior is the desired part of the design of the contract, the expression can be surrounded with the **`unchecked`** block. For example,

- Using SafeMath (i.e., Solidity 8.0 or later) causes a **`revert`** to ocur when **SafeMath** is used (Line 12)
- Subtracting 1 from the minimum value of an **`uint256`** (i.e., 0) as initialized in Line 10, causes a Wrap Around returning a value of **`type(uint256).max`**
- Deactivate **SafeMath** by placing the statement within a **`unchecked`** block (i.e., **{** and **}**) (Line 12)

```
pragma solidity ^0.8.1;
// SPDX-License-Identifier: MIT

contract MyContract
{
  function test()
    external
    pure
    returns(uint256)
    { uint256 x = 0;
      unchecked { x--; }
      return x;
    } // End test function
} // End MyContract
```

☐ [char]Review

1)
Wikipedia, Operator overloading, Accessed: 4 November 2021, https://en.wikipedia.org/wiki/Operator_overloading

2)
OpenZeppelin.com, SafeMath, Accessed: 27 December 2021, https://docs.openzeppelin.com/contracts/2.x/api/math