Solidity Documentation

Release 0.4.9-develop

Ethereum

February 13, 2017

Contents

Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript and it is designed to target the Ethereum Virtual Machine.

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

As you will see, it is possible to create contracts for voting, crowdfunding, blind auctions, multi-signature wallets and more.

Note: The best way to try out Solidity right now is using the Browser-Based Compiler (it can take a while to load, please be patient).

CHAPTER 1

Useful links

- Ethereum
- Changelog
- Story Backlog
- Source Code
- Ethereum Stackexchange
- Gitter Chat

CHAPTER 2

Available Solidity Integrations

- **Browser-Based Compiler** Browser-based IDE with integrated compiler and Solidity runtime environment without server-side components.
- **Ethereum Studio** Specialized web IDE that also provides shell access to a complete Ethereum environment.
- **Visual Studio Extension** Solidity plugin for Microsoft Visual Studio that includes the Solidity compiler.
- **Package for SublimeText — Solidity language syntax** Solidity syntax highlighting for SublimeText editor.
- **Atom Ethereum interface** Plugin for the Atom editor that features syntax highlighting, compilation and a runtime environment (requires backend node).
- **Atom Solidity Linter** Plugin for the Atom editor that provides Solidity linting.
- **Solium** A commandline linter for Solidity which strictly follows the rules prescribed by the Solidity Style Guide.
- **Visual Studio Code extension** Solidity plugin for Microsoft Visual Studio Code that includes syntax highlighting and the Solidity compiler.
- **Emacs Solidity** Plugin for the Emacs editor providing syntax highlighting and compilation error reporting.
- **Vim Solidity** Plugin for the Vim editor providing syntax highlighting.
- **Vim Syntastic** Plugin for the Vim editor providing compile checking.

Discontinued:

- **Mix IDE** Qt based IDE for designing, debugging and testing solidity smart contracts.

CHAPTER 3

Solidity Tools

- **Dapple** Package and deployment manager for Solidity.
- **Solidity REPL** Try Solidity instantly with a command-line Solidity console.
- **solgraph** Visualize Solidity control flow and highlight potential security vulnerabilities.
- **evmdis** EVM Disassembler that performs static analysis on the bytecode to provide a higher level of abstraction than raw EVM operations.

CHAPTER 4

Third-Party Solidity Parsers and Grammars

- **solidity-parser** Solidity parser for JavaScript
- **Solidity Grammar for ANTLR 4** Solidity grammar for the ANTLR 4 parser generator

CHAPTER 5

Language Documentation

On the next pages, we will first see a *simple smart contract* written in Solidity followed by the basics about *blockchains* and the *Ethereum Virtual Machine*.

The next section will explain several *features* of Solidity by giving useful *example contracts* Remember that you can always try out the contracts in your browser!

The last and most extensive section will cover all aspects of Solidity in depth.

If you still have questions, you can try searching or asking on the Ethereum Stackexchange site, or come to our gitter channel. Ideas for improving Solidity or this documentation are always welcome!

See also Russian version ( ).

CHAPTER 6

Contents

Keyword Index, Search Page

6.1 Introduction to Smart Contracts

6.1.1 A Simple Smart Contract

Let us begin with the most basic example. It is fine if you do not understand everything right now, we will go into more detail later.

Storage

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get() constant returns (uint) {
        return storedData;
    }
}
```

The first line simply tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality (up to, but not including, version 0.5.0). This is to ensure that the contract does not suddenly behave differently with a new compiler version.

A contract in the sense of Solidity is a collection of code (its functions) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line `uint storedData;` declares a state variable called `storedData` of type `uint` (unsigned integer of 256 bits). You can think of it as a single slot in a database that can be queried and altered by calling functions of the code that manages the database. In the case of Ethereum, this is always the owning contract. And in this case, the functions `set` and `get` can be used to modify or retrieve the value of the variable.

To access a state variable, you do not need the prefix `this.` as is common in other languages.

This contract does not yet do much apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without (feasible) a way to prevent you from publishing this

number. Of course, anyone could just call `set` again with a different value and overwrite your number, but the number will still be stored in the history of the blockchain. Later, we will see how you can impose access restrictions so that only you can alter the number.

Subcurrency Example

The following contract will implement the simplest form of a cryptocurrency. It is possible to generate coins out of thin air, but only the person that created the contract will be able to do that (it is trivial to implement a different issuance scheme). Furthermore, anyone can send coins to each other without any need for registering with username and password - all you need is an Ethereum keypair.

```
pragma solidity ^0.4.0;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount);
    }
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type address that is publicly accessible. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts or keypairs belonging to external persons. The keyword `public` automatically generates a function that allows you to access the current value of the state variable. Without this keyword, other contracts have no way to access the variable and only the code of this contract can write to it. The function will look something like this:

```
function minter() returns (address) { return minter; }
```

Of course, adding a function exactly like that will not work because we would have a function and a state variable with the same name, but hopefully, you get the idea - the compiler figures that out for you.

The next line, `mapping (address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The type maps addresses to unsigned integers. Mappings can be seen as hashtables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation

is all zeros. This analogy does not go too far, though, as it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. So either keep in mind (or better, keep a list or use a more advanced data type) what you added to the mapping or use it in a context where this is not needed, like this one. The accessor function created by the `public` keyword is a bit more complex in this case. It roughly looks like the following:

```
function balances(address _account) returns (uint) {
    return balances[_account];
}
```

As you see, you can use this function to easily query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares a so-called "event" which is fired in the last line of the function `send`. User interfaces (as well as server appliances of course) can listen for those events being fired on the blockchain without much cost. As soon as it is fired, the listener will also receive the arguments `from`, `to` and `amount`, which makes it easy to track transactions. In order to listen for this event, you would use

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
}
```

Note how the automatically generated function `balances` is called from the user interface.

The special function `Coin` is the constructor which is run during creation of the contract and cannot be called afterwards. It permanently stores the address of the person creating the contract: `msg` (together with `tx` and `block`) is a magic global variable that contains some properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

Finally, the functions that will actually end up with the contract and can be called by users and contracts alike are `mint` and `send`. If `mint` is called by anyone except the account that created the contract, nothing will happen. On the other hand, `send` can be used by anyone (who already has some of these coins) to send coins to anyone else. Note that if you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the fact that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By the use of events it is relatively easy to create a "blockchain explorer" that tracks transactions and balances of your new coin.

6.1.2 Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, ...) are just there to provide a certain set of features and promises. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon's AWS works internally in order to use it?

Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make

(assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

Blocks

One major obstacle to overcome is what, in Bitcoin terms, is called a "double-spend attack": What happens if two transactions exist in the network that both want to empty an account, a so-called conflict?

The abstract answer to this is that you do not have to care. An order of the transactions will be selected for you, the transactions will be bundled into what is called a "block" and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time and that is where the word "blockchain" derives from. Blocks are added to the chain in rather regular intervals - for Ethereum this is roughly every 17 seconds.

As part of the "order selection mechanism" (which is called "mining") it may happen that blocks are reverted from time to time, but only at the "tip" of the chain. The more blocks that are added on top, the less likely it is. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

6.1.3 The Ethereum Virtual Machine

Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called "nonce").

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in "Wei" to be exact) which can be modified by sending transactions that include Ether.

Transactions

A transaction is a message that is sent from one account to another account (which might be the same or the special zero-account, see below). It can include binary data (its payload) and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is the zero-account (the account with the address `0`), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the "nonce"). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code.

Gas

Upon creation, each transaction is charged with a certain amount of **gas**, whose purpose is to limit the amount of work that is needed to execute the transaction and to pay for this execution. While the EVM executes the transaction, the gas is gradually depleted according to specific rules.

The **gas price** is a value set by the creator of the transaction, who has to pay `gas_price * gas` up front from the sending account. If some gas is left after the execution, it is refunded in the same way.

If the gas is used up at any point (i.e. it is negative), an out-of-gas exception is triggered, which reverts all modifications made to the state in the current call frame.

Storage, Memory and the Stack

Each account has a persistent memory area which is called **storage**. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract and it is comparatively costly to read and even more so, to modify storage. A contract can neither read nor write to any storage apart from its own.

The second memory area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (ie. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on an area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words. The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signalled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions "bubble up" the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it finished execution, it can return data which will be stored at a location in the caller's memory preallocated by the caller.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls.

Delegatecall / Callcode and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the "library" feature in Solidity: Reusable library code that can be applied to a contract's storage in order to e.g. implement a complex data structure.

Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement **events**. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in bloom filters, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain ("light clients") can still find these logs.

Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

Self-destruct

The only possibility that code is removed from the blockchain is when a contract at that address performs the `selfdestruct` operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state.

> **Warning:** Even if a contract's code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

Note: The pruning of old contracts may or may not be implemented by Ethereum clients. Additionally, archive nodes could choose to keep the contract storage and code indefinitely.

Note: Currently **external accounts** cannot be removed from the state.

6.2 Installing Solidity

6.2.1 Versioning

Solidity versions follow semantic versioning and in addition to releases, **nightly development builds** are also made available. The nightly builds are not guaranteed to be working and despite best efforts they might contain undocumented and/or broken changes. We recommend using the latest release. Package installers below will use the latest release.

6.2.2 Browser-Solidity

If you just want to try Solidity for small contracts, you can try browser-solidity which does not need any installation. If you want to use it without connection to the Internet, you can go to https://github.com/ethereum/browser-solidity/tree/gh-pages and download the .ZIP file as explained on that page.

6.2.3 npm / Node.js

This is probably the most portable and most convenient way to install Solidity locally.

A platform-independent JavaScript library is provided by compiling the C++ source into JavaScript using Emscripten for browser-solidity and there is also an npm package available.

To install it, simply use

```
npm install solc
```

Details about the usage of the Node.js package can be found in the solc-js repository.

6.2.4 Docker

We provide up to date docker builds for the compiler. The `stable` repository contains released versions while the `nightly` repository contains potentially unstable changes in the develop branch.

```
docker run ethereum/solc:stable solc --version
```

Currenty, the docker image only contains the compiler executable, so you have to do some additional work to link in the source and output directories.

6.2.5 Binary Packages

Binary packages of Solidity available at solidity/releases.

We also have PPAs for Ubuntu. For the latest stable version.

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

If you want to use the cutting edge developer version:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

Homebrew is missing pre-built bottles at the time of writing, following a Jenkins to TravisCI migration, but Homebrew should still work just fine as a means to build-from-source. We will re-add the pre-built bottles soon.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
brew linkapps solidity
```

6.2.6 Building from Source

Clone the Repository

To clone the source code, execute the following command:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

If you want to help developing Solidity, you should fork Solidity and add your personal fork as a second remote:

```
cd solidity
git remote add personal git@github.com:[username]/solidity.git
```

Solidity has git submodules. Ensure they are properly loaded:

```
git submodule update --init --recursive
```

Prerequisites - macOS

For macOS, ensure that you have the latest version of Xcode installed. This contains the Clang C++ compiler, the Xcode IDE and other Apple development tools which are required for building C++ applications on OS X. If you are installing Xcode for the first time, or have just installed a new version then you will need to agree to the license before you can do command-line builds:

```
sudo xcodebuild -license accept
```

Our OS X builds require you to install the Homebrew package manager for installing external dependencies. Here's how to uninstall Homebrew, if you ever want to start again from scratch.

Prerequisites - Windows

You will need to install the following dependencies for Windows builds of Solidity:

Software	Notes
Git for Windows	Command-line tool for retrieving source from Github.
CMake	Cross-platform build file generator.
Visual Studio 2015	C++ compiler and dev environment.

External Dependencies

We now have a "one button" script which installs all required external dependencies on macOS, Windows and on numerous Linux distros. This used to be a multi-step manual process, but is now a one-liner:

```
./scripts/install_deps.sh
```

Or, on Windows:

```
scripts\install_deps.bat
```

Command-Line Build

Building Solidity is quite similar on Linux, macOS and other Unices:

```
mkdir build
cd build
cmake .. && make
```

And even on Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 14 2015 Win64" ..
```

This latter set of instructions should result in the creation of **solidity.sln** in that build directory. Double-clicking on that file should result in Visual Studio firing up. We suggest building **RelWithDebugInfo** configuration, but all others work.

Alternatively, you can build for Windows on the command-line, like so:

```
cmake --build . --config RelWithDebInfo
```

6.2.7 The version string in detail

The Solidity version string contains four parts: - the version number - pre-release tag, usually set to `develop.YYYY.MM.DD` or `nightly.YYYY.MM.DD` - commit in the format of `commit.GITHASH` - platform has arbitrary number of items, containing details about the platform and compiler

If there are local modifications, the commit will be postfixed with `.mod`.

These parts are combined as required by Semver, where the Solidity pre-release tag equals to the Semver pre-release and the Solidity commit and platform combined make up the Semver build metadata.

A relase example: `0.4.8+commit.60cc1668.Emscripten.clang`.

A pre-release example: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

6.2.8 Important information about versioning

After a release is made, the patch version level is bumped, because we assume that only patch level changes follow. When changes are merged, the version should be bumped according to semver and the severity of the change. Finally, a release is always made with the version of the current nightly build, but without the `prerelease` specifier.

Example:

0. the 0.4.0 release is made
1. nightly build has a version of 0.4.1 from now on
2. non-breaking changes are introduced - no change in version
3. a breaking change is introduced - version is bumped to 0.5.0
4. the 0.5.0 release is made

This behaviour works well with the version pragma.

6.3 Solidity by Example

6.3.1 Voting

The following contract is quite complex, but showcases a lot of Solidity's features. It implements a voting contract. Of course, the main problems of electronic voting is how to assign voting rights to the correct persons and how to prevent manipulation. We will not solve all problems here, but at least we will show how delegated voting can be done so that vote counting is **automatic and completely transparent** at the same time.

The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually.

The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust.

At the end of the voting time, `winningProposal()` will return the proposal with the largest number of votes.

```
pragma solidity ^0.4.0;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal
    {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;
```

```
    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    function Ballot(bytes32[] proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) {
        if (msg.sender != chairperson || voters[voter].voted) {
            // `throw` terminates and reverts all changes to
            // the state and to Ether balances. It is often
            // a good idea to use this if functions are
            // called incorrectly. But watch out, this
            // will also consume all provided gas.
            throw;
        }
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) {
        // assigns reference
        Voter sender = voters[msg.sender];
        if (sender.voted)
            throw;

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (
            voters[to].delegate != address(0) &&
            voters[to].delegate != msg.sender
```

```
        ) {
            to = voters[to].delegate;
        }

        // We found a loop in the delegation, not allowed.
        if (to == msg.sender) {
            throw;
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter delegate = voters[to];
        if (delegate.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) {
        Voter sender = voters[msg.sender];
        if (sender.voted)
            throw;
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() constant
            returns (uint winningProposal)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() constant
```

```
            returns (bytes32 winnerName)
    {
        winnerName = proposals[winningProposal()].name;
    }
}
```

Possible Improvements

Currently, many transactions are needed to assign the rights to vote to all participants. Can you think of a better way?

6.3.2 Blind Auction

In this section, we will show how easy it is to create a completely blind auction contract on Ethereum. We will start with an open auction where everyone can see the bids that are made and then extend this contract into a blind auction where it is not possible to see the actual bid until the bidding period ends.

Simple Open Auction

The general idea of the following simple auction contract is that everyone can send their bids during a bidding period. The bids already include sending money / ether in order to bind the bidders to their bid. If the highest bid is raised, the previously highest bidder gets her money back. After the end of the bidding period, the contract has to be called manually for the beneficiary to receive his money - contracts cannot activate themselves.

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address public beneficiary;
    uint public auctionStart;
    uint public biddingTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change
    bool ended;

    // Events that will be fired on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
```

```
    /// beneficiary address `_beneficiary`.
    function SimpleAuction(
        uint _biddingTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingTime = _biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
        // be able to receive Ether.
        if (now > auctionStart + biddingTime) {
            // Revert the call if the bidding
            // period is over.
            throw;
        }
        if (msg.value <= highestBid) {
            // If the bid is not higher, send the
            // money back.
            throw;
        }
        if (highestBidder != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it can be prevented by the caller by e.g.
            // raising the call stack to 1023. It is always safer
            // to let the recipient withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() returns (bool) {
        var amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
```

```
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() {
        // It is a good guideline to structure functions that interact
        // with other contracts (i.e. they call functions or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be perfromed multiple times.
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. Conditions
        if (now <= auctionStart + biddingTime)
            throw; // auction did not yet end
        if (ended)
            throw; // this function has already been called

        // 2. Effects
        ended = true;
        AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        if (!beneficiary.send(highestBid))
            throw;
    }
}
```

Blind Auction

The previous open auction is extended to a blind auction in the following. The advantage of a blind auction is that there is no time pressure towards the end of the bidding period. Creating a blind auction on a transparent computing platform might sound like a contradiction, but cryptography comes to the rescue.

During the **bidding period**, a bidder does not actually send her bid, but only a hashed version of it. Since it is currently considered practically impossible to find two (sufficiently long) values whose hash values are equal, the bidder commits to the bid by that. After the end of the bidding period, the bidders have to reveal their bids: They send their values unencrypted and the contract checks that the hash value is the same as the one provided during the bidding period.

Another challenge is how to make the auction **binding and blind** at the same time: The only way to prevent the bidder from just not sending the money after he won the auction is to make her send it together with the bid. Since value transfers cannot be blinded in Ethereum, anyone can see the value.

The following contract solves this problem by accepting any value that is at least as large as the bid. Since this can of course only be checked during the reveal phase, some bids might be **invalid**, and this is on purpose (it even provides an explicit flag to place invalid bids with high value transfers): Bidders can confuse competition by placing several high or low invalid bids.

```
pragma solidity ^0.4.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address public beneficiary;
    uint public auctionStart;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modifiers are a convenient way to validate inputs to
    /// functions. `onlyBefore` is applied to `bid` below:
    /// The new function body is the modifier's body where
    /// `_` is replaced by the old function body.
    modifier onlyBefore(uint _time) { if (now >= _time) throw; _; }
    modifier onlyAfter(uint _time) { if (now <= _time) throw; _; }

    function BlindAuction(
        uint _biddingTime,
        uint _revealTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// Place a blinded bid with `_blindedBid` = keccak256(value,
    /// fake, secret).
    /// The sent ether is only refunded if the bid is correctly
    /// revealed in the revealing phase. The bid is valid if the
    /// ether sent together with the bid is at least "value" and
    /// "fake" is not true. Setting "fake" to true and sending
    /// not the exact amount are ways to hide the real bid but
    /// still make the required deposit. The same address can
    /// place multiple bids.
    function bid(bytes32 _blindedBid)
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
```

```
        }));
    }

    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    function reveal(
        uint[] _values,
        bool[] _fake,
        bytes32[] _secret
    )
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        if (
            _values.length != length ||
            _fake.length != length ||
            _secret.length != length
        ) {
            throw;
        }

        uint refund;
        for (uint i = 0; i < length; i++) {
            var bid = bids[msg.sender][i];
            var (value, fake, secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != keccak256(value, fake, secret)) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bid.deposit;
            if (!fake && bid.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bid.blindedBid = 0;
        }
        if (!msg.sender.send(refund))
            throw;
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != 0) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
```

```
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Withdraw a bid that was overbid.
    function withdraw() returns (bool) {
        var amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns (see the remark above about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)){
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        onlyAfter(revealEnd)
    {
        if (ended)
            throw;
        AuctionEnded(highestBidder, highestBid);
        ended = true;
        // We send all the money we have, because some
        // of the refunds might have failed.
        if (!beneficiary.send(this.balance))
            throw;
    }
}
```

6.3.3 Safe Remote Purchase

```
pragma solidity ^0.4.0;

contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    function Purchase() payable {
        seller = msg.sender;
        value = msg.value / 2;
        if (2 * value != msg.value) throw;
    }
```

```
    modifier require(bool _condition) {
        if (!_condition) throw;
        _;
    }

    modifier onlyBuyer() {
        if (msg.sender != buyer) throw;
        _;
    }

    modifier onlySeller() {
        if (msg.sender != seller) throw;
        _;
    }

    modifier inState(State _state) {
        if (state != _state) throw;
        _;
    }

    event aborted();
    event purchaseConfirmed();
    event itemReceived();

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        onlySeller
        inState(State.Created)
    {
        aborted();
        state = State.Inactive;
        if (!seller.send(this.balance))
            throw;
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        inState(State.Created)
        require(msg.value == 2 * value)
        payable
    {
        purchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        onlyBuyer
        inState(State.Locked)
    {
        itemReceived();
```

```
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;
        // This actually allows both the buyer and the seller to
        // block the refund.
        if (!buyer.send(value) || !seller.send(this.balance))
            throw;
    }
}
```

6.3.4 Micropayment Channel

To be written.

6.4 Solidity in Depth

This section should provide you with all you need to know about Solidity. If something is missing here, please contact us on Gitter or make a pull request on Github.

6.4.1 Layout of a Solidity Source File

Source files can contain an arbitrary number of contract definitions, include directives and pragma directives.

Version Pragma

Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible changes. We try to keep such changes to an absolute minimum and especially introduce changes in a way that changes in semantics will also require changes in the syntax, but this is of course not always possible. Because of that, it is always a good idea to read through the changelog at least for releases that contain breaking changes, those releases will always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows:

```
pragma solidity ^0.4.0;
```

Such a source file will not compile with a compiler earlier than version 0.4.0 and it will also not work on a compiler starting form version 0.5.0 (this second condition is added by using `^`). The idea behind this is that there will be no breaking changes until version `0.5.0`, so we can always be sure that our code will compile the way we intended it to. We do not fix the exact version of the compiler, so that bugfix releases are still possible.

It is possible to specify much more complex rules for the compiler version, the expression follows those used by npm.

Importing other Source Files

Syntax and Semantics

Solidity supports import statements that are very similar to those available in JavaScript (from ES6 on), although Solidity does not know the concept of a "default export".

At a global level, you can use import statements of the following form:

```
import "filename";
```

This statement imports all global symbols from "filename" (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity).

```
import * as symbolName from "filename";
```

...creates a new global symbol `symbolName` whose members are all the global symbols from `"filename"`.

```
import {symbol1 as alias, symbol2} from "filename";
```

...creates new global symbols `alias` and `symbol2` which reference `symbol1` and `symbol2` from `"filename"`, respectively.

Another syntax is not part of ES6, but probably convenient:

```
import "filename" as symbolName;
```

which is equivalent to `import * as symbolName from "filename";`.

Paths

In the above, `filename` is always treated as a path with `/` as directory separator, `.` as the current and `..` as the parent directory. When `.` or `..` is followed by a character except `/`, it is not considered as the current or the parent directory. All path names are treated as absolute paths unless they start with the current `.` or the parent directory `...`

To import a file `x` from the same directory as the current file, use `import "./x" as x;`. If you use `import "x" as x;` instead, a different file could be referenced (in a global "include directory").

It depends on the compiler (see below) how to actually resolve the paths. In general, the directory hierarchy does not need to strictly map onto your local filesystem, it can also map to resources discovered via e.g. ipfs, http or git.

Use in Actual Compilers

When the compiler is invoked, it is not only possible to specify how to discover the first element of a path, but it is possible to specify path prefix remappings so that e.g. `github.com/ethereum/dapp-bin/library` is remapped to `/usr/local/dapp-bin/library` and the compiler will read the files from there. If multiple remappings can be applied, the one with the longest key is tried first. This allows for a "fallback-remapping" with e.g. `""` maps to `"/usr/local/include/solidity"`. Furthermore, these remappings can depend on the context, which allows you to configure packages to import e.g. different versions of a library of the same name.

solc:

For solc (the commandline compiler), these remappings are provided as `context:prefix=target` arguments, where both the `context:` and the `=target` parts are optional (where target defaults to prefix in that case). All remapping values that are regular files are compiled (including their dependencies). This mechanism is completely backwards-compatible (as long as no filename contains = or :) and thus not a breaking change. All imports in files in or below the directory `context` that import a file that starts with `prefix` are redirected by replacing `prefix` by `target`.

So as an example, if you clone `github.com/ethereum/dapp-bin/` locally to `/usr/local/dapp-bin`, you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

and then run the compiler as

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

As a more complex example, suppose you rely on some module that uses a very old version of dapp-bin. That old version of dapp-bin is checked out at `/usr/local/dapp-bin_old`, then you can use

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
     source.sol
```

so that all imports in `module2` point to the old version but imports in `module1` get the new version.

Note that solc only allows you to include files from certain directories: They have to be in the directory (or subdirectory) of one of the explicitly specified source files or in the directory (or subdirectory) of a remapping target. If you want to allow direct absolute includes, just add the remapping `=/`.

If there are multiple remappings that lead to a valid file, the remapping with the longest common prefix is chosen.

browser-solidity:

The browser-based compiler provides an automatic remapping for github and will also automatically retrieve the file over the network: You can import the iterable mapping by e.g. `import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;`.

Other source code providers may be added in the future.

Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Additionally, there is another type of comment called a natspec comment, for which the documentation is not yet written. They are written with a triple slash (`///`) or a double asterisk block(`/** ... */`) and they should be used directly above function declarations or statements. You can use Doxygen-style tags inside these comments to document functions, annotate conditions for formal verification, and provide a **confirmation text** which is shown to users when they attempt to invoke a function.

In the following example we document the title of the contract, the explanation for the two input parameters and two returned values.

```
   pragma solidity ^0.4.0;

/** @title Shape calculator.*/
contract shapeCalculator{
    /**@dev Calculates a rectangle's surface and perimeter.
      * @param w Width of the rectangle.
      * @param h Height of the rectangle.
      * @return s The calculated surface.
      * @return p The calculated perimeter.
      */
    function rectangle(uint w, uint h) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

6.4.2 Structure of a Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of *State Variables*, *Functions*, *Function Modifiers*, *Events*, *Structs Types* and *Enum Types*. Furthermore, contracts can inherit from other contracts.

State Variables

State variables are values which are permanently stored in contract storage.

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

See the *Types* section for valid state variable types and *Visibility and Accessors* for possible choices for visibility.

Functions

Functions are the executable units of code within a contract.

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    function bid() payable { // Function
        // ...
    }
}
```

Function Calls can happen internally or externally and have different levels of visibility (*Visibility and Accessors*) towards other contracts.

Function Modifiers

Function modifiers can be used to amend the semantics of functions in a declarative way (see *Function Modifiers* in contracts section).

```
pragma solidity ^0.4.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        if (msg.sender != seller) throw;
        _;
    }

    function abort() onlySeller { // Modifier usage
        // ...
```

```
    }
}
```

Events

Events are convenience interfaces with the EVM logging facilities.

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() payable {
        // ...
        HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}
```

See *Events* in contracts section for information on how events are declared and can be used from within a dapp.

Structs Types

Structs are custom defined types that can group several variables (see *Structs* in types section).

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

Enum Types

Enums can be used to create custom types with a finite set of values (see *Enums* in types section).

```
pragma solidity ^0.4.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

6.4.3 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified (or at least known - see *Type Deduction* below) at compile-time. Solidity provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see *Order of Precedence of Operators*.

Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- `!` (logical negation)
- `&&` (logical conjunction, "and")
- `||` (logical disjunction, "or")
- == (equality)
- `!=` (inequality)

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int` / `uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators:

- Comparisons: <=, <, ==, `!=`, >=, > (evaluate to `bool`)
- Bit operators: `&`, `|`, ^ (bitwise exclusive or), ~ (bitwise negation)
- Arithmetic operators: +, −, unary −, unary +, `*`, `/`, `%` (remainder), `**` (exponentiation), << (left shift), >> (right shift)

Division always truncates (it just maps to the DIV opcode of the EVM), but it does not truncate if both operators are *literals* (or literal expressions).

Division by zero and modulus with zero throws a runtime exception.

The result of a shift operation is the type of the left operand. The expression `x << y` is equivalent to `x * 2**y` and `x >> y` is equivalent to `x / 2**y`. This means that shifting negative numbers sign extends. Shifting by a negative amount throws a runtime exception.

Address

`address`: Holds a 20 byte value (size of an Ethereum address). Address types also have members and serve as base for all contracts.

Operators:

- <=, <, ==, `!=`, >= and >

Members of Addresses

- `balance` and `send`

For a quick reference, see *Address Related*.

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to an address using the `send` function:

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.send(10);
```

Note: If `x` is a contract address, its code (more specifically: its fallback function, if present) will be executed together with the `send` call (this is a limitation of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted. In this case, `send` returns `false`.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send` or even better: Use a pattern where the recipient withdraws the money.

- `call`, `callcode` and `delegatecall`

Furthermore, to interface with contracts that do not adhere to the ABI, the function `call` is provided which takes an arbitrary number of arguments of any type. These arguments are padded to 32 bytes and concatenated. One exception is the case where the first argument is encoded to exactly four bytes. In this case, it is not padded to allow the use of function signatures here.

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

`call` returns a boolean indicating whether the invoked function terminated (`true`) or caused an EVM exception (`false`). It is not possible to access the actual data returned (for this we would need to know the encoding and size in advance).

In a similar way, the function `delegatecall` can be used: The difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used. Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values.

All three functions `call`, `delegatecall` and `callcode` are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

Note: All contracts inherit the members of address, so it is possible to query the balance of the current contract using `this.balance`.

Warning: All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns.

Fixed-size byte arrays

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32`. `byte` is an alias for `bytes1`.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: `&`, `|`, ^ (bitwise exclusive or), ~ (bitwise negation), << (left shift), >> (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

The shifting operator works with any integer type as right operand (but will return the type of the left operand), which denotes the number of bits to shift by. Shifting by a negative amount will cause a runtime exception.

Members:

- `.length` yields the fixed length of the byte array (read-only).

Dynamically-sized byte array

`bytes`: Dynamically-sized byte array, see *Arrays*. Not a value-type!

`string`: Dynamically-sized UTF-8-encoded string, see *Arrays*. Not a value-type!

As a rule of thumb, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of `bytes1` to `bytes32` because they are much cheaper.

Fixed Point Numbers

COMING SOON...

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of `address` type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce a warning and are treated as regular rational number literals.

Rational and Integer Literals

Integer literals are formed from a sequence of numbers in the range 0-9. They are interpreted as decimals. For example, `69` means sixty nine. Octal literals do not exist in Solidity and leading zeros are ignored. For example, `0100` means one hundred.

Decimal literals are formed by a `.` with at least one number on one side. Examples include `1.`, `.1` and `1.3`.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with a non-literal expression). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, `(2**800 + 1) - 2**800` results in the constant `1` (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, `.5 * 8` results in the integer `4` (although non-integers were used in between).

If the result is not an integer, an appropriate `ufixed` or `fixed` type is used whose number of fractional bits is as large as required (approximating the rational number in the worst case).

In `var x = 1/4;`, `x` will receive the type `ufixed0x8` while in `var x = 1/3` it will receive the type `ufixed0x256` because `1/3` is not finitely representable in binary and will thus be approximated.

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Note: Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions *1 + 2* and *2 + 1* both belong to the same number literal type for the rational number three.

Note: Most finite decimal fractions like `5.3743` are not finitely representable in binary. The correct type for `5.3743` is `ufixed8x248` because that allows to best approximate the number. If you want to use the number together with types like `ufixed` (i.e. `ufixed128x128`), you have to explicitly specify the desired precision: `x + ufixed(5.3743)`.

Warning: Division on integer literals used to truncate in earlier versions, but it will now convert into a rational number, i.e. `5 / 2` is not equal to `2`, but to `2.5`.

Note: Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Even though we know that the value of the expression assigned to `b` in the following example evaluates to an integer, it still uses fixed point types (and not rational number literals) in between and so the code does not compile

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

String Literals

String literals are written with either double or single-quotes (`"foo"` or `'bar'`). They do not imply trailing zeroes as in C; *"foo"*‘ represents three bytes not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

String literals support escape characters, such as `\n`, `\xNN` and `\uNNNN`. `\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

Hexadecimal Literals

Hexademical Literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`). Their content must be a hexadecimal string and their value will be the binary representation of those values.

Hexademical Literals behave like String Literals and have the same convertibility restrictions.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversions check the value ranges at runtime and a failure causes an exception. Enums needs at least one member.

```
pragma solidity ^0.4.0;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is just
    // large enough to hold all enum values, i.e. if you have more values,
    // `uint16` will be used and so on.
    function getChoice() returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() returns (uint) {
        return uint(defaultChoice);
    }
}
```

Function Types

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions:

Internal functions can only be used inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:

```
function (<parameter types>) {internal|external} [constant] [payable] [returns (<return types>)]
```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted.

There are two ways to access a function in the current contract: Either directly by its name, `f`, or using `this.f`. The former will result in an internal function, the latter in an external function.

If a function type variable is not initialized, calling it will result in an exception. The same happens if you call a function after using `delete` on it.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

Example that shows how to use internal function types:

```
pragma solidity ^0.4.5;

library ArrayUtils {
  // internal functions can be used in internal library functions because
  // they will be part of the same code context
  function map(uint[] memory self, function (uint) returns (uint) f)
    internal
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint x, uint y) returns (uint) f
  )
    internal
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal returns (uint[] memory r) {
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
  function pyramid(uint l) returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal returns (uint) {
    return x + y;
  }
}
```

Another example that uses external function types:

```
pragma solidity ^0.4.5;

contract Oracle {
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
  function query(bytes data, function(bytes memory) external callback) {
    requests.push(Request(data, callback));
    NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) {
    // Here goes the check that the reply comes from a trusted source
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // known contract
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) {
    if (msg.sender != address(oracle)) throw;
    // Use the data
  }
}
```

Note that lambda or inline functions are planned but not yet supported.

Reference Types

Complex types, i.e. types which do not always fit into 256 bits have to be handled more carefully than the value-types we have already seen. Since copying them can be quite expensive, we have to think about whether we want them to be stored in **memory** (which is not persisting) or **storage** (where the state variables are held).

Data location

Every complex type, i.e. *arrays* and *structs*, has an additional annotation, the "data location", about whether it is stored in memory or in storage. Depending on the context, there is always a default, but it can be overridden by appending either `storage` or `memory` to the type. The default for function parameters (including return parameters) is `memory`, the default for local variables is `storage` and the location is forced to `storage` for state variables (obviously).

There is also a third data location, "calldata", which is a non-modifyable non-persistent area where function arguments are stored. Function parameters (not return parameters) of external functions are forced to "calldata" and it behaves mostly like memory.

Data locations are important because they change how assignments behave: Assignments between storage and memory and also to a state variable (even from other state variables) always create an independent copy. Assignments to local storage variables only assign a reference though, and this reference always points to the state variable even if the latter is changed in the meantime. On the other hand, assignments from a memory stored reference type to another memory-stored reference type does not create a copy.

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; // the data location of x is storage

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) {}
}
```

Summary

Forced data location:

- parameters (not return) of external functions: calldata
- state variables: storage

Default data location:

- parameters (also return) of functions: memory
- all other local variables: storage

Arrays

Arrays can have a compile-time fixed size or they can be dynamic. For storage arrays, the element type can be arbitrary (i.e. also other arrays, mappings or structs). For memory arrays, it cannot be a mapping and has to be an ABI type if it is an argument of a publicly-visible function.

An array of fixed size `k` and element type `T` is written as `T[k]`, an array of dynamic size as `T[]`. As an example, an array of 5 dynamic arrays of `uint` is `uint[][5]` (note that the notation is reversed when compared to some other languages). To access the second uint in the third dynamic array, you use `x[2][1]` (indices are zero-based and access works in the opposite way of the declaration, i.e. `x[2]` shaves off one level in the type from the right).

Variables of type `bytes` and `string` are special arrays. A `bytes` is similar to `byte[]`, but it is packed tightly in calldata. `string` is equal to `bytes` but does not allow length or index access (for now).

So `bytes` should always be preferred over `byte[]` because it is cheaper.

Note: If you want to access the byte-representation of a string `s`, use `bytes(s).length` / `bytes(s)[7] = 'x';`. Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual

characters!

It is possible to mark arrays `public` and have Solidity create an accessor. The numeric index will become a required parameter for the accessor.

Allocating Memory Arrays Creating arrays with variable length in memory can be done using the `new` keyword. As opposed to storage arrays, it is **not** possible to resize memory arrays by assigning to the `.length` member.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint len) {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // Here we have a.length == 7 and b.length == len
        a[6] = 8;
    }
}
```

Array Literals / Inline Arrays Array literals are arrays that are written as an expression and are not assigned to a variable right away.

```
pragma solidity ^0.4.0;

contract C {
    function f() {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) {
        // ...
    }
}
```

The type of an array literal is a memory array of fixed size whose base type is the common type of the given elements. The type of `[1, 2, 3]` is `uint8[3] memory`, because the type of each of these constants is `uint8`. Because of that, it was necessary to convert the first element in the example above to `uint`. Note that currently, fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
pragma solidity ^0.4.0;

contract C {
    function f() {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] x = [uint(1), 3, 4];
}
```

It is planned to remove this restriction in the future but currently creates some complications because of how arrays are passed in the ABI.

Members

length: Arrays have a `length` member to hold their number of elements. Dynamic arrays can be resized in storage (not in memory) by changing the `.length` member. This does not happen automatically when attempting to access elements outside the current length. The size of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push: Dynamic storage arrays and `bytes` (not `string`) have a member function called `push` that can be used to append an element at the end of the array. The function returns the new length.

Warning: It is not yet possible to use arrays of arrays in external functions.

Warning: Due to limitations of the EVM, it is not possible to return dynamic content from external function calls. The function `f` in `contract C { function f() returns (uint[]) { ... } }` will return something if called from web3.js, but not if called from Solidity.
The only workaround for now is to use large statically-sized arrays.

```
pragma solidity ^0.4.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of arrays but an array of pairs.
    bool[2][] m_pairsOfFlags;
    // newPairs is stored in memory - the default for function arguments

    function setAllFlagPairs(bool[2][] newPairs) {
        // assignment to a storage array replaces the complete array
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) {
        // access to a non-existing index will throw an exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) {
        // if the new size is smaller, removed array elements will be cleared
        m_pairsOfFlags.length = newSize;
    }

    function clear() {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes data) {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 8;
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) returns (uint) {
        return m_pairsOfFlags.push(flag);
    }
```

```
    function createMemoryArray(uint size) returns (bytes) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```
pragma solidity ^0.4.0;

contract CrowdFunding {
    // Defines a new type with two fields.
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // Creates new struct and saves in storage. We leave out the mapping type.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) payable {
        Campaign c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) returns (bool reached) {
        Campaign c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
```

```
        if (!c.beneficiary.send(amount))
            throw;
        return true;
    }
}
```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can itself contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable (of the default storage data location). This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

Mappings

Mapping types are declared as `mapping(_KeyType => _ValueType)`. Here `_KeyType` can be almost any type except for a mapping, a dynamically sized array, a contract, an enum and a struct. `_ValueType` can actually be any type, including mappings.

Mappings can be seen as hashtables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's *default value*. The similarity ends here, though: The key data is not actually stored in a mapping, only its `keccak256` hash used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being "set".

Mappings are only allowed for state variables (or as storage reference types in internal functions).

It is possible to mark mappings `public` and have Solidity create an accessor. The `_KeyType` will become a required parameter for the accessor and it will return `_ValueType`.

The `_ValueType` can be a mapping too. The accessor will have one parameter for each `_KeyType`, recursively.

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() returns (uint) {
        return MappingExample(<address>).balances(this);
    }
}
```

Note: Mappings are not iterable, but it is possible to implement a data structure on top of them. For an example, see iterable mapping.

Operators Involving LValues

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as shorthands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `a |=`, `&=` and `^=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` / `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements reset. For structs, it assigns a struct with all members reset.

`delete` has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted.

It is important to note that `delete a` really behaves like an assignment to `a`, i.e. it stores a new object in `a`.

```
pragma solidity ^0.4.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x which still holds a copy
        uint[] y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a complex object,
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local variables
        // referencing storage objects can only be made from existing storage objects.
    }
}
```

Conversions between Elementary Types

Implicit Conversions

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost: `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256` (because `uint256` cannot hold e.g. `-1`). Furthermore, unsigned integers can be converted to bytes of the same or larger size, but not vice-versa. Any type that can be converted to `uint160` can also be converted to `address`.

Explicit Conversions

If the compiler does not allow implicit conversion but you know what you are doing, an explicit type conversion is sometimes possible. Note that this may give you some unexpected behaviour so be sure to test to ensure that the result

is what you want! Take the following example where you are converting a negative `int8` to a `uint`:

```
int8 y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If a type is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

Type Deduction

For convenience, it is not always necessary to explicitly specify the type of a variable, the compiler automatically infers it from the type of the first expression that is assigned to the variable:

```
uint24 x = 0x123;
var y = x;
```

Here, the type of `y` will be `uint24`. Using `var` is not possible for function parameters or return parameters.

Warning: The type is only deduced from the first assignment, so the loop in the following snippet is infinite, as `i` will have the type `uint8` and any value of this type is smaller than `2000`. `for (var i = 0; i < 2000; i++) { ... }`

6.4.4 Units and Globally Available Variables

Ether Units

A literal number can take a suffix of `wei`, `finney`, `szabo` or `ether` to convert between the subdenominations of Ether, where Ether currency numbers without a postfix are assumed to be Wei, e.g. `2 ether == 2000 finney` evaluates to `true`.

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks = 7 days`
- `1 years = 365 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

These suffixes cannot be applied to variables. If you want to interpret some input variable in e.g. days, you can do it in the following way:

```
function f(uint start, uint daysAfter) {
    if (now >= start + daysAfter * 1 days) { ... }
}
```

Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain.

Block and Transaction Properties

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks excluding current
- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Note: The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

If you want to implement access restrictions in library functions using `msg.sender`, you have to manually supply the value of `msg.sender` as an argument.

Note: The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Mathematical and Cryptographic Functions

`addmod(uint x, uint y, uint k) returns (uint):` compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`.

mulmod(uint x, uint y, uint k) returns (uint): compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`.

keccak256(...) returns (bytes32): compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments

sha3(...) returns (bytes32): alias to *keccak256()*

sha256(...) returns (bytes32): compute the SHA-256 hash of the (tightly packed) arguments

ripemd160(...) returns (bytes20): compute RIPEMD-160 hash of the (tightly packed) arguments

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address): recover the address associated with the public key from elliptic curve signature or return zero on error

In the above, "tightly packed" means that the arguments are concatenated without padding. This means that the following are all identical:

```
keccak256("ab", "c")
keccak256("abc")
keccak256(0x616263)
keccak256(6382179)
keccak256(97, 98, 99)
```

If padding is needed, explicit type conversions can be used: `keccak256("\x00\x12")` is the same as `keccak256(uint16(0x12))`.

Note that constants will be packed using the minimum number of bytes required to store them. This means that, for example, `keccak256(0) == keccak256(uint8(0))` and `keccak256(0x12345678) == keccak256(uint32(0x12345678))`.

It might be that you run into Out-of-Gas for `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*. The reason for this is that those are implemented as so-called precompiled contracts and these contracts only really exist after they received the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution runs into an Out-of-Gas error. A workaround for this problem is to first send e.g. 1 Wei to each of the contracts before you use them in your actual contracts. This is not an issue on the official or test net.

Address Related

<address>.balance(uint256): balance of the *Address* in Wei

<address>.send(uint256 amount) returns (bool): send given amount of Wei to *Address*, returns `false` on failure

For more information, see the section on *Address*.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send` or even better: Use a pattern where the recipient withdraws the money.

Contract Related

this (current contract's type): the current contract, explicitly convertible to *Address*

selfdestruct(address recipient): destroy the current contract, sending its funds to the given *Address*

Furthermore, all functions of the current contract are callable directly including the current function.

6.4.5 Expressions and Control Structures

Input Parameters and Output Parameters

As in Javascript, functions may take parameters as input; unlike in Javascript and C, they may also return arbitrary number of parameters as output.

Input Parameters

The input parameters are declared the same way as variables are. As an exception, unused parameters can omit the variable name. For example, suppose we want our contract to accept one kind of external calls with two integers, we would write something like:

```
contract Simple {
    function taker(uint _a, uint _b) {
        // do something with _a and _b.
    }
}
```

Output Parameters

The output parameters can be declared with the same syntax after the `returns` keyword. For example, suppose we wished to return two results: the sum and the product of the two given integers, then we would write:

```
contract Simple {
    function arithmetics(uint _a, uint _b) returns (uint o_sum, uint o_product) {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

The names of output parameters can be omitted. The output values can also be specified using `return` statements. The `return` statements are also capable of returning multiple values, see *Returning Multiple Values*. Return parameters are initialized to zero; if they are not explicitly set, they stay to be zero.

Input parameters and output parameters can be used as expressions in the function body. There, they are also usable in the left-hand side of assignment.

Control Structures

Most of the control structures from JavaScript are available in Solidity except for `switch` and `goto`. So there is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, `? :`, with the usual semantics known from C or JavaScript.

Parentheses can *not* be omitted for conditionals, but curly brances can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

Returning Multiple Values

When a function has multiple output parameters, `return (v0, v1, ..., vn)` can return multiple values. The number of components must be the same as the number of output parameters.

Function Calls

Internal Function Calls

Functions of the current contract can be called directly ("internally"), also recursively, as seen in this nonsensical example:

```
contract C {
    function g(uint a) returns (uint ret) { return f(); }
    function f() returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract can be called internally.

External Function Calls

The expressions `this.g(8);` and `c.g(2);` (where `c` is a contract instance) are also valid function calls, but this time, the function will be called "externally", via a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

When calling functions of other contracts, the amount of Wei sent with the call and the gas can be specified:

```
contract InfoFeed {
    function info() payable returns (uint ret) { return 42; }
}


contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) { feed = InfoFeed(addr); }
    function callFeed() { feed.info.value(10).gas(800)(); }
}
```

The modifier `payable` has to be used for `info`, because otherwise, we would not be able to send Ether to it in the call `feed.info.value(10).gas(800)()`.

Note that the expression `InfoFeed(addr)` performs an explicit type conversion stating that "we know that the type of the contract at the given address is `InfoFeed`" and this does not execute a constructor. Explicit type conversions have to be handled with extreme caution. Never call a function on a contract where you are not sure about its type.

We could also have used `function setFeed(InfoFeed _feed) { feed = _feed; }` directly. Be careful about the fact that `feed.info.value(10).gas(800)` only (locally) sets the value and amount of gas sent with the function call and only the parentheses at the end perform the actual call.

Function calls cause exceptions if the called contract does not exist (in the sense that the account does not contain code) or if the called contract itself throws an exception or goes out of gas.

Warning: Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Named Calls and Anonymous Function Parameters

Function call arguments can also be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint key, uint value) { ... }

    function g() {
        // named arguments
        f({value: 2, key: 3});
    }
}
```

Omitted Function Parameter Names

The names of unused parameters (especially return parameters) can be omitted. Those names will still be present on the stack, but they are inaccessible.

```
pragma solidity ^0.4.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) returns(uint) {
        return k;
    }
}
```

Creating Contracts via `new`

A contract can create a new contract using the `new` keyword. The full code of the contract being created has to be known in advance, so recursive creation-dependencies are not possible.

```
pragma solidity ^0.4.0;

contract D {
    uint x;
    function D(uint a) payable {
        x = a;
    }
```

```
}


contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) {
        D newD = new D(arg);
    }

    function createAndEndowD(uint arg, uint amount) {
        // Send ether along with the creation
        D newD = (new D).value(amount)(arg);
    }
}
```

As seen in the example, it is possible to forward Ether to the creation, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done. See *Order of Precedence of Operators* for more information.

Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose size is a constant at compile-time. Those tuples can be used to return multiple values at the same time and also assign them to multiple variables (or LValues in general) at the same time:

```
contract C {
    uint[] data;

    function f() returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() {
        // Declares and assigns the variables. Specifying the type explicitly is not possible.
        var (x, b, y) = f();
        // Assigns to a pre-existing variable.
        (x, y) = (2, 7);
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        // If the tuple ends in an empty component,
        // the rest of the values are discarded.
        (data.length,) = f(); // Sets the length to 7
        // The same can be done on the left side.
        (,data[3]) = f(); // Sets data[3] to 2
        // Components can only be left out at the left-hand-side of assignments, with
```

```
        // one exception:
        (x,) = (1,);
        // (1,) is the only way to specify a 1-component tuple, because (1) is
        // equivalent to 1.
    }
}
```

Complications for Arrays and Structs

The semantics of assignment are a bit more complicated for non-value types like arrays and structs. Assigning *to* a state variable always creates an independent copy. On the other hand, assigning to a local variable creates an independent copy only for elementary types, i.e. static types that fit into 32 bytes. If structs or arrays (including `bytes` and `string`) are assigned from a state variable to a local variable, the local variable holds a reference to the original state variable. A second assignment to the local variable does not modify the state but only changes the reference. Assignments to members (or elements) of the local variable *do* change the state.

Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is. For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. Finally, for dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or string.

A variable declared anywhere within a function will be in scope for the *entire function*, regardless of where it is declared. This happens because Solidity inherits its scoping rules from JavaScript. This is in contrast to many languages where variables are only scoped where they are declared until the end of the semantic block. As a result, the following code is illegal and cause the compiler to throw an error, `Identifier already declared`:

```
pragma solidity ^0.4.0;

contract ScopingErrors {
    function scoping() {
        uint i = 0;

        while (i++ < 1) {
            uint same1 = 0;
        }

        while (i++ < 2) {
            uint same1 = 0;// Illegal, second declaration of same1
        }
    }

    function minimalScoping() {
        {
            uint same2 = 0;
        }

        {
            uint same2 = 0;// Illegal, second declaration of same2
        }
    }

    function forLoopScoping() {
```

```
        for (uint same3 = 0; same3 < 1; same3++) {
        }

        for (uint same3 = 0; same3 < 1; same3++) {// Illegal, second declaration of same3
        }
    }
}
```

In addition to this, if a variable is declared, it will be initialized at the beginning of the function to its default value. As a result, the following code is legal, despite being poorly written:

```
function foo() returns (uint) {
    // baz is implicitly initialized as 0
    uint bar = 5;
    if (true) {
        bar += baz;
    } else {
        uint baz = 10;// never executes
    }
    return bar;// returns 5
}
```

Exceptions

There are some cases where exceptions are thrown automatically (see below). You can use the `throw` instruction to throw an exception manually. The effect of an exception is that the currently executing call is stopped and reverted (i.e. all changes to the state and balances are undone) and the exception is also "bubbled up" through Solidity function calls (exceptions are `send` and the low-level functions `call`, `delegatecall` and `callcode`, those return `false` in case of an exception).

Catching exceptions is not yet possible.

In the following example, we show how `throw` can be used to easily revert an Ether transfer and also how to check the return value of `send`:

```
pragma solidity ^0.4.0;

contract Sharer {
    function sendHalf(address addr) payable returns (uint balance) {
        if (!addr.send(msg.value / 2))
            throw; // also reverts the transfer to Sharer
        return this.balance;
    }
}
```

Currently, Solidity automatically generates a runtime exception in the following situations:

1. If you access an array at a too large or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
2. If you access a fixed-length `bytesN` at a too large or negative index.
3. If you call a function via a message call but it does not finish properly (i.e. it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall` or `callcode` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
4. If you create a contract using the `new` keyword but the contract creation does not finish properly (see above for the definition of "not finish properly").
5. If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).

6. If you shift by a negative amount.
7. If you convert a value too big or negative into an enum type.
8. If you perform an external function call targeting a contract that contains no code.
9. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
10. If your contract receives Ether via a public accessor function.
11. If you call a zero-initialized variable of internal function type.

Internally, Solidity performs an "invalid jump" when a user-provided exception is thrown. In contrast, it performs an invalid operation (instruction `0xfe`) if a runtime exception is encountered. In both cases, this causes the EVM to revert all changes made to the state. The reason for this is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to retain the atomicity of transactions, the safest thing to do is to revert all changes and make the whole transaction (or at least call) without effect.

Inline Assembly

For more fine-grained control especially in order to enhance the language by writing libraries, it is possible to interleave Solidity statements with inline assembly in a language close to the one of the virtual machine. Due to the fact that the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity's inline assembly tries to facilitate that and other issues arising when writing manual assembly by the following features:

- functional-style opcodes: `mul(1, add(2, 3))` instead of `push1 3 push1 2 add push1 1 mul`
- assembly-local variables: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- access to external variables: `function f(uint x) { assembly { x := sub(x, 1) } }`
- labels: `let x := 10 repeat:  x := sub(x, 1) jumpi(repeat, eq(x, 0))`

We now want to describe the inline assembly language in detail.

Warning: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This discards several important safety features of Solidity.

Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is not possible at all with "plain Solidity" and the idea is that assembly libraries will be used to enhance the language in such ways.

```
pragma solidity ^0.4.0;

library GetCode {
    function at(address _addr) returns (bytes o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
```

```
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

Inline assembly could also be beneficial in cases where the optimizer fails to produce efficient code. Please be aware that assembly is much more difficult to write because the compiler does not perform checks, so you should use it for complex things only if you really know what you are doing.

```
pragma solidity ^0.4.0;

library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := mload(add(add(_data, 0x20), i))
            }
        }
    }
}
```

Syntax

Inline assembly parses comments, literals and identifiers exactly as Solidity, so you can use the usual `//` and `/* */` comments. Inline assembly is initiated by `assembly { ... }` and inside these curly braces, the following can be used (see the later sections for more details)

- literals, e.g. `0x123`, `42` or `"abc"` (strings up to 32 characters)
- opcodes (in "instruction style"), e.g. `mload sload dup1 sstore`, for a list see below
- opcodes in functional style, e.g. `add(1, mload(0))`
- labels, e.g. `name:`
- variable declarations, e.g. `let x := 7` or `let x := add(y, 3)`
- identifiers (externals, labels or assembly-local variables), e.g. `jump(name)`, `3 x add`
- assignments (in "instruction style"), e.g. `3 =: x`
- assignments in functional style, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`

Opcodes

This document does not want to be a full description of the Ethereum virtual machine, but the following list can be used as a reference of its opcodes.

If an opcode takes arguments (always from the top of the stack), they are given in parentheses. Note that the order of arguments can be seen as being reversed compared to the instructional style (explained below). Opcodes marked with - do not push an item onto the stack, those marked with * are special and all others push exactly one item onte the stack.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to (excluding) position `b` and `storage[p]` signifies the storage contents at position `p`.

The opcodes `pushi` and `jumpdest` cannot be used directly.

stop	-	stop execution, identical to return(0,0)
add(x, y)		x + y
sub(x, y)		x - y
mul(x, y)		x * y
div(x, y)		x / y
sdiv(x, y)		x / y, for signed numbers in two's complement
mod(x, y)		x % y
smod(x, y)		x % y, for signed numbers in two's complement
exp(x, y)		x to the power of y
not(x)		~x, every bit of x is negated
lt(x, y)		1 if x < y, 0 otherwise
gt(x, y)		1 if x > y, 0 otherwise
slt(x, y)		1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)		1 if x == y, 0 otherwise
iszero(x)		1 if x == 0, 0 otherwise
and(x, y)		bitwise and of x and y
or(x, y)		bitwise or of x and y
xor(x, y)		bitwise xor of x and y
byte(n, x)		nth byte of x, where the most significant byte is the 0th byte
addmod(x, y, m)		(x + y) % m with arbitrary precision arithmetics
mulmod(x, y, m)		(x * y) % m with arbitrary precision arithmetics
signextend(i, x)		sign extend from (i*8+7)th bit counting from least significant
sha3(p, n)		keccak(mem[p...(p+n)))
jump(label)	-	jump to label / code position
jumpi(label, cond)	-	jump to label if cond is nonzero
pc		current position in code
pop(x)	-	remove the element pushed by x
dup1 ... dup16		copy ith stack slot to the top (counting from top)
swap1 ... swap16	*	swap topmost and ith stack slot below it
mload(p)		mem[p..(p+32))
mstore(p, v)	-	mem[p..(p+32)) := v
mstore8(p, v)	-	mem[p] := v & 0xff - only modifies a single byte
sload(p)		storage[p]
sstore(p, v)	-	storage[p] := v
msize		size of memory, i.e. largest accessed memory index

Table 6.1 – continued from previous pag

gas		gas still available to execution
address		address of the current contract / execution context
balance(a)		wei balance at address a
caller		call sender (excluding delegatecall)
callvalue		wei sent together with the current call
calldataload(p)		calldata starting from position p (32 bytes)
calldatasize		size of calldata in bytes
calldatacopy(t, f, s)	-	copy s bytes from calldata at position f to mem at position t
codesize		size of the code of the current contract / execution context
codecopy(t, f, s)	-	copy s bytes from code at position f to mem at position t
extcodesize(a)		size of the code at address a
extcodecopy(a, t, f, s)	-	like codecopy(t, f, s) but take code at address a
create(v, p, s)		create new contract with code mem[p..(p+s)) and send v wei and return the new addre
call(g, a, v, in, insize, out, outsize)		call contract at address a with input mem[in..(in+insize)) providing g gas and v wei a
callcode(g, a, v, in, insize, out, outsize)		identical to *call* but only use the code from a and stay in the context of the current co
delegatecall(g, a, in, insize, out, outsize)		identical to *callcode* but also keep `caller` and `callvalue`
return(p, s)	-	end execution, return data mem[p..(p+s))
selfdestruct(a)	-	end execution, destroy current contract and send funds to a
invalid	-	end execution with invalid instruction
log0(p, s)	-	log without topics and data mem[p..(p+s))
log1(p, s, t1)	-	log with topic t1 and data mem[p..(p+s))
log2(p, s, t1, t2)	-	log with topics t1, t2 and data mem[p..(p+s))
log3(p, s, t1, t2, t3)	-	log with topics t1, t2, t3 and data mem[p..(p+s))
log4(p, s, t1, t2, t3, t4)	-	log with topics t1, t2, t3, t4 and data mem[p..(p+s))
origin		transaction sender
gasprice		gas price of the transaction
blockhash(b)		hash of block nr b - only for last 256 blocks excluding current
coinbase		current mining beneficiary
timestamp		timestamp of the current block in seconds since the epoch
number		current block number
difficulty		difficulty of the current block
gaslimit		block gas limit of the current block

Literals

You can use integer constants by typing them in decimal or hexadecimal notation and an appropriate `PUSHi` instruction will automatically be generated. The following creates code to add 2 and 3 resulting in 5 and then computes the bitwise and with the string "abc". Strings are stored left-aligned and cannot be longer than 32 bytes.

```
assembly { 2 3 add "abc" and }
```

Functional Style

You can type opcode after opcode in the same way they will end up in bytecode. For example adding `3` to the contents in memory at position `0x80` would be

```
3 0x80 mload add 0x80 mstore
```

As it is often hard to see what the actual arguments for certain opcodes are, Solidity inline assembly also provides a "functional style" notation where the same code would be written as follows

```
mstore(0x80, add(mload(0x80), 3))
```

Functional style and instructional style can be mixed, but any opcode inside a functional style expression has to return exactly one stack slot (most of the opcodes do).

Note that the order of arguments is reversed in functional-style as opposed to the instruction-style way. If you use functional-style, the first argument will end up on the stack top.

Access to External Variables and Functions

Solidity variables and other identifiers can be accessed by simply using their name. For storage and memory variables, this will push the address and not the value onto the stack. Also note that non-struct and non-array storage variable addresses occupy two slots on the stack: One for the address and one for the byte offset inside the storage slot. In assignments (see below), we can even use local Solidity variables to assign to.

Functions external to inline assembly can also be accessed: The assembly will push their entry label (with virtual function resolution applied). The calling semantics in solidity are:

- the caller pushes return label, arg1, arg2, ..., argn
- the call returns with ret1, ret2, ..., retn

This feature is still a bit cumbersome to use, because the stack offset essentially changes during the call, and thus references to local variables will be wrong. It is planned that the stack height changes can be specified in inline assembly.

```
pragma solidity ^0.4.0;

contract C {
    uint b;
    function f(uint x) returns (uint r) {
        assembly {
            b pop // remove the offset, we know it is zero
            sload
            x
            mul
            =: r  // assign to return variable r
        }
    }
}
```

Labels

Another problem in EVM assembly is that `jump` and `jumpi` use absolute addresses which can change easily. Solidity inline assembly provides labels to make the use of jumps easier. The following code computes an element in the Fibonacci series.

```
{
    let n := calldataload(4)
    let a := 1
    let b := a
loop:
    jumpi(loopend, eq(n, 0))
    a add swap1
    n := sub(n, 1)
    jump(loop)
loopend:
```

```
    mstore(0, a)
    return(0, 0x20)
}
```

Please note that automatically accessing stack variables can only work if the assembler knows the current stack height. This fails to work if the jump source and target have different stack heights. It is still fine to use such jumps, you should just not access any stack variables (even assembly variables) in that case.

Furthermore, the stack height analyser goes through the code opcode by opcode (and not according to control flow), so in the following case, the assembler will have a wrong impression about the stack height at label `two`:

```
{
    jump(two)
    one:
        // Here the stack height is 1 (because we pushed 7),
        // but the assembler thinks it is 0 because it reads
        // from top to bottom.
        // Accessing stack variables here will lead to errors.
        jump(three)
    two:
        7 // push something onto the stack
        jump(one)
    three:
}
```

Note: `invalidJumpLabel` is a pre-defined label. Jumping to this location will always result in an invalid jump, effectively aborting execution of the code.

Declaring Assembly-Local Variables

You can use the `let` keyword to declare variables that are only visible in inline assembly and actually only in the current `{...}`-block. What happens is that the `let` instruction will create a new stack slot that is reserved for the variable and automatically removed again when the end of the block is reached. You need to provide an initial value for the variable which can be just `0`, but it can also be a complex functional-style expression.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint x) returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
            } // y is "deallocated" here
            b := add(b, v)
        } // v is "deallocated" here
    }
}
```

Assignments

Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

There are two kinds of assignments: Functional-style and instruction-style. For functional-style assignments (`variable := value`), you need to provide a value in a functional-style expression that results in exactly one stack value and for instruction-style (`=: variable`), the value is just taken from the stack top. For both ways, the colon points to the name of the variable.

```
assembly {
    let v := 0 // functional-style assignment as part of variable declaration
    let g := add(v, 2)
    sload(10)
    =: v // instruction style assignment, puts the result of sload(10) into v
}
```

Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. The only thing the assembler does for you is re-arranging functional-style opcodes, managing jump labels, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached. Especially for those two last cases, it is important to know that the assembler only counts stack height from top to bottom, not necessarily following control flow. Furthermore, operations like swap will only swap the contents of the stack but not the location of variables.

Conventions in Solidity

In contrast to EVM assembly, Solidity knows types which are narrower than 256 bits, e.g. `uint24`. In order to make them more efficient, most arithmetic operations just treat them as 256-bit numbers and the higher-order bits are only cleaned at the point where it is necessary, i.e. just shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher order bits first.

Solidity manages memory in a very simple way: There is a "free memory pointer" at position `0x40` in memory. If you want to allocate memory, just use the memory from that point on and update the pointer accordingly.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (yes, this is even true for `byte[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and then only the array elements follow.

Warning: Statically-sized memory arrays do not have a length field, but it will be added soon to allow better convertibility between statically- and dynamically-sized arrays, so please do not rely on that.

6.4.6 Contracts

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables are inaccessible.

Creating Contracts

Contracts can be created "from outside" or from Solidity contracts. When a contract is created, its constructor (a function with the same name as the contract) is executed once.

A constructor is optional. Only one constructor is allowed and this means overloading is not supported.

From `web3.js`, i.e. the JavaScript API, this is done as follows:

```
// Need to specify some source including contract name for the data param below
var source = "contract CONTRACT_NAME { function CONTRACT_NAME(unit a, uint b) {} }";

// The json abi array generated by the compiler
var abiArray = [
    {
        "inputs":[
            {"name":"x","type":"uint256"},
            {"name":"y","type":"uint256"}
        ],
        "type":"constructor"
    },
    {
        "constant":true,
        "inputs":[],
        "name":"x",
        "outputs":[{"name":"","type":"bytes32"}],
        "type":"function"
    }
];

var MyContract_ = web3.eth.contract(source);
MyContract = web3.eth.contract(MyContract_.CONTRACT_NAME.info.abiDefinition);
// deploy new contract
var contractInstance = MyContract.new(
    10,
    11,
    {from: myAccount, gas: 1000000}
);
```

Internally, constructor arguments are passed after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
pragma solidity ^0.4.0;

contract OwnedToken {
    // TokenCreator is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    function OwnedToken(bytes32 _name) {
        // State variables are accessed via their name
        // and not via e.g. this.owner. This also applies
```

```
        // to functions and especially in the constructors,
        // you can only call them like that ("internall"),
        // because the contract itself does not exist yet.
        owner = msg.sender;
        // We do an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is TokenCreator, there is
        // no real way to check that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) {
        // Only the creator can alter the name --
        // the comparison is possible since contracts
        // are implicitly convertible to addresses.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;
        // We also want to ask the creator if the transfer
        // is fine. Note that this calls a function of the
        // contract defined below. If the call fails (e.g.
        // due to out-of-gas), the execution here stops
        // immediately.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name)
       returns (OwnedToken tokenAddress)
    {
        // Create a new Token contract and return its address.
        // From the JavaScript side, the return type is simply
        // "address", as this is the closest type available in
        // the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) {
        // Again, the external type of "tokenAddress" is
        // simply "address".
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(
        address currentOwner,
        address newOwner
    ) returns (bool ok) {
        // Check some arbitrary condition.
        address tokenAddress = msg.sender;
        return (keccak256(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }
```

```
}
```

Visibility and Accessors

Since Solidity knows two kinds of function calls (internal ones that do not create an actual EVM call (also called a "message call") and external ones that do), there are four types of visibilities for functions and state variables.

Functions can be specified as being `external`, `public`, `internal` or `private`, where the default is `public`. For state variables, `external` is not possible and the default is `internal`.

`external`: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). External functions are sometimes more efficient when they receive large arrays of data.

`public`: Public functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic accessor function (see below) is generated.

`internal`: Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using `this`.

`private`: Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

Note: Everything that is inside a contract is visible to all external observers. Making something `private` only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, `D`, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract `E` is derived from `C` and, thus, can call `compute`.

```
pragma solidity ^0.4.0;

contract C {
    uint private data;

    function f(uint a) private returns(uint b) { return a + 1; }
    function setData(uint a) { data = a; }
    function getData() public returns(uint) { return data; }
    function compute(uint a, uint b) internal returns (uint) { return a+b; }
}


contract D {
    function readData() {
        C c = new C();
        uint local = c.f(7); // error: member "f" is not visible
```

```
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member "compute" is not visible
    }
}


contract E is C {
    function g() {
        C c = new C();
        uint val = compute(3, 5);  // acces to internal member (from derivated to parent contract)
    }
}
```

Accessor Functions

The compiler automatically creates accessor functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. The initialization of state variables can be done at declaration.

```
pragma solidity ^0.4.0;

contract C {
    uint public data = 42;
}


contract Caller {
    C c = new C();
    function f() {
        uint local = c.data();
    }
}
```

The accessor functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it is evaluated as a state variable and if it is accessed externally (i.e. with `this.`), it is evaluated as a function.

```
pragma solidity ^0.4.0;

contract C {
    uint public data;
    function x() {
        data = 3; // internal access
        uint val = this.data(); // external access
    }
}
```

The next example is a bit more complex:

```
pragma solidity ^0.4.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
```

```
    mapping (uint => mapping(bool => Data[])) public data;
}
```

It will generate a function of the following form:

```
function data(uint arg1, bool arg2, uint arg3) returns (uint a, bytes3 b) {
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
}
```

Note that the mapping in the struct is omitted because there is no good way to provide the key for the mapping.

Function Modifiers

Modifiers can be used to easily change the behaviour of functions, for example to automatically check a condition prior to executing the function. They are inheritable properties of contracts and may be overridden by derived contracts.

```
pragma solidity ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;

    // This contract only defines a modifier but does not use
    // it - it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // "_;" in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        if (msg.sender != owner)
            throw;
        _;
    }
}


contract mortal is owned {
    // This contract inherits the "onlyOwner"-modifier from
    // "owned" and applies it to the "close"-function, which
    // causes that calls to "close" only have an effect if
    // they are made by the stored owner.
    function close() onlyOwner {
        selfdestruct(owner);
    }
}


contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}
```

```
contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    function Register(uint initialPrice) { price = initialPrice; }

    // It is important to also provide the
    // "payable" keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        if (locked) throw;
        locked = true;
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within msg.sender.call cannot call f again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() noReentrancy returns (uint) {
        if (!msg.sender.call()) throw;
        return 7;
    }
}
```

Multiple modifiers can be applied to a function by specifying them in a whitespace-separated list and will be evaluated in order.

Warning: In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the "_" in the preceding modifier.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

Constant State Variables

State variables can be declared as constant (this is not yet implemented for array and struct types and not possible for mapping types).

```
pragma solidity ^0.4.0;
```

```
contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
}
```

This has the effect that the compiler does not reserve a storage slot for these variables and every occurrence is replaced by their constant value.

The value expression can only contain integer arithmetics.

Constant Functions

Functions can be declared constant. These functions promise not to modify the state.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint a, uint b) constant returns (uint) {
        return a * (b + 42);
    }
}
```

Note: Accessor methods are marked constant.

Warning: The compiler does not enforce yet that a constant method is not modifying state.

Fallback Function

A contract can have exactly one unnamed function. This function cannot have arguments and cannot return anything. It is executed on a call to the contract if none of the other functions matches the given function identifier (or if no data was supplied at all).

Furthermore, this function is executed whenever the contract receives plain Ether (without data). In such a context, there is usually very little gas available to the function call (to be precise, 2300 gas), so it is important to make fallback functions as cheap as possible.

In particular, the following operations will consume more gas than the stipend provided to a fallback function:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Please ensure you test your fallback function thoroughly to ensure the execution cost is less than 2300 gas before deploying a contract.

Warning: Contracts that receive Ether but do not define a fallback function throw an exception, sending back the Ether (this was different before Solidity v0.4.0). So if you want your contract to receive Ether, you have to implement a fallback function.

```
pragma solidity ^0.4.0;

contract Test {
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the "payable"
    // modifier.
    function() { x = 1; }
    uint x;
}


// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    function() payable { }
}


contract Caller {
    function callTest(Test test) {
        test.call(0xabcdef01); // hash does not exist
        // results in test.x becoming == 1.

        // The following call will fail, reject the
        // Ether and return false:
        test.send(2 ether);
    }
}
```

Events

Events allow the convenient usage of the EVM logging facilities, which in turn can be used to "call" JavaScript callbacks in the user interface of a dapp, which listen for these events.

Events are inheritable members of contracts. When they are called, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract and will be incorporated into the blockchain and stay there as long as a block is accessible (forever as of Frontier and Homestead, but this might change with Serenity). Log and event data is not accessible from within contracts (not even from the contract that created a log).

SPV proofs for logs are possible, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain (but be aware of the fact that ultimately, also the block headers have to be supplied because the contract can only see the last 256 block hashes).

Up to three parameters can receive the attribute `indexed` which will cause the respective arguments to be searched for: It is possible to filter for specific values of indexed arguments in the user interface.

If arrays (including `string` and `bytes`) are used as indexed arguments, the Keccak-256 hash of it is stored as topic instead.

The hash of the signature of the event is one of the topics except if you declared the event with `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name.

All non-indexed arguments will be stored in the data part of the log.

Note: Indexed arguments will not be stored themselves, you can only search for the values, but it is impossible to retrieve the values themselves.

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) {
        // Any call to this function (even deeply nested) can
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

The use in the JavaScript API would be as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at(0x123 /* address */);

var event = clientReceipt.Deposit();

// watch for changes
event.watch(function(error, result){
    // result will contain various information
    // including the argumets given to the Deposit
    // call.
    if (!error)
        console.log(result);
});

// Or pass a callback to start watching immediately
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

Low-Level Interface to Logs

It is also possible to access the low-level interface to the logging mechanism via the functions `log0`, `log1`, `log2`, `log3` and `log4`. `logi` takes `i + 1` parameter of type `bytes32`, where the first argument will be used for the data part of the log and the others as topics. The event call above can be performed in the same way as

```
log3(
    msg.value,
    0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20,
    msg.sender,
    _id
);
```

where the long hexadecimal number is equal to `keccak256("Deposit(address,hash256,uint256)")`, the signature of the event.

Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

Inheritance

Solidity supports multiple inheritance by copying code including polymorphism.

All function calls are virtual, which means that the most derived function is called, except when the contract name is explicitly given.

Even if a contract inherits from multiple other contracts, only a single contract is created on the blockchain, the code from the base contracts is always copied into the final contract.

The general inheritance system is very similar to Python's, especially concerning multiple inheritance.

Details are given in the following example.

```
pragma solidity ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}


// Use "is" to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
contract Config {
    function lookup(uint id) returns (address adr);
}


contract NameReg {
    function register(bytes32 name);
    function unregister();
 }
```

```
// Multiple inheritance is possible. Note that "owned" is
// also a base class of "mortal", yet there is only a single
// instance of "owned" (as for virtual inheritance in C++).
contract named is owned, mortal {
    function named(bytes32 name) {
        Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs.  If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    function kill() {
        if (msg.sender == owner) {
            Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
            mortal.kill();
        }
    }
}


// If a constructor takes an argument, it needs to be
// provided in the header (or modifier-invocation-style at
// the constructor of the derived contract (see below)).
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) {
      if (msg.sender == owner) info = newInfo;
   }

   function get() constant returns(uint r) { return info; }

   uint info;
}
```

Note that above, we call `mortal.kill()` to "forward" the destruction request. The way this is done is problematic, as seen in the following example:

```
pragma solidity ^0.4.0;

contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


contract Base1 is mortal {
    function kill() { /* do cleanup 1 */ mortal.kill(); }
}


contract Base2 is mortal {
    function kill() { /* do cleanup 2 */ mortal.kill(); }
```

```
}


contract Final is Base1, Base2 {
}
```

A call to `Final.kill()` will call `Base2.kill` as the most derived override, but this function will bypass `Base1.kill`, basically because it does not even know about `Base1`. The way around this is to use `super`:

```
pragma solidity ^0.4.0;

contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


contract Base1 is mortal {
    function kill() { /* do cleanup 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() { /* do cleanup 2 */ super.kill(); }
}


contract Final is Base2, Base1 {
}
```

If `Base1` calls a function of `super`, it does not simply call this function on one of its base contracts, it rather calls this function on the next base contract in the final inheritance graph, so it will call `Base2.kill()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base1, Base2, mortal, owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Arguments for Base Constructors

Derived contracts need to provide all arguments needed for the base constructors. This can be done at two places:

```
pragma solidity ^0.4.0;

contract Base {
    uint x;
    function Base(uint _x) { x = _x; }
}


contract Derived is Base(7) {
    function Derived(uint _y) Base(_y * _y) {
    }
}
```

Either directly in the inheritance list (`is Base(7)`) or in the way a modifier would be invoked as part of the header of the derived constructor (`Base(_y * _y)`). The first way to do it is more convenient if the constructor argument is a constant and defines the behaviour of the contract or describes it. The second way has to be used if the constructor

arguments of the base depend on those of the derived contract. If, as in this silly example, both places are used, the modifier-style argument takes precedence.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems, one of them being the Diamond Problem. Solidity follows the path of Python and uses "C3 Linearization" to force a specific order in the DAG of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important. In the following code, Solidity will give the error "Linearization of inheritance graph impossible".

```
pragma solidity ^0.4.0;

contract X {}
contract A is X {}
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

A simple rule to remember is to specify the base classes in the order from "most base-like" to "most derived".

Inheriting Different Kinds of Members of the Same Name

When the inheritance results in a contract with a function and a modifier of the same name, it is considered as an error. This error is produced also by an event and a modifier of the same name, and a function and an event of the same name. As an exception, a state variable accessor can override a public function.

Abstract Contracts

Contract functions can lack an implementation as in the following example (note that the function declaration header is terminated by `;`):

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() returns (bytes32);
}
```

Such contracts cannot be compiled (even if they contain implemented functions alongside non-implemented functions), but they can be used as base contracts:

```
pragma solidity ^0.4.0;

contract Cat is Feline {
    function utterance() returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it will itself be abstract.

Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the `DELEGATECALL` (`CALLCODE` until Homestead) feature of the EVM. This means that if

library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise).

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (`L.f()` if `L` is the name of the library). Furthermore, `internal` functions of libraries are visible in all contracts, just as if the library were a base contract. Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and memory types will be passed by reference and not copied. In order to realise this in the EVM, code of internal library functions (and all functions called from therein) will be pulled into the calling contract and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

The following example illustrates how to use libraries (but be sure to check out *using for* for a more advanced example to implement a set).

```
pragma solidity ^0.4.0;

library Set {
  // We define a new struct datatype that will be used to
  // hold its data in the calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage
  // reference" and thus only its storage address and not
  // its contents is passed as part of the call.  This is a
  // special feature of library functions.  It is idiomatic
  // to call the first parameter 'self', if the function can
  // be seen as a method of that object.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}


contract C {
    Set.Data knownValues;

    function register(uint value) {
```

```
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        if (!Set.insert(knownValues, value))
            throw;
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Of course, you do not have to follow this way to use libraries - they can also be used without defining struct data types, functions also work without any storage reference parameters, can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (`DELEGATECALL`) to an external contract/library. If you use libraries, take care that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of *CALLCODE*, `msg.sender` and `msg.value` changed, though).

The following example shows how to use memory types and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
pragma solidity ^0.4.0;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    function fromUint(uint x) internal returns (bigint r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint _a, bigint _b) internal returns (bigint r) {
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && carry > 0))
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }
```

```
    function max(uint a, uint b) private returns (uint) {
        return a > b ? a : b;
    }
}


contract C {
    using BigInt for BigInt.bigint;

    function f() {
        var x = BigInt.fromUint(7);
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

As the compiler cannot know where the library will be deployed at, these addresses have to be filled into the final bytecode by a linker (see *Using the Commandline Compiler* for how to use the commandline compiler for linking). If the addresses are not given as arguments to the compiler, the compiled hex code will contain placeholders of the form `__Set_______` (where `Set` is the name of the library). The address can be filled manually by replacing all those 40 symbols by the hex encoding of the address of the library contract.

Restrictions for libraries in comparison to contracts:

- No state variables
- Cannot inherit nor be inherited
- Cannot recieve Ether

(These might be lifted at a later point.)

Using For

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`). These functions will receive the object they are called on as their first parameter (like the `self` variable in Python).

The effect of `using A for *;` is that the functions from the library `A` are attached to any type.

In both situations, all functions, even those where the type of the first parameter does not match the type of the object, are attached. The type is checked at the point the function is called and function overload resolution is performed.

The `using A for B;` directive is active for the current scope, which is limited to a contract for now but will be lifted to the global scope later, so that by including a module, its data types including library functions are available without having to add further code.

Let us rewrite the set example from the *Libraries* in this way:

```
pragma solidity ^0.4.0;

// This is the same code as before, just without comments
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
```

```
        return true;
    }

    function remove(Data storage self, uint value)
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    using Set for Set.Data; // this is the crucial change
    Set.Data knownValues;

    function register(uint value) {
        // Here, all variables of type Set.Data have
        // corresponding member functions.
        // The following function call is identical to
        // Set.insert(knownValues, value)
        if (!knownValues.insert(value))
            throw;
    }
}
```

It is also possible to extend elementary types in that way:

```
pragma solidity ^0.4.0;

library Search {
    function indexOf(uint[] storage self, uint value) returns (uint) {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}


contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) {
        data.push(value);
    }

    function replace(uint _old, uint _new) {
        // This performs the library function call
        uint index = data.indexOf(_old);
        if (index == uint(-1))
```

```
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Note that all library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used.

6.4.7 Solidity Assembly

Solidity defines an assembly language that can also be used without Solidity. This assembly language can also be used as "inline assembly" inside Solidity source code. We start with describing how to use inline assembly and how it differs from standalone assembly and then specify assembly itself.

TODO: Write about how scoping rules of inline assembly are a bit different and the complications that arise when for example using internal functions of libraries. Furhermore, write about the symbols defined by the compiler.

Inline Assembly

For more fine-grained control especially in order to enhance the language by writing libraries, it is possible to interleave Solidity statements with inline assembly in a language close to the one of the virtual machine. Due to the fact that the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity's inline assembly tries to facilitate that and other issues arising when writing manual assembly by the following features:

- functional-style opcodes: `mul(1, add(2, 3))` instead of `push1 3 push1 2 add push1 1 mul`
- assembly-local variables: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- access to external variables: `function f(uint x) { assembly { x := sub(x, 1) } }`
- labels: `let x := 10 repeat: x := sub(x, 1) jumpi(repeat, eq(x, 0))`
- loops: `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- switch statements: `switch x case 0: { y := mul(x, 2) } default: { y := 0 }`
- function calls: `function f(x) -> (y) { switch x case 0: { y := 1 } default: { y := mul(x, f(sub(x, 1))) } }`

Note: Of the above, loops, function calls and switch statements are not yet implemented.

We now want to describe the inline assembly language in detail.

Warning: Inline assembly is still a relatively new feature and might change if it does not prove useful, so please try to keep up to date.

Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is not possible at all with "plain Solidity" and the idea is that assembly libraries will be used to enhance the language in such ways.

```
library GetCode {
    function at(address _addr) returns (bytes o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

Inline assembly could also be beneficial in cases where the optimizer fails to produce efficient code. Please be aware that assembly is much more difficult to write because the compiler does not perform checks, so you should use it only if you really know what you are doing.

```
library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := mload(add(add(_data, 0x20), mul(i, 0x20)))
            }
        }
    }
}
```

Syntax

Assembly parses comments, literals and identifiers exactly as Solidity, so you can use the usual `//` and `/* */` comments. Inline assembly is marked by `assembly { ... }` and inside these curly braces, the following can be used (see the later sections for more details)

- literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)
- opcodes (in "instruction style"), e.g. `mload sload dup1 sstore`, for a list see below
- opcode in functional style, e.g. `add(1, mlod(0))`
- labels, e.g. `name:`
- variable declarations, e.g. `let x := 7` or `let x := add(y, 3)`

- identifiers (labels or assembly-local variables and externals if used as inline assembly), e.g. `jump(name)`, `3 x add`
- assignments (in "instruction style"), e.g. `3 =: x`
- assignments in functional style, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`

Opcodes

This document does not want to be a full description of the Ethereum virtual machine, but the following list can be used as a reference of its opcodes.

If an opcode takes arguments (always from the top of the stack), they are given in parentheses. Note that the order of arguments can be seed to be reversed in non-functional style (explained below). Opcodes marked with − do not push an item onto the stack, those marked with ⋆ are special and all others push exactly one item onte the stack.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to (excluding) position `b` and `storage[p]` signifies the storage contents at position `p`.

The opcodes `pushi` and `jumpdest` cannot be used directly.

In the grammar, opcodes are represented as pre-defined identifiers.

stop	-	stop execution, identical to return(0,0)
add(x, y)		x + y
sub(x, y)		x - y
mul(x, y)		x * y
div(x, y)		x / y
sdiv(x, y)		x / y, for signed numbers in two's complement
mod(x, y)		x % y
smod(x, y)		x % y, for signed numbers in two's complement
exp(x, y)		x to the power of y
not(x)		~x, every bit of x is negated
lt(x, y)		1 if x < y, 0 otherwise
gt(x, y)		1 if x > y, 0 otherwise
slt(x, y)		1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)		1 if x == y, 0 otherwise
iszero(x)		1 if x == 0, 0 otherwise
and(x, y)		bitwise and of x and y
or(x, y)		bitwise or of x and y
xor(x, y)		bitwise xor of x and y
byte(n, x)		nth byte of x, where the most significant byte is the 0th byte
addmod(x, y, m)		(x + y) % m with arbitrary precision arithmetics
mulmod(x, y, m)		(x * y) % m with arbitrary precision arithmetics
signextend(i, x)		sign extend from (i*8+7)th bit counting from least significant
sha3(p, n)		keccak(mem[p...(p+n)))
jump(label)	-	jump to label / code position
jumpi(label, cond)	-	jump to label if cond is nonzero
pc		current position in code

Table 6.2 – continued from previous page

pop	*	remove topmost stack slot
dup1 ... dup16		copy ith stack slot to the top (counting from top)
swap1 ... swap16	*	swap topmost and ith stack slot below it
mload(p)		mem[p..(p+32))
mstore(p, v)	-	mem[p..(p+32)) := v
mstore8(p, v)	-	mem[p] := v & 0xff - only modifies a single byte
sload(p)		storage[p]
sstore(p, v)	-	storage[p] := v
msize		size of memory, i.e. largest accessed memory index
gas		gas still available to execution
address		address of the current contract / execution context
balance(a)		wei balance at address a
caller		call sender (excluding delegatecall)
callvalue		wei sent together with the current call
calldataload(p)		call data starting from position p (32 bytes)
calldatasize		size of call data in bytes
calldatacopy(t, f, s)	-	copy s bytes from calldata at position f to mem at position t
codesize		size of the code of the current contract / execution context
codecopy(t, f, s)	-	copy s bytes from code at position f to mem at position t
extcodesize(a)		size of the code at address a
extcodecopy(a, t, f, s)	-	like codecopy(t, f, s) but take code at address a
create(v, p, s)		create new contract with code mem[p..(p+s)) and send v wei and return the new addre
call(g, a, v, in, insize, out, outsize)		call contract at address a with input mem[in..(in+insize)] providing g gas and v wei a
callcode(g, a, v, in, insize, out, outsize)		identical to call but only use the code from a and stay in the context of the current co
delegatecall(g, a, in, insize, out, outsize)		identical to callcode but also keep `caller` and `callvalue`
return(p, s)	*	end execution, return data mem[p..(p+s))
selfdestruct(a)	*	end execution, destroy current contract and send funds to a
log0(p, s)	-	log without topics and data mem[p..(p+s))
log1(p, s, t1)	-	log with topic t1 and data mem[p..(p+s))
log2(p, s, t1, t2)	-	log with topics t1, t2 and data mem[p..(p+s))
log3(p, s, t1, t2, t3)	-	log with topics t1, t2, t3 and data mem[p..(p+s))
log4(p, s, t1, t2, t3, t4)	-	log with topics t1, t2, t3, t4 and data mem[p..(p+s))
origin		transaction sender
gasprice		gas price of the transaction
blockhash(b)		hash of block nr b - only for last 256 blocks excluding current
coinbase		current mining beneficiary
timestamp		timestamp of the current block in seconds since the epoch
number		current block number
difficulty		difficulty of the current block
gaslimit		block gas limit of the current block

Literals

You can use integer constants by typing them in decimal or hexadecimal notation and an appropriate `PUSHi` instruction will automatically be generated. The following creates code to add 2 and 3 resulting in 5 and then computes the bitwise and with the string “abc”. Strings are stored left-aligned and cannot be longer than 32 bytes.

```
assembly { 2 3 add "abc" and }
```

Functional Style

You can type opcode after opcode in the same way they will end up in bytecode. For example adding `3` to the contents in memory at position `0x80` would be

```
3 0x80 mload add 0x80 mstore
```

As it is often hard to see what the actual arguments for certain opcodes are, Solidity inline assembly also provides a "functional style" notation where the same code would be written as follows

```
mstore(0x80, add(mload(0x80), 3))
```

Functional style and instructional style can be mixed, but any opcode inside a functional style expression has to return exactly one stack slot (most of the opcodes do).

Note that the order of arguments is reversed in functional-style as opposed to the instruction-style way. If you use functional-style, the first argument will end up on the stack top.

Access to External Variables and Functions

Solidity variables and other identifiers can be accessed by simply using their name. For storage and memory variables, this will push the address and not the value onto the stack. Also note that non-struct and non-array storage variable addresses occupy two slots on the stack: One for the address and one for the byte offset inside the storage slot. In assignments (see below), we can even use local Solidity variables to assign to.

Functions external to inline assembly can also be accessed: The assembly will push their entry label (with virtual function resolution applied). The calling semantics in solidity are:

- the caller pushes return label, arg1, arg2, ..., argn
- the call returns with ret1, ret2, ..., retn

This feature is still a bit cumbersome to use, because the stack offset essentially changes during the call, and thus references to local variables will be wrong. It is planned that the stack height changes can be specified in inline assembly.

```
contract C {
    uint b;
    function f(uint x) returns (uint r) {
        assembly {
            b pop // remove the offset, we know it is zero
            sload
            x
            mul
            =: r  // assign to return variable r
        }
    }
}
```

Labels

Another problem in EVM assembly is that `jump` and `jumpi` use absolute addresses which can change easily. Solidity inline assembly provides labels to make the use of jumps easier. The following code computes an element in the Fibonacci series.

```
{
    let n := calldataload(4)
    let a := 1
    let b := a
loop:
    jumpi(loopend, eq(n, 0))
    a add swap1
    n := sub(n, 1)
    jump(loop)
loopend:
    mstore(0, a)
    return(0, 0x20)
}
```

Please note that automatically accessing stack variables can only work if the assembler knows the current stack height. This fails to work if the jump source and target have different stack heights. It is still fine to use such jumps, but you should just not access any stack variables (even assembly variables) in that case.

Furthermore, the stack height analyser goes through the code opcode by opcode (and not according to control flow), so in the following case, the assembler will have a wrong impression about the stack height at label `two`:

```
{
    jump(two)
    one:
        // Here the stack height is 1 (because we pushed 7),
        // but the assembler thinks it is 0 because it reads
        // from top to bottom.
        // Accessing stack variables here will lead to errors.
        jump(three)
    two:
        7 // push something onto the stack
        jump(one)
    three:
}
```

Declaring Assembly-Local Variables

You can use the `let` keyword to declare variables that are only visible in inline assembly and actually only in the current `{...}`-block. What happens is that the `let` instruction will create a new stack slot that is reserved for the variable and automatically removed again when the end of the block is reached. You need to provide an initial value for the variable which can be just `0`, but it can also be a complex functional-style expression.

```
contract C {
    function f(uint x) returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
            } // y is "deallocated" here
            b := add(b, v)
        } // v is "deallocated" here
    }
}
```

Assignments

Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

There are two kinds of assignments: functional-style and instruction-style. For functional-style assignments (`variable := value`), you need to provide a value in a functional-style expression that results in exactly one stack value and for instruction-style (`=: variable`), the value is just taken from the stack top. For both ways, the colon points to the name of the variable. The assignment is performed by replacing the variable's value on the stack by the new value.

```
assembly {
    let v := 0 // functional-style assignment as part of variable declaration
    let g := add(v, 2)
    sload(10)
    =: v // instruction style assignment, puts the result of sload(10) into v
}
```

Switch

You can use a switch statement as a very basic version of "if/else". It takes the value of an expression and compares it to several constants. The branch corresponding to the matching constant is taken. Contrary to the error-prone behaviour of some programming languages, control flow does not continue from one case to the next. There can be a fallback or default case called `default`.

```
assembly {
    let x := 0
    switch calldataload(4)
    case 0: {
        x := calldataload(0x24)
    }
    default: {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases does not require curly braces, but the body of a case does require them.

Loops

Assembly supports a simple for-style loop. For-style loops have a header containing an initializing part, a condition and a post-iteration part. The condition has to be a functional-style expression, while the other two can also be blocks. If the initializing part is a block that declares any variables, the scope of these variables is extended into the body (including the condition and the post-iteration part).

The following example computes the sum of an area in memory.

```
assembly {
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

Functions

Assembly allows the definition of low-level functions. These take their arguments (and a return PC) from the stack and also put the results onto the stack. Calling a function looks the same way as executing a functional-style opcode.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function. There is no explicit `return` statement.

If you call a function that returns multiple values, you have to assign them to a tuple using `(a, b) := f(x)` or `let (a, b) := f(x)`.

The following example implements the power function by square-and-multiply.

```
assembly {
    function power(base, exponent) -> (result) {
        switch exponent
        0: { result := 1 }
        1: { result := base }
        default: {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                1: { result := mul(base, result) }
        }
    }
}
```

Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, managing jump labels, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached. Especially for those two last cases, it is important to know that the assembler only counts stack height from top to bottom, not necessarily following control flow. Furthermore, operations like swap will only swap the contents of the stack but not the location of variables.

Conventions in Solidity

In contrast to EVM assembly, Solidity knows types which are narrower than 256 bits, e.g. `uint24`. In order to make them more efficient, most arithmetic operations just treat them as 256 bit numbers and the higher-order bits are only cleaned at the point where it is necessary, i.e. just shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher order bits first.

Solidity manages memory in a very simple way: There is a "free memory pointer" at position `0x40` in memory. If you want to allocate memory, just use the memory from that point on and update the pointer accordingly.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (yes, this is even true for `byte[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and then only the array elements follow.

Warning: Statically-sized memory arrays do not have a length field, but it will be added soon to allow better convertibility between statically- and dynamically-sized arrays, so please do not rely on that.

Standalone Assembly

The assembly language described as inline assembly above can also be used standalone and in fact, the plan is to use it as an intermediate language for the Solidity compiler. In this form, it tries to achieve several goals:

1. Programs written in it should be readable, even if the code is generated by a compiler from Solidity.
2. The translation from assembly to bytecode should contain as few "surprises" as possible.
3. Control flow should be easy to detect to help in formal verification and optimization.

In order to achieve the first and last goal, assembly provides high-level constructs like `for` loops, `switch` statements and function calls. It should be possible to write assembly programs that do not make use of explicit `SWAP`, `DUP`, `JUMP` and `JUMPI` statements, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

The second goal is achieved by introducing a desugaring phase that only removes the higher level constructs in a very regular way and still allows inspecting the generated low-level assembly code. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...), which follow very simple and regular scoping rules and cleanup of local variables from the stack.

Scoping: An identifier that is declared (label, variable, function, assembly) is only visible in the block where it was declared (including nested blocks inside the current block). It is not legal to access local variables across function borders, even if they would be in scope. Shadowing is allowed, but two identifiers with the same name cannot be declared in the same block. Local variables cannot be accessed before they were declared, but labels, functions and assemblies can. Assemblies are special blocks that are used for e.g. returning runtime code or creating contracts. No identifier from an outer assembly is visible in a sub-assembly.

If control flow passes over the end of a block, pop instructions are inserted that match the number of local variables declared in that block, unless the `}` is directly preceded by an opcode that does not have a continuing control flow path. Whenever a local variable is referenced, the code generator needs to know its current relative position in the stack and thus it needs to keep track of the current so-called stack height. At the end of a block, this implicit stack height is always reduced by the number of local variables whether ther is a continuing control flow or not.

This means that the stack height before and after the block should be the same. If this is not the case, a warning is issued, unless the last instruction in the block did not have a continuing control flow path.

Why do we use higher-level constructs like `switch`, `for` and functions:

Using `switch`, `for` and functions, it should be possible to write complex code without using `jump` or `jumpi` manually. This makes it much easier to analyze the control flow, which allows for improved formal verification and optimization.

Furthermore, if manual jumps are allowed, computing the stack height is rather complicated. The position of all local variables on the stack needs to be known, otherwise neither references to local variables nor removing local variables automatically from the stack at the end of a block will work properly. Because of that, every label that is preceded by an instruction that ends or diverts control flow should be annotated with the current stack layout. This annotation is performed automatically during the desugaring phase.

Example:

We will follow an example compilation from Solidity to desugared assembly. We consider the runtime bytecode of the following Solidity program:

```
contract C {
  function f(uint x) returns (uint y) {
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
```

```
  }
}
```

The following assembly will be generated:

```
{
  mstore(0x40, 0x60) // store the "free memory pointer"
  // function dispatcher
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b: {
    let (r) = f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
    return(ret, 0x20)
  }
  default: { jump(invalidJumpLabel) }
  // memory allocator
  function $allocate(size) -> (pos) {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // the contract function
  function f(x) -> (y) {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

After the desugaring phase it looks as follows:

```
{
  mstore(0x40, 0x60)
  {
    let $0 := div(calldataload(0), exp(2, 226))
    jumpi($case1, eq($0, 0xb3de648b))
    jump($caseDefault)
    $case1:
    {
      // the function call - we put return label and arguments on the stack
      $ret1 calldataload(4) jump($fun_f)
      $ret1 [r]: // a label with a [...]-annotation resets the stack height
                 // to "current block + number of local variables". It also
                 // introduces a variable, r:
                 // r is at top of stack, $0 is below (from enclosing block)
      $ret2 0x20 jump($fun_allocate)
      $ret2 [ret]: // stack here: $0, r, ret (top)
      mstore(ret, r)
      return(ret, 0x20)
      // although it is useless, the jump is automatically inserted,
      // since the desugaring process does not analyze control-flow
      jump($endswitch)
    }
    $caseDefault:
    {
      jump(invalidJumpLabel)
      jump($endswitch)
    }
```

```
    $endswitch:
  }
  jump($afterFunction)
  $fun_allocate:
  {
    $start[$retpos, size]:
    // output variables live in the same scope as the arguments.
    let pos := 0
    {
      pos := mload(0x40)
      mstore(0x40, add(pos, size))
    }
    swap1 pop swap1 jump
  }
  $fun_f:
  {
    start [$retpos, x]:
    let y := 0
    {
      let i := 0
      $for_begin:
      jumpi($for_end, iszero(lt(i, x)))
      {
        y := mul(2, y)
      }
      $for_continue:
      { i := add(i, 1) }
      jump($for_begin)
      $for_end:
    } // Here, a pop instruction is inserted for i
    swap1 pop swap1 jump
  }
  $afterFunction:
  stop
}
```

Assembly happens in four stages:

1. Parsing
2. Desugaring (removes switch, for and functions)
3. Opcode stream generation
4. Bytecode generation

We will specify steps one to three in a pseudo-formal way. More formal specifications will follow.

Parsing / Grammar

The tasks of the parser are the following:

- Turn the byte stream into a token stream, discarding C++-style comments (a special comment exists for source references, but we will not explain it here).
- Turn the token stream into an AST according to the grammar below
- Register identifiers with the block they are defined in (annotation to the AST node) and note from which point on, variables can be accessed.

The assembly lexer follows the one defined by Solidity itself.

Whitespace is used to delimit tokens and it consists of the characters Space, Tab and Linefeed. Comments are regular JavaScript/C++ comments and are interpreted in the same way as Whitespace.

Grammar:

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
    AssemblyBlock |
    FunctionalAssemblyExpression |
    AssemblyLocalDefinition |
    FunctionalAssemblyAssignment |
    AssemblyAssignment |
    LabelDefinition |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
    'break' | 'continue' |
    SubAssembly | 'dataSize' '(' Identifier ')' |
    LinkerSymbol |
    'errorLabel' | 'bytecodeSize' |
    NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9]*
FunctionalAssemblyExpression = Identifier '(' ( AssemblyItem ( ',' AssemblyItem )* )? ')'
AssemblyLocalDefinition = 'let' IdentifierOrList ':=' FunctionalAssemblyExpression
FunctionalAssemblyAssignment = IdentifierOrList ':=' FunctionalAssemblyExpression
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyAssignment = '=:' Identifier
LabelDefinition = Identifier ( '[' ( IdentifierList | NumberLiteral ) ']' )? ':'
AssemblySwitch = 'switch' FunctionalAssemblyExpression AssemblyCase*
    ( 'default' ':' AssemblyBlock )?
AssemblyCase = 'case' FunctionalAssemblyExpression ':' AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | FunctionalAssemblyExpression)
    FunctionalAssemblyExpression ( AssemblyBlock | FunctionalAssemblyExpression) AssemblyBlock
SubAssembly = 'assembly' Identifier AssemblyBlock
LinkerSymbol = 'linkerSymbol' '(' StringLiteral ')'
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Desugaring

An AST transformation removes for, switch and function constructs. The result is still parseable by the same parser, but it will not use certain constructs. If jumpdests are added that are only jumped to and not continued at, information about the stack content is added, unless no local variables of outer scopes are accessed or the stack height is the same as for the previous instruction.

Pseudocode:

```
desugar item: AST -> AST =
match item {
```

```
AssemblyFunctionDefinition('function' name '(' arg1, ..., argn ')' '->' ( '(' ret1, ..., retm ')' boc
  <name>:
  {
    $<name>_start [$retPC, $argn, ..., arg1]:
    let ret1 := 0 ... let retm := 0
    { desugar(body) }
    swap and pop items so that only ret1, ... retn, $retPC are left on the stack
    jump
  }
AssemblyFor('for' { init } condition post body) ->
  {
    init // cannot be its own block because we want variable scope to extend into the body
    // find I such that there are no labels $forI_*
    $forI_begin:
    jumpi($forI_end, iszero(condition))
    { body }
    $forI_continue:
    { post }
    jump($forI_begin)
    $forI_end:
  }
'break' ->
  {
    // find nearest enclosing scope with label $forI_end
    pop all local variables that are defined at the current point
    but not at $forI_end
    jump($forI_end)
  }
'continue' ->
  {
    // find nearest enclosing scope with label $forI_continue
    pop all local variables that are defined at the current point
    but not at $forI_continue
    jump($forI_continue)
  }
AssemblySwitch(switch condition cases ( default: defaultBlock )? ) ->
  {
    // find I such that there is no $switchI* label or variable
    let $switchI_value := condition
    for each of cases match {
      case val: -> jumpi($switchI_caseJ, eq($switchI_value, val))
    }
    if default block present: ->
      { defaultBlock jump($switchI_end) }
    for each of cases match {
      case val: { body } -> $switchI_caseJ: { body jump($switchI_end) }
    }
    $switchI_end:
  }
FunctionalAssemblyExpression( identifier(arg1, arg2, ..., argn) ) ->
  {
    if identifier is function <name> with n args and m ret values ->
      {
        // find I such that $funcallI_* does not exist
        $funcallI_return argn  ... arg2 arg1 jump(<name>)
        if the current context is `let (id1, ..., idm) := f(...)` ->
          $funcallI_return [id1, ..., idm]:
        else ->
```

```
          $funcallI_return[m - n - 1]:
          turn the functional expression that leads to the function call
          into a statement stream
      }
    else -> desugar(children of node)
  }
default node ->
  desugar(children of node)
}
```

Opcode Stream Generation

During opcode stream generation, we keep track of the current stack height, so that accessing stack variables by name is possible.

Pseudocode:

```
codegen item: AST -> opcode_stream =
match item {
AssemblyBlock({ items }) ->
  join(codegen(item) for item in items)
  if last generated opcode has continuing control flow:
    POP for all local variables registered at the block (including variables
    introduced by labels)
    warn if the stack height at this point is not the same as at the start of the block
Identifier(id) ->
  lookup id in the syntactic stack of blocks
  match type of id
    Local Variable ->
      DUPi where i = 1 + stack_height - stack_height_of_identifier(id)
    Label ->
      // reference to be resolved during bytecode generation
      PUSH<bytecode position of label>
    SubAssembly ->
      PUSH<bytecode position of subassembly data>
FunctionalAssemblyExpression(id ( arguments ) ) ->
  join(codegen(arg) for arg in arguments.reversed())
  id (which has to be an opcode, might be a function name later)
AssemblyLocalDefinition(let (id1, ..., idn) := expr) ->
  register identifiers id1, ..., idn as locals in current block at current stack height
  codegen(expr) - assert that expr returns n items to the stack
FunctionalAssemblyAssignment((id1, ..., idn) := expr) ->
  lookup id1, ..., idn in the syntactic stack of blocks, assert that they are variables
  codegen(expr)
  for j = n, ..., i:
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(idj)
  POP
AssemblyAssignment(=: id) ->
  look up id in the syntactic stack of blocks, assert that it is a variable
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(id)
  POP
LabelDefinition(name [id1, ..., idn] :) ->
  JUMPDEST
  // register new variables id1, ..., idn and set the stack height to
  // stack_height_at_block_start + number_of_local_variables
LabelDefinition(name [number] :) ->
  JUMPDEST
```

```
  // adjust stack height by +number (can be negative)
NumberLiteral(num) ->
  PUSH<num interpreted as decimal and right-aligned>
HexLiteral(lit) ->
  PUSH32<lit interpreted as hex and left-aligned>
StringLiteral(lit) ->
  PUSH32<lit utf-8 encoded and left-aligned>
SubAssembly(assembly <name> block) ->
  append codegen(block) at the end of the code
dataSize(<name>) ->
  assert that <name> is a subassembly ->
  PUSH32<size of code generated from subassembly <name>>
linkerSymbol(<lit>) ->
  PUSH32<zeros> and append position to linker table
}
```

6.4.8 Miscellaneous

Layout of State Variables in Storage

Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position `0`. Multiple items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only that many bytes that are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

Warning: When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.
It is only beneficial to use reduced-size arguments if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.
Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and `struct` members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

The elements of structs and arrays are stored after each other, just as if they were given explicitly.

Due to their unpredictable size, mapping and dynamically-sized array types use a Keccak-256 hash computation to find the starting position of the value or the array data. These starting positions are always full stack slots.

The mapping or the dynamic array itself occupies an (unfilled) slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings to mappings or arrays of arrays). For a dynamic array, this slot stores the number of elements in the array (byte arrays and strings are an exception here, see below). For a mapping, the slot is unused (but it is needed so that two equal mappings after each other will use a different hash distribution). Array data is located at `keccak256(p)` and the value corresponding to a mapping key `k` is located

at `keccak256(k . p)` where `.` is concatenation. If the value is again a non-elementary type, the positions are found by adding an offset of `keccak256(k . p)`.

`bytes` and `string` store their data in the same slot where also the length is stored if they are short. In particular: If the data is at most `31` bytes long, it is stored in the higher-order bytes (left aligned) and the lowest-order byte stores `length * 2`. If it is longer, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`.

So for the following contract snippet:

```
contract C {
  struct s { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => s)) data;
}
```

The position of `data[4][9].b` is at `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`.

Layout in Memory

Solidity reserves three 256-bit slots:

- 0 - 64: scratch space for hashing methods
- 64 - 96: currently allocated memory size (aka. free memory pointer)

Scratch space can be used between statements (ie. within inline assembly).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Warning: There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifecycle, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one shouldn't expect the free memory to be zeroed out.

Layout of Call Data

When a Solidity contract is deployed and when it is called from an account, the input data is assumed to be in the format in the ABI specification. The ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Internals - Cleaning Up Variables

When a value is shorter than 256-bit, in some cases the remaining bits must be cleaned. The Solidity compiler is designed to clean such remaining bits before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to the memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

On the other hand, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

Different types have different rules for cleaning up invalid values:

Type	Valid Values	Invalid Values Mean
enum of n members	0 until n - 1	exception
bool	0 or 1	1
signed integers	sign-extended word	currently silently wraps; in the future exceptions will be thrown
unsigned integers	higher bits zeroed	currently silently wraps; in the future exceptions will be thrown

Esoteric Features

There are some types in Solidity's type system that have no counterpart in the syntax. One of these types are the types of functions. But still, using `var` it is possible to have local variables of these types:

```
contract FunctionSelector {
  function select(bool useB, uint x) returns (uint z) {
    var f = a;
    if (useB) f = b;
    return f(x);
  }

  function a(uint x) returns (uint z) {
    return x * x;
  }

  function b(uint x) returns (uint z) {
    return 2 * x;
  }
}
```

Calling `select(false, x)` will compute `x * x` and `select(true, x)` will compute `2 * x`.

Internals - The Optimizer

The Solidity optimizer operates on assembly, so it can be and also is used by other languages. It splits the sequence of instructions into basic blocks at JUMPs and JUMPDESTs. Inside these blocks, the instructions are analysed and every modification to the stack, to memory or storage is recorded as an expression which consists of an instruction and a list of arguments which are essentially pointers to other expressions. The main idea is now to find expressions that are always equal (on every input) and combine them into an expression class. The optimizer first tries to find each new expression in a list of already known expressions. If this does not work, the expression is simplified according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is done recursively, we can also apply the latter rule if the second factor is a more complex expression where we know that it will always evaluate to one. Modifications to storage and memory locations have to erase knowledge about storage and memory locations which are not known to be different: If we first write to location x and then to location y and both are input variables, the second could overwrite the first, so we actually do not know what is stored at x after we wrote to y. On the other hand, if a simplification of the expression x - y evaluates to a non-zero constant, we know that we can keep our knowledge about what is stored at x.

At the end of this process, we know which expressions have to be on the stack in the end and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s). If we know the targets of all JUMP and JUMPI instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown JUMP. If a JUMPI is found whose condition evaluates to a constant, it is transformed to an unconditional jump.

As the last step, the code in each block is completely re-generated. A dependency graph is created from the expressions on the stack at the end of the block and every operation that is not part of this graph is essentially dropped. Now code is generated that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed) and finally, generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a JUMPI and during the analysis, the condition evaluates to a constant, the JUMPI is replaced depending on the value of the constant, and thus code like

```
var x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

is simplified to code which can also be compiled from

```
data[7] = 9;
return 1;
```

even though the instructions contained a jump in the beginning.

Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling.

Both kinds of source mappings use integer indentifiers to refer to source files. These are regular array indices into a list of source files usually called `"sourceList"`, which is part of the combined-json and the output of the json / npm compiler.

The source mappings inside the AST use the following notation:

`s:l:f`

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated: It is a list of `s:l:f:j` separated by `;`. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above and `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop.

In order to compress these source mappings especially for bytecode, the following rules are used:

- If a field is empty, the value of the preceding element is used.
- If a `:` is missing, all following fields are considered empty.

This means the following source mappings represent the same information:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2::2;;`

Using the Commandline Compiler

One of the build targets of the Solidity repository is `solc`, the solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. Before you deploy your contract, activate the optimizer while compiling using `solc --optimize --bin sourceFile.sol`. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast --asm sourceFile.sol`.

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide path redirects using `context:prefix=path` in the following way:

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ =/usr/local/lib/fallback file.so
```

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin` and if it does not find the file there, it will look at `/usr/local/lib/fallback` (the empty prefix always matches). `solc` will not read files from the filesystem that lie outside of the remapping targets and outside of the directories where explicitly specified source files reside, so things like `import "/etc/passwd";` only work if you add `=/` as a remapping.

You can restrict remappings to only certain source files by prefixing a context.

The section on *Importing other Source Files* provides more details on remappings.

If there are multiple matches due to remappings, the one with the longest common prefix is selected.

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__LibraryName_____`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points:

Either add `--libraries "Math:0x1234567890123456789 Heap:0xabcdef0123456"` to your command to provide an address for each library or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__LibraryName_____`-format given above and are linked in-place (if the input is read from stdin, it is written to stdout). All options except `--libraries` are ignored (including `-o`) in this case.

Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the current contract. It can be used to query the compiler version, the sources used, the ABI and NatSpec documentation in order to more safely interact with the contract and to verify its source code.

The compiler appends a Swarm hash of the metadata file to the end of the bytecode (for details, see below) of each contract, so that you can retrieve the file in an authenticated way without having to resort to a centralized data provider.

Of course, you have to publish the metadata file to Swarm (or some other service) so that others can access it. The file can be output by using `solc --metadata` and the file will be called `ContractName_meta.json`. It will contain Swarm references to the source code, so you have to upload all source files and the metadata file.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum and sort the keys of all objects to arrive at a unique formatting. Comments are of course also not permitted and used here only for explanatory purposes.

```
{
  // Required: The version of the metadata format
  version: "1",
```

```
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  language: "Solidity",
  // Required: Details about the compiler, contents are specific
  // to the language.
  compiler: {
    // Required for Solidity: Version of the compiler
    version: "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Optional: Hash of the compiler binary which produced this output
    keccak256: "0x123..."
  },
  // Required: Compilation source files/source units, keys are file names
  sources:
  {
    "myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but a
      // Swarm URL is recommended
      "urls": [ "bzzr://56ab..." ]
    },
    "mortal": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract mortal is owned { function kill() { if (msg.sender == owner) selfdestruct
    }
  },
  // Required: Compiler settings
  settings:
  {
    // Required for Solidity: Sorted list of remappings
    remappings: [ ":g/dir" ],
    // Optional: Optimizer settings (enabled defaults to false)
    optimizer: {
      enabled: true,
      runs: 500
    },
    // Required for Solidity: File and name of the contract or library this
    // metadata is created for.
    compilationTarget: {
      "myFile.sol": "MyContract"
    },
    // Required for Solidity: Addresses for libraries used
    libraries: {
      "MyLib": "0x123123..."
    }
  },
  // Required: Generated information about the contract.
  output:
  {
    // Required: ABI definition of the contract
    abi: [ ... ],
    // Required: NatSpec user documentation of the contract
    userdoc: [ ... ],
    // Required: NatSpec developer documentation of the contract
    devdoc: [ ... ],
```

```
  }
}
```

Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping `{"bzzr0": <Swarm hash>}` is stored [CBOR](https://tools.ietf.org/html/rfc7049)-encoded. Since the beginning of that encoding is not easy to find, its length is added in a two-byte big-endian encoding. The current version of the Solidity compiler thus adds the following to the end of the deployed bytecode:

```
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and use the Swarm hash to retrieve the file.

Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way: A component that wants to interact with a contract (e.g. Mist) retrieves the code of the contract, from that the Swarm hash of a file which is then retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, Mist can use the userdoc to display a confirmation message to the user whenever they interact with the contract.

Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from Swarm via the link in the metadata file. The compiler of the correct version (which is checked to be part of the "official" compilers) is invoked on that input with the specified settings. The resulting bytecode is compared to the data of the creation transaction or CREATE opcode data. This automatically verifies the metadata since its hash is part of the bytecode. Excess data corresponds to the constructor input data, which should be decoded according to the interface and presented to the user.

Tips and Tricks

- Use `delete` on arrays to delete all its elements.
- Use shorter types for struct elements and sort them such that short types are grouped together. This can lower the gas costs as multiple SSTORE operations might be combined into a single (SSTORE costs 5000 or 20000 gas, so this is what you want to optimise). Use the gas price estimator (with optimiser enabled) to check!
- Make your state variables public - the compiler will create *getters* for you for free.
- If you end up checking conditions on input or state a lot at the beginning of your functions, try using *Function Modifiers*.
- If your contract has a function called `send` but you want to use the built-in send-function, use `address(contractVariable).send(amount)`.
- Initialise storage structs with a single assignment: `x = MyStruct({a: 1, b: 2});`

Cheatsheet

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	`++`, `--`
	Function-like call	`<func>(<args...>)`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++`, `--`
	Unary plus and minus	`+`, `-`
	Unary operations	`delete`
	Logical NOT	`!`
	Bitwise NOT	`~`
3	Exponentiation	`**`
4	Multiplication, division and modulo	`*`, `/`, `%`
5	Addition and subtraction	`+`, `-`
6	Bitwise shift operators	`<<`, `>>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<`, `>`, `<=`, `>=`
11	Equality operators	`==`, `!=`
12	Logical AND	`&&`
13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
15	Assignment operators	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
16	Comma operator	`,`

Global Variables

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas
- `msg.sender` (`address`): sender of the message (current call)
- `msg.value` (`uint`): number of wei sent with the message

- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)
- `keccak256(...) returns (bytes32)`: compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments
- `sha3(...) returns (bytes32)`: an alias to *keccak256()*
- `sha256(...) returns (bytes32)`: compute the SHA-256 hash of the (tightly packed) arguments
- `ripemd160(...) returns (bytes20)`: compute the RIPEMD-160 hash of the (tightly packed) arguments
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)`: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`
- `mulmod(uint x, uint y, uint k) returns (uint)`: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`
- `this` (current contract's type): the current contract, explicitly convertible to `address`
- `super`: the contract one level higher in the inheritance hierarchy
- `selfdestruct(address recipient)`: destroy the current contract, sending its funds to the given address
- `<address>.balance` (`uint256`): balance of the address in Wei
- `<address>.send(uint256 amount) returns (bool)`: send given amount of Wei to address, returns `false` on failure

Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externally and internally (creates accessor function for storage/state variables)
- `private`: only visible in the current contract
- `external`: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal`: only visible internally

Modifiers

- `constant` for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- `constant` for functions: Disallows modification of state - this is not enforced yet.
- `anonymous` for events: Does not store event signature as topic.
- `indexed` for event parameters: Stores the parameter as topic.
- `payable` for functions: Allows them to receive Ether together with a call.

Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

`abstract`, `after`, `case`, `catch`, `default`, `final`, `in`, `inline`, `interface`, `let`, `match`, `null`, `of`, `pure`, `relocatable`, `static`, `switch`, `try`, `type`, `typeof`, `view`.

Language Grammar

```
SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

// Pragma actually parses anything up to the trailing ';' to be fully forward-compatible.
PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
        | 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
        | 'import' '{' Identifier ('as' Identifier)? ( ',' Identifier ('as' Identifier)? )* '}' 'fro

ContractDefinition = ( 'contract' | 'library' ) Identifier
                     ( 'is' InheritanceSpecifier (',' InheritanceSpecifier )* )?
                     '{' ContractPart* '}'

ContractPart = StateVariableDeclaration | UsingForDeclaration
             | StructDefinition | ModifierDefinition | FunctionDefinition | EventDefinition | EnumDe

InheritanceSpecifier = UserDefinedTypeName ( '(' Expression ( ',' Expression )* ')' )?

StateVariableDeclaration = TypeName ( 'public' | 'internal' | 'private' )? Identifier ('=' Expressio
UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'
StructDefinition = 'struct' Identifier '{'
                     ( VariableDeclaration ';' (VariableDeclaration ';')* )? '}'
ModifierDefinition = 'modifier' Identifier ParameterList? Block
FunctionDefinition = 'function' Identifier? ParameterList
                     ( FunctionCall | Identifier | 'constant' | 'payable' | 'external' | 'public' |
                     ( 'returns' ParameterList )? ( ';' | Block )
EventDefinition = 'event' Identifier IndexedParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

IndexedParameterList = '(' ( TypeName 'indexed'? Identifier? (',' TypeName 'indexed'? Identifier?)* )
ParameterList =        '(' ( TypeName            Identifier? (',' TypeName            Identifier?)* )
TypeNameList =         '(' ( TypeName (',' TypeName )* )? ')'

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
         | Mapping
         | ArrayTypeName
         | FunctionTypeName

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
```

```
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' TypeNameList ( 'internal' | 'external' | 'constant' | 'payable' )*
                   ( 'returns' TypeNameList )?
StorageLocation = 'memory' | 'storage'

Block = '{' Statement* '}'
Statement = IfStatement | WhileStatement | ForStatement | Block | InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | SimpleStatement ) ';'

ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
WhileStatement = 'while' '(' Expression ')' Statement
PlaceholderStatement = '_'
SimpleStatement = VariableDefinition | ExpressionStatement
ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';' (ExpressionStatement)? ')' Stateme
InlineAssemblyStatement = 'assembly' InlineAssemblyBlock
DoWhileStatement = 'do' Statement 'while' '(' Expression ')'
Continue = 'continue'
Break = 'break'
Return = 'return' Expression?
Throw = 'throw'
VariableDefinition = VariableDeclaration ( '=' Expression )?

// Precedence by order (see github.com/ethereum/solidity/pull/732)
Expression =
  ( Expression ('++' | '--') | FunctionCall | IndexAccess | MemberAccess | '(' Expression ')' )
  | ('!' | '~' | 'delete' | '++' | '--' | '+' | '-') Expression
  | Expression '**' Expression
  | Expression ('*' | '/' | '%') Expression
  | Expression ('+' | '-') Expression
  | Expression ('<<' | '>>') Expression
  | Expression '&' Expression
  | Expression '^' Expression
  | Expression '|' Expression
  | Expression ('<' | '>' | '<=' | '>=') Expression
  | Expression ('==' | '!=') Expression
  | Expression '&&' Expression
  | Expression '||' Expression
  | Expression '?' Expression ':' Expression
  | Expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/=' | '%=') Expressi
  | Expression? (',' Expression)
  | PrimaryExpression

PrimaryExpression = Identifier
                  | BooleanLiteral
                  | NumberLiteral
                  | HexLiteral
                  | StringLiteral
                  | ElementaryTypeNameExpression

ExpressionList = Expression ( ',' Expression )*
NameValueList = Identifier ':' Expression ( ',' Identifier ':' Expression )*

FunctionCall = ( PrimaryExpression | NewExpression | TypeName ) ( ( '.' Identifier ) | ( '[' Expressi
FunctionCallArguments = '{' NameValueList? '}'
                      | ExpressionList?
```

```
NewExpression = 'new' TypeName
MemberAccess = Expression '.' Identifier
IndexAccess = Expression '[' Expression? ']'

BooleanLiteral = 'true' | 'false'
NumberLiteral = ( HexNumber | DecimalNumber ) (' ' NumberUnit)?
NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether'
           | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*

HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | 'var'
                   | Int | Uint | Byte | Fixed | Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' | 'int64' | 'int72'

Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' | 'uint56' | 'uint64'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' | 'bytes6' | 'bytes7'

Fixed = 'fixed' | 'fixed0x8' | 'fixed0x16' | 'fixed0x24' | 'fixed0x32' | 'fixed0x40' | 'fixed0x48' |

Ufixed = 'ufixed' | 'ufixed0x8' | 'ufixed0x16' | 'ufixed0x24' | 'ufixed0x32' | 'ufixed0x40' | 'ufixed

InlineAssemblyBlock = '{' AssemblyItem* '}'

AssemblyItem = Identifier | FunctionalAssemblyExpression | InlineAssemblyBlock | AssemblyLocalBinding
AssemblyLocalBinding = 'let' Identifier ':=' FunctionalAssemblyExpression
AssemblyAssignment = Identifier ':=' FunctionalAssemblyExpression | '=:' Identifier
FunctionalAssemblyExpression = Identifier '(' AssemblyItem? ( ',' AssemblyItem )* ')'
```

6.5 Security Considerations

While it is usually quite easy to build software that works as expected, it is much harder to check that nobody can use it in a way that was **not** anticipated.

In Solidity, this is even more important because you can use smart contracts to handle tokens or, possibly, even more valuable things. Furthermore, every execution of a smart contract happens in public and, in addition to that, the source code is often available.

Of course you always have to consider how much is at stake: You can compare a smart contract with a web service that is open to the public (and thus, also to malicous actors) and perhaps even open source. If you only store your grocery list on that web service, you might not have to take too much care, but if you manage your bank account using that web service, you should be more careful.

This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug.

As always, with open source documentation, please help us extend this section (especially, some examples would not hurt)!

6.5.1 Pitfalls

Private Information and Randomness

Everything you use in a smart contract is publicly visible, even local variables and state variables marked `private`.

Using random numbers in smart contracts is quite tricky if you do not want miners to be able to cheat.

Re-Entrancy

Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed. To give an example, the following code contains a bug (it is just a snippet and not a complete contract):

```
pragma solidity ^0.4.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

The problem is not too serious here because of the limited gas as part of `send`, but it still exposes a weakness: Ether transfer always includes code execution, so the recipient could be a contract that calls back into `withdraw`. This would let it get multiple refunds and basically retrieve all the Ether in the contract.

To avoid re-entrancy, you can use the Checks-Effects-Interactions pattern as outlined further below:

```
pragma solidity ^0.4.0;

contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        if (!msg.sender.send(share))
            throw;
    }
}
```

Note that re-entrancy is not only an effect of Ether transfer but of any function call on another contract. Furthermore, you also have to take multi-contract situations into account. A called contract could modify the state of another contract you depend on.

Gas Limit and Loops

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to `constant` functions that are only executed

to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

Sending and Receiving Ether

- Neither contracts nor "external accounts" are currently able to prevent that someone sends them Ether. Contracts can react on and reject a regular transfer, but there are ways to move Ether without creating a message call. One way is to simply "mine to" the contract address and the second way is using `selfdestruct(x)`.
- If a contract receives Ether (without a function being called), the fallback function is executed. If it does not have a fallback function, the Ether will be rejected (by throwing an exception). During the execution of the fallback function, the contract can only rely on the "gas stipend" (2300 gas) being available to it at that time. This stipend is not enough to access storage in any way. To be sure that your contract can receive Ether in that way, check the gas requirements of the fallback function (for example in the "details" section in browser-solidity).
- There is a way to forward more gas to the receiving contract using `addr.call.value(x)()`. This is essentially the same as `addr.send(x)`, only that it forwards all remaining gas and opens up the ability for the recipient to perform more expensive actions. This might include calling back into the sending contract or other state changes you might not have thought of. So it allows for great flexibility for honest users but also for malicious actors.
- If you want to send Ether using `address.send`, there are certain details to be aware of:
  1. If the recipient is a contract, it causes its fallback function to be executed which can, in turn, call back the sending contract.
  2. Sending Ether can fail due to the call depth going above 1024. Since the caller is in total control of the call depth, they can force the transfer to fail; make sure to always check the return value of `send`. Better yet, write your contract using a pattern where the recipient can withdraw Ether instead.
  3. Sending Ether can also fail because the execution of the recipient contract requires more than the allotted amount of gas (explicitly by using `throw` or because the operation is just too expensive) - it "runs out of gas" (OOG). If the return value of `send` is checked, this might provide a means for the recipient to block progress in the sending contract. Again, the best practice here is to use a *"withdraw" pattern instead of a "send" pattern*.

Callstack Depth

External function calls can fail any time because they exceed the maximum call stack of 1024. In such situations, Solidity throws an exception. Malicious actors might be able to force the call stack to a high value before they interact with your contract.

Note that `.send()` does **not** throw an exception if the call stack is depleted but rather returns `false` in that case. The low-level functions `.call()`, `.callcode()` and `.delegatecall()` behave in the same way.

tx.origin

Never use tx.origin for authorization. Let's say you have a wallet contract like this:

```
pragma solidity ^0.4.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    function TxUserWallet() {
```

```
        owner = msg.sender;
    }

    function transfer(address dest, uint amount) {
        if (tx.origin != owner) { throw; }
        if (!dest.call.value(amount)()) throw;
    }
}
```

Now someone tricks you into sending ether to the address of this attack wallet:

```
pragma solidity ^0.4.0;

contract TxAttackWallet {
    address owner;

    function TxAttackWallet() {
        owner = msg.sender;
    }

    function() {
        TxUserWallet(msg.sender).transfer(owner, msg.sender.balance);
    }
}
```

If your wallet had checked `msg.sender` for authorization, it would get the address of the attack wallet, instead of the owner address. But by checking `tx.origin`, it gets the original address that kicked off the transaction, which is still the owner address. The attack wallet instantly drains all your funds.

Minor Details

- In `for (var i = 0; i < arrayName.length; i++) { ... }`, the type of `i` will be `uint8`, because this is the smallest type that is required to hold the value `0`. If the array has more than 255 elements, the loop will not terminate.
- The `constant` keyword for functions is currently not enforced by the compiler. Furthermore, it is not enforced by the EVM, so a contract function that "claims" to be constant might still cause changes to the state.
- Types that do not occupy the full 32 bytes might contain "dirty higher order bits". This is especially important if you access `msg.data` - it poses a malleability risk: You can craft transactions that call a function `f(uint8 x)` with a raw byte argument of `0xff000001` and with `0x00000001`. Both are fed to the contract and both will look like the number `1` as far as `x` is concerned, but `msg.data` will be different, so if you use `keccak256(msg.data)` for anything, you will get different results.

6.5.2 Recommendations

Restrict the Amount of Ether

Restrict the amount of Ether (or other tokens) that can be stored in a smart contract. If your source code, the compiler or the platform has a bug, these funds may be lost. If you want to limit your loss, limit the amount of Ether.

Keep it Small and Modular

Keep your contracts small and easily understandable. Single out unrelated functionality in other contracts or into libraries. General recommendations about source code quality of course apply: Limit the amount of local variables,

the length of functions and so on. Document your functions so that others can see what your intention was and whether it is different than what the code does.

Use the Checks-Effects-Interactions Pattern

Most functions will first perform some checks (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.). These checks should be done first.

As the second step, if all checks passed, effects to the state variables of the current contract should be made. Interaction with other contracts should be the very last step in any function.

Early contracts delayed some effects and waited for external function calls to return in a non-error state. This is often a serious mistake because of the re-entrancy problem explained above.

Note that, also, calls to known contracts might in turn cause calls to unknown contracts, so it is probably better to just always apply this pattern.

Include a Fail-Safe Mode

While making your system fully decentralised will remove any intermediary, it might be a good idea, especially for new code, to include some kind of fail-safe mechanism:

You can add a function in your smart contract that performs some self-checks like "Has any Ether leaked?", "Is the sum of the tokens equal to the balance of the contract?" or similar things. Keep in mind that you cannot use too much gas for that, so help through off-chain computations might be needed there.

If the self-check fails, the contract automatically switches into some kind of "failsafe" mode, which, for example, disables most of the features, hands over control to a fixed and trusted third party or just converts the contract into a simple "give me back my money" contract.

6.5.3 Formal Verification

Using formal verification, it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler. There is a prototype in Solidity that performs formal verification and it will be better documented soon.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

6.6 Style Guide

6.6.1 Introduction

This guide is intended to provide coding conventions for writing solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from python's pep8 style guide.

The goal of this guide is *not* to be the right way or the best way to write solidity code. The goal of this guide is *consistency*. A quote from python's pep8 captures this concept well.

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important. But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

6.6.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

Blank Lines

Surround top level declarations in solidity source with two blank lines.

Yes:

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

No:

```
contract A {
    ...
}
contract B {
    ...
}

contract C {
    ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes:

```
contract A {
    function spam();
    function ham();
}


contract B is A {
    function spam() {
        ...
    }

    function ham() {
        ...
    }
}
```

No:

```
contract A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }
}
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes:

```
import "owned";


contract A {
    ...
}


contract B is owned {
    ...
}
```

No:

```
contract A {
    ...
}
```

```
import "owned";


contract B is owned {
    ...
}
```

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

- constructor
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the *constant* functions last.

Yes:

```
contract A {
    function A() {
        ...
    }

    function() {
        ...
    }

    // External functions
    // ...

    // External functions that are constant
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No:

```
contract A {

    // External functions
    // ...
```

```
    // Private functions
    // ...

    // Public functions
    // ...

    function A() {
        ...
    }

    function() {
        ...
    }

    // Internal functions
    // ...
}
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single-line function declarations.

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin);
```

No:

```
function spam(uint i , Coin coin) ;
```

More than one space around an assignment or other operator to align with another:

Yes:

```
x = 1;
y = 2;
long_variable = 3;
```

No:

```
x             = 1;
y             = 2;
long_variable = 3;
```

Don't include a whitespace in the fallback function:

Yes:

```
function() {
    ...
}
```

No:

```
function () {
    ...
}
```

Control Structures

The braces denoting the body of a contract, library, functions and structs should:

- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be proceeded by a single space.

Yes:

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceeded by a single space.

Yes:

```
function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) public onlyowner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) returns (uint)
{
    return x + 1;
}

function increment(uint x) returns (uint){
    return x + 1;
}

function increment(uint x) returns (uint) {
    return x + 1;
    }

function increment(uint x) returns (uint) {
    return x + 1;}
```

The visibility modifiers for a function should come before any custom modifiers.

Yes:

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

No:

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

For long function declarations, it is recommended to drop each argument onto it's own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the same indentation level as the function declaration.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
) {
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to it's own line.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
```

```
                                        returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
    }
}
```

No:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    {
        // do something with param5
    }
}

contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
        // do something with param5
    }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgement as this guide does not try to cover all possible permutations for function declarations.

Mappings

TODO

Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

Other Recommendations

- Strings should be quoted with double-quotes instead of single-quotes.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Surround operators with a single space on either side.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statement. You should always use the same amount of whitespace on either side of an operator:

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

6.6.3 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supercede any conventions outlined in this document.

Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores`

Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

Names to Avoid

- `l` - Lowercase letter el
- `O` - Uppercase letter oh
- `I` - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

Contract and Library Names

Contracts and libraries should be named using the CapWords style.

Events

Events should be named using the CapWords style.

Function Names

Functions should use mixedCase.

Function Arguments

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

Local and State Variables

Use mixedCase.

Constants

Constants should be named with all capital letters with underscores separating words. (for example:`MAX_BLOCKS`)

Modifiers

Use mixedCase.

Avoiding Collisions

- `single_trailing_underscore_`

This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

General Recommendations

TODO

6.7 Common Patterns

6.7.1 Withdrawal from Contracts

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct `send` call, this is not recommended as it introduces a potential security risk. You may read more about this on the *Security Considerations* page.

This is an example of the withdrawal pattern in practice in a contract where the goal is to send the most money to the contract in order to become the "richest", inspired by King of the Ether.

In the following contract, if you are usurped as the richest, you will recieve the funds of the person who has gone on to become the new richest.

```
pragma solidity ^0.4.0;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    function WithdrawalContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() returns (bool) {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        if (msg.sender.send(amount)) {
            return true;
        } else {
            pendingWithdrawals[msg.sender] = amount;
            return false;
        }
    }
}
```

This is as opposed to the more intuitive sending pattern:

```
pragma solidity ^0.4.0;

contract SendContract {
    address public richest;
    uint public mostSent;

    function SendContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() returns (bool) {
        if (msg.value > mostSent) {
            // Check if call succeeds to prevent an attacker
            // from trapping the previous person's funds in
            // this contract through a callstack attack
            if (!richest.send(msg.value)) {
                throw;
            }
```

```
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }
}
```

Notice that, in this example, an attacker could trap the contract into an unusable state by causing `richest` to be the address of a contract that has a fallback function which consumes more than the 2300 gas stipend. That way, whenever `send` is called to deliver funds to the "poisoned" contract, it will cause execution to always fail because there will not be enough gas to finish the execution of the fallback function.

6.7.2 Restricting Access

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract's state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract's state by **other contracts**. That is actually the default unless you declare make your state variables `public`.

Furthermore, you can restrict who can make modifications to your contract's state or call your contract's functions and this is what this page is about.

The use of **function modifiers** makes these restrictions highly readable.

```
pragma solidity ^0.4.0;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // creating this contract.
    address public owner = msg.sender;
    uint public creationTime = now;

    // Modifiers can be used to change
    // the body of a function.
    // If this modifier is used, it will
    // prepend a check that only passes
    // if the function is called from
    // a certain address.
    modifier onlyBy(address _account)
    {
        if (msg.sender != _account)
            throw;
        // Do not forget the "_;"! It will
        // be replaced by the actual function
        // body when the modifier is used.
        _;
    }

    /// Make `_newOwner` the new owner of this
    /// contract.
    function changeOwner(address _newOwner)
        onlyBy(owner)
    {
```

```
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        if (now < _time) throw;
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // This modifier requires a certain
    // fee being associated with a function call.
    // If the caller sent too much, he or she is
    // refunded, but only after the function body.
    // This was dangerous before Solidity version 0.4.0,
    // where it was possible to skip the part after `_;`.
    modifier costs(uint _amount) {
        if (msg.value < _amount)
            throw;
        _;
        if (msg.value > _amount)
            msg.sender.send(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
        costs(200 ether)
    {
        owner = _newOwner;
        // just some example condition
        if (uint(owner) & 0 == 1)
            // This did not refund for Solidity
            // before version 0.4.0.
            return;
        // refund overpaid fees
    }
}
```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

6.7.3 State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

An example for this is a blind auction contract which starts in the stage “accepting blinded bids”, then transitions to “revealing bids” which is ended by “determine auction autcome”.

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier `atStage` ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier `timeTransitions`, which should be used for all functions.

Note: **Modifier Order Matters**. If atStage is combined with timedTransitions, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier `transitionNext` can be used to automatically go to the next stage when the function finishes.

Note: **Modifier May be Skipped**. This only applies to Solidity before version 0.4.0: Since modifiers are applied by simply replacing code and not by using a function call, the code in the transitionNext modifier can be skipped if the function itself uses return. If you want to do that, make sure to call nextStage manually from those functions. Starting with version 0.4.0, modifier code will run even if the function explicitly returns.

```
pragma solidity ^0.4.0;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }

    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        if (stage != _stage) throw;
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
        _;
    }
```

```
    // Order of the modifiers matters here!
    function bid()
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }

    function reveal()
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

6.8 Contributing

Help is always appreciated!

To get started, you can try *Building from Source* in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

In particular, we need help in the following areas:

- Improving the documentation
- Responding to questions from other users on StackExchange and the Solidity Gitter

- Fixing and responding to Solidity's GitHub issues, especially those tagged as up-for-grabs which are meant as introductory issues for external contributors.

6.8.1 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details:

- Which version of Solidity you are using
- What was the source code (if applicable)
- Which platform are you running on
- How to reproduce the issue
- What was the result of the issue
- What the expected behaviour is

Reducing the source code that caused the issue to a bare minimum is always very helpful and sometimes even clarifies a misunderstanding.

6.8.2 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch.

Additionally, if you are writing a new feature, please ensure you write appropriate Boost test cases and place them under `test/`.

However, if you are making a larger change, please consult with the Gitter channel, first.

Finally, please make sure you respect the coding standards for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

Thank you for your help!

6.8.3 Running the compiler tests

Solidity includes different types of tests. They are included in the application called `soltest`. Some of them require the `cpp-ethereum` client in testing mode.

To run `cpp-ethereum` in testing mode: `eth --test -d /tmp/testeth`.

To run the tests: `soltest -- --ipcpath /tmp/testeth/geth.ipc`.

To run a subset of tests, filters can be used: `soltest -t TestSuite/TestName -- --ipcpath /tmp/testeth/geth.ipc`, where `TestName` can be a wildcard `*`.

Alternatively, there is a testing script at `scripts/test.sh` which executes all tests.

6.9 Frequently Asked Questions

This list was originally compiled by fivedogit.

6.9.1 Basic Questions

Example contracts

There are some contract examples by fivedogit and there should be a test contract for every single feature of Solidity.

Create and publish the most basic contract possible

A quite simple contract is the greeter

Is it possible to do something on a specific block number? (e.g. publish a contract or execute a transaction)

Transactions are not guaranteed to happen on the next block or any future specific block, since it is up to the miners to include transactions and not up to the submitter of the transaction. This applies to function calls/transactions and contract creation transactions.

If you want to schedule future calls of your contract, you can use the alarm clock.

What is the transaction "payload"?

This is just the bytecode "data" sent along with the request.

Is there a decompiler available?

There is no decompiler to Solidity. This is in principle possible to some degree, but for example variable names will be lost and great effort will be necessary to make it look similar to the original source code.

Bytecode can be decompiled to opcodes, a service that is provided by several blockchain explorers.

Contracts on the blockchain should have their original source code published if they are to be used by third parties.

Create a contract that can be killed and return funds

First, a word of warning: Killing contracts sounds like a good idea, because "cleaning up" is always good, but as seen above, it does not really clean up. Furthermore, if Ether is sent to removed contracts, the Ether will be forever lost.

If you want to deactivate your contracts, it is preferable to **disable** them by changing some internal state which causes all functions to throw. This will make it impossible to use the contract and ether sent to the contract will be returned automatically.

Now to answering the question: Inside a constructor, `msg.sender` is the creator. Save it. Then `selfdestruct(creator);` to kill and return funds.

example

Note that if you `import "mortal"` at the top of your contracts and declare `contract SomeContract is mortal { ...` and compile with a compiler that already has it (which includes browser-solidity), then `kill()` is taken care of for you. Once a contract is "mortal", then you can `contractname.kill.sendTransaction({from:eth.coinbase})`, just the same as my examples.

Store Ether in a contract

The trick is to create the contract with `{from:someaddress, value: web3.toWei(3,"ether")...}`

See endowment_retriever.sol.

Use a non-constant function (req `sendTransaction`) to increment a variable in a contract

See value_incrementer.sol.

Get a contract to return its funds to you (not using `selfdestruct(...)`).

This example demonstrates how to send funds from a contract to an address.

See endowment_retriever.

Can you return an array or a `string` from a solidity function call?

Yes. See array_receiver_and_returner.sol.

What is problematic, though, is returning any variably-sized data (e.g. a variably-sized array like `uint[]`) from a fuction **called from within Solidity**. This is a limitation of the EVM and will be solved with the next protocol update.

Returning variably-sized data as part of an external transaction or call is fine.

How do you represent `double`/`float` in Solidity?

This is not yet possible.

Is it possible to in-line initialize an array like so: `string[] myarray = ["a", "b"];`

Yes. However it should be noted that this currently only works with statically sized memory arrays. You can even create an inline memory array in the return statement. Pretty cool, huh?

Example:

```
contract C {
    function f() returns (uint8[5]) {
        string[4] memory adaArr = ["This", "is", "an", "array"];
        return ([1, 2, 3, 4, 5]);
    }
}
```

Are timestamps (`now`, `block.timestamp`) reliable?

This depends on what you mean by "reliable". In general, they are supplied by miners and are therefore vulnerable.

Unless someone really messes up the blockchain or the clock on your computer, you can make the following assumptions:

You publish a transaction at a time X, this transaction contains same code that calls `now` and is included in a block whose timestamp is Y and this block is included into the canonical chain (published) at a time Z.

The value of `now` will be identical to Y and X <= Y <= Z.

Never use `now` or `block.hash` as a source of randomness, unless you know what you are doing!

Can a contract function return a `struct`?

Yes, but only in `internal` function calls.

If I return an `enum`, I only get integer values in web3.js. How to get the named values?

Enums are not supported by the ABI, they are just supported by Solidity. You have to do the mapping yourself for now, we might provide some help later.

What is the deal with `function () { ... }` inside Solidity contracts? How can a function not have a name?

This function is called "fallback function" and it is called when someone just sent Ether to the contract without providing any data or if someone messed up the types so that they tried to call a function that does not exist.

The default behaviour (if no fallback function is explicitly given) in these situations is to throw an exception.

If the contract is meant to receive Ether with simple transfers, you should implement the fallback function as

`function() payable { }`

Another use of the fallback function is to e.g. register that your contract received ether by using an event.

Attention: If you implement the fallback function take care that it uses as little gas as possible, because `send()` will only supply a limited amount.

Is it possible to pass arguments to the fallback function?

The fallback function cannot take parameters.

Under special circumstances, you can send data. If you take care that none of the other functions is invoked, you can access the data by `msg.data`.

Can state variables be initialized in-line?

Yes, this is possible for all types (even for structs). However, for arrays it should be noted that you must declare them as static memory arrays.

Examples:

```
contract C {
    struct S {
        uint a;
        uint b;
    }

    S public x = S(1, 2);
    string name = "Ada";
    string[4] memory adaArr = ["This", "is", "an", "array"];
}


contract D {
```

```
    C c = new C();
}
```

How do structs work?

See struct_and_for_loop_tester.sol.

How do for loops work?

Very similar to JavaScript. There is one point to watch out for, though:

If you use `for (var i = 0; i < a.length; i ++) { a[i] = i; }`, then the type of `i` will be inferred only from `0`, whose type is `uint8`. This means that if `a` has more than `255` elements, your loop will not terminate because `i` can only hold values up to `255`.

Better use `for (uint i = 0; i < a.length...`

See struct_and_for_loop_tester.sol.

What character set does Solidity use?

Solidity is character set agnostic concerning strings in the source code, although UTF-8 is recommended. Identifiers (variables, functions, ...) can only use ASCII.

What are some examples of basic string manipulation (`substring`, `indexOf`, `charAt`, etc)?

There are some string utility functions at stringUtils.sol which will be extended in the future. In addition, Arachnid has written solidity-stringutils.

For now, if you want to modify a string (even when you only want to know its length), you should always convert it to a `bytes` first:

```
contract C {
    string s;

    function append(byte c) {
        bytes(s).push(c);
    }

    function set(uint i, byte c) {
        bytes(s)[i] = c;
    }
}
```

Can I concatenate two strings?

You have to do it manually for now.

Why is the low-level function `.call()` less favorable than instantiating a contract with a variable (`ContractB b;`) and executing its functions (`b.doSomething();`)?

If you use actual functions, the compiler will tell you if the types or your arguments do not match, if the function does not exist or is not visible and it will do the packing of the arguments for you.

See ping.sol and pong.sol.

Is unused gas automatically refunded?

Yes and it is immediate, i.e. done as part of the transaction.

When returning a value of say `uint` type, is it possible to return an `undefined` or "null"-like value?

This is not possible, because all types use up the full value range.

You have the option to `throw` on error, which will also revert the whole transaction, which might be a good idea if you ran into an unexpected situation.

If you do not want to throw, you can return a pair:

```
contract C {
    uint[] counters;

    function getCounter(uint index)
        returns (uint counter, bool error) {
            if (index >= counters.length)
                return (0, true);
            else
                return (counters[index], false);
    }

    function checkCounter(uint index) {
        var (counter, error) = getCounter(index);
        if (error) {
            ...
        } else {
            ...
        }
    }
}
```

Are comments included with deployed contracts and do they increase deployment gas?

No, everything that is not needed for execution is removed during compilation. This includes, among others, comments, variable names and type names.

What happens if you send ether along with a function call to a contract?

It gets added to the total balance of the contract, just like when you send ether when creating a contract. You can only send ether along to a function that has the `payable` modifier, otherwise an exception is thrown.

Is it possible to get a tx receipt for a transaction executed contract-to-contract?

No, a function call from one contract to another does not create its own transaction, you have to look in the overall transaction. This is also the reason why several block explorer do not show Ether sent between contracts correctly.

What is the `memory` keyword? What does it do?

The Ethereum Virtual Machine has three areas where it can store items.

The first is "storage", where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and quite expensive to use.

The second is "memory", this is used to hold temporary values. It is erased between (external) function calls and is cheaper to use.

The third one is the stack, which is used to hold small local variables. It is almost free to use, but can only hold a limited amount of values.

For almost all types, you cannot specify where they should be stored, because they are copied everytime they are used.

The types where the so-called storage location is important are structs and arrays. If you e.g. pass such variables in function calls, their data is not copied if it can stay in memory or stay in storage. This means that you can modify their content in the called function and these modifications will still be visible in the caller.

There are defaults for the storage location depending on which type of variable it concerns:

- state variables are always in storage
- function arguments are always in memory
- local variables always reference storage

Example:

```
contract C {
    uint[] data1;
    uint[] data2;

    function appendOne() {
        append(data1);
    }

    function appendTwo() {
        append(data2);
    }

    function append(uint[] storage d) {
        d.push(1);
    }
}
```

The function `append` can work both on `data1` and `data2` and its modifications will be stored permanently. If you remove the `storage` keyword, the default is to use `memory` for function arguments. This has the effect that at the point where `append(data1)` or `append(data2)` is called, an independent copy of the state variable is created in memory and `append` operates on this copy (which does not support `.push` - but that is another issue). The modifications to this independent copy do not carry back to `data1` or `data2`.

A common mistake is to declare a local variable and assume that it will be created in memory, although it will be created in storage:

```
/// THIS CONTRACT CONTAINS AN ERROR
contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x;
        x.push(2);
        data = x;
    }
}
```

The type of the local variable `x` is `uint[] storage`, but since storage is not dynamically allocated, it has to be assigned from a state variable before it can be used. So no space in storage will be allocated for `x`, but instead it functions only as an alias for a pre-existing variable in storage.

What will happen is that the compiler interprets `x` as a storage pointer and will make it point to the storage slot `0` by default. This has the effect that `someVariable` (which resides at storage slot `0`) is modified by `x.push(2)`.

The correct way to do this is the following:

```
contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x = data;
        x.push(2);
    }
}
```

What is the difference between `bytes` and `byte[]`?

`bytes` is usually more efficient: When used as arguments to functions (i.e. in CALLDATA) or in memory, every single element of a `byte[]` is padded to 32 bytes which wastes 31 bytes per element.

Is it possible to send a value while calling an overloaded function?

It's a known missing feature. https://www.pivotaltracker.com/story/show/92020468 as part of https://www.pivotaltracker.com/n/projects/1189488

Best solution currently see is to introduce a special case for gas and value and just re-check whether they are present at the point of overload resolution.

6.9.2 Advanced Questions

How do you get a random number in a contract? (Implement a self-returning gambling contract.)

Getting randomness right is often the crucial part in a crypto project and most failures result from bad random number generators.

If you do not want it to be safe, you build something similar to the coin flipper but otherwise, rather use a contract that supplies randomness, like the RANDAO.

Get return value from non-constant function from another contract

The key point is that the calling contract needs to know about the function it intends to call.

See ping.sol and pong.sol.

Get contract to do something when it is first mined

Use the constructor. Anything inside it will be executed when the contract is first mined.

See replicator.sol.

How do you create 2-dimensional arrays?

See 2D_array.sol.

Note that filling a 10x10 square of `uint8` + contract creation took more than `800,000` gas at the time of this writing. 17x17 took `2,000,000` gas. With the limit at 3.14 million... well, there's a pretty low ceiling for what you can create right now.

Note that merely "creating" the array is free, the costs are in filling it.

Note2: Optimizing storage access can pull the gas costs down considerably, because 32 `uint8` values can be stored in a single slot. The problem is that these optimizations currently do not work across loops and also have a problem with bounds checking. You might get much better results in the future, though.

What does `p.recipient.call.value(p.amount)(p.data)` do?

Every external function call in Solidity can be modified in two ways:

1. You can add Ether together with the call
2. You can limit the amount of gas available to the call

This is done by "calling a function on the function":

`f.gas(2).value(20)()` calls the modified function `f` and thereby sending 20 Wei and limiting the gas to 2 (so this function call will most likely go out of gas and return your 20 Wei).

In the above example, the low-level function `call` is used to invoke another contract with `p.data` as payload and `p.amount` Wei is sent with that call.

What happens to a `struct`'s mapping when copying over a `struct`?

This is a very interesting question. Suppose that we have a contract field set up like such:

```
struct user {
    mapping(string => address) usedContracts;
}

function somefunction {
   user user1;
   user1.usedContracts["Hello"] = "World";
   user user2 = user1;
}
```

In this case, the mapping of the struct being copied over into the userList is ignored as there is no "list of mapped keys". Therefore it is not possible to find out which values should be copied over.

How do I initialize a contract with only a specific amount of wei?

Currently the approach is a little ugly, but there is little that can be done to improve it. In the case of a `contract A` calling a new instance of `contract B`, parentheses have to be used around `new B` because `B.value` would refer to a member of `B` called `value`. You will need to make sure that you have both contracts aware of each other's presence and that `contract B` has a `payable` constructor. In this example:

```
contract B {
    function B() payable {}
}


contract A {
    address child;

    function test() {
        child = (new B).value(10)(); //construct a new B with 10 wei
    }
}
```

Can a contract function accept a two-dimensional array?

This is not yet implemented for external calls and dynamic arrays - you can only use one level of dynamic arrays.

What is the relationship between `bytes32` and `string`? Why is it that `bytes32 somevar = "stringliteral";` works and what does the saved 32-byte hex value mean?

The type `bytes32` can hold 32 (raw) bytes. In the assignment `bytes32 samevar = "stringliteral";`, the string literal is interpreted in its raw byte form and if you inspect `somevar` and see a 32-byte hex value, this is just `"stringliteral"` in hex.

The type `bytes` is similar, only that it can change its length.

Finally, `string` is basically identical to `bytes` only that it is assumed to hold the UTF-8 encoding of a real string. Since `string` stores the data in UTF-8 encoding it is quite expensive to compute the number of characters in the string (the encoding of some characters takes more than a single byte). Because of that, `string s; s.length` is not yet supported and not even index access `s[2]`. But if you want to access the low-level byte encoding of the string, you can use `bytes(s).length` and `bytes(s)[2]` which will result in the number of bytes in the UTF-8 encoding of the string (not the number of characters) and the second byte (not character) of the UTF-8 encoded string, respectively.

Can a contract pass an array (static size) or string or `bytes` (dynamic size) to another contract?

Sure. Take care that if you cross the memory / storage boundary, independent copies will be created:

```
contract C {
    uint[20] x;

    function f() {
        g(x);
        h(x);
    }

    function g(uint[20] y) {
        y[2] = 3;
```

```
    }

    function h(uint[20] storage y) {
        y[3] = 4;
    }
}
```

The call to `g(x)` will not have an effect on `x` because it needs to create an independent copy of the storage value in memory (the default storage location is memory). On the other hand, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

Sometimes, when I try to change the length of an array with ex: `arrayname.length = 7;` I get a compiler error `Value must be an lvalue`. Why?

You can resize a dynamic array in storage (i.e. an array declared at the contract level) with `arrayname.length = <some new length>;`. If you get the "lvalue" error, you are probably doing one of two things wrong.

1. You might be trying to resize an array in "memory", or
2. You might be trying to resize a non-dynamic array.

```
int8[] memory memArr;         // Case 1
memArr.length++;              // illegal
int8[5] storageArr;           // Case 2
somearray.length++;           // legal
int8[5] storage storageArr2; // Explicit case 2
somearray2.length++;          // legal
```

Important note: In Solidity, array dimensions are declared backwards from the way you might be used to declaring them in C or Java, but they are access as in C or Java.

For example, `int8[][5] somearray;` are 5 dynamic `int8` arrays.

The reason for this is that `T[5]` is always an array of 5 `T`'s, no matter whether `T` itself is an array or not (this is not the case in C or Java).

Is it possible to return an array of strings (`string[]`) from a Solidity function?

Not yet, as this requires two levels of dynamic arrays (`string` is a dynamic array itself).

If you issue a call for an array, it is possible to retrieve the whole array? Or must you write a helper function for that?

The automatic accessor function for a public state variable of array type only returns individual elements. If you want to return the complete array, you have to manually write a function to do that.

What could have happened if an account has storage value(s) but no code? Example: http://test.ether.camp/account/5f740b3a43fbb99724ce93a879805f4dc89178b5

The last thing a constructor does is returning the code of the contract. The gas costs for this depend on the length of the code and it might be that the supplied gas is not enough. This situation is the only one where an "out of gas" exception does not revert changes to the state, i.e. in this case the initialisation of the state variables.

https://github.com/ethereum/wiki/wiki/Subtleties

After a successful CREATE operation's sub-execution, if the operation returns x, 5 * len(x) gas is subtracted from the remaining gas before the contract is created. If the remaining gas is less than 5 * len(x), then no gas is subtracted, the code of the created contract becomes the empty string, but this is not treated as an exceptional condition - no reverts happen.

How do I use `.send()`?

If you want to send 20 Ether from a contract to the address `x`, you use `x.send(20 ether);`. Here, `x` can be a plain address or a contract. If the contract already explicitly defines a function `send` (and thus overwrites the special function), you can use `address(x).send(20 ether);`.

Note that the call to `send` may fail in certain conditions, such as if you have insufficient funds, so you should always check the return value. `send` returns `true` if the send was successful and `false` otherwise.

What does the following strange check do in the Custom Token contract?

```
if (balanceOf[_to] + _value < balanceOf[_to])
    throw;
```

Integers in Solidity (and most other machine-related programming languages) are restricted to a certain range. For `uint256`, this is `0` up to `2**256 - 1`. If the result of some operation on those numbers does not fit inside this range, it is truncated. These truncations can have serious consequences, so code like the one above is necessary to avoid certain attacks.

More Questions?

If you have more questions or your question is not answered here, please talk to us on gitter or file an issue.

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y