Solidity Documentation

Version 0.6.8

Ethereum

juin 03, 2020

Les bases

Solidity est un langage haut-niveau, orienté objet dédié à l'implémentation de smart contracts. Les smart contracts (littéralement contrats intelligents) sont des programes qui régissent le comportement de comptes dans l'état d'Ethereum.

Solidity a été influencé par C++, Python et JavaScript et est conçu pour cibler la machine virtuelle Ethereum (EVM).

Solidity est statiquement typé, supporte l'héritage, les librairies et les bibliothèques, ainsi que les types complexes définis par l'utilisateur parmi d'autres caractéristiques.

Avec Solidity, vous pouvez créer des contrats pour des usages tels que le vote, le crowdfunding, les enchères à l'aveugle, et portefeuilles multi-signature.

When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes as well as new features and bug fixes are introduced regularly. We currently use a 0.x version number to indicate this fast pace of change.

Avertissement : Solidity recently released the 0.6.x version that introduced a lot of breaking changes. Make sure you read *the full list*.

CHAPITRE 1

Documentation du langage

If you are new to the concept of smart contracts we recommend you start with *an example smart contract* written in Solidity. When you are ready for more detail, we recommend you read the *« Solidity by Example »* and « Language Description » sections to learn the core concepts of the language.

For further reading, try *the basics of blockchains* and details of the *Ethereum Virtual Machine*.

Indication : Rappelez-vous que vous pouvez toujours essayer les contrats dans votre navigateur ! Remix is a web browser based IDE that allows you to write Solidity smart contracts, then deploy and run the smart contracts. It can take a while to load, so please be patient.

Avertissement : As humans write software, it can have bugs. You should follow established software development best-practices when writing your smart contracts, this includes code review, testing, audits, and correctness proofs. Smart contract users are sometimes more confident with code than their authors, and blockchains and smart contracts have their own unique issues to watch out for, so before working on production code, make sure you read the *Security Considerations* section.

CHAPITRE 2

Traductions

Cette documentation est traduite en plusieurs langues par des bénévoles de la communauté avec divers degrés d'exhaustivité et d'actualité. La version anglaise reste la référence.

- French (en cours)
- Italian (en cours)
- Japanese
- Korean (en cours)
- Russian (plutôt dépassée)
- Simplified Chinese (en cours)
- Spanish
- Turkish (partielle)

CHAPITRE 3

Sommaire

Keyword Index, Search Page

3.1 Introduction aux Smart Contracts

3.1.1 Un Smart Contract simple

Commençons par un exemple de base qui fixe la valeur d'une variable et l'expose pour l'accès par d'autres contrats. C'est très bien si vous ne comprenez pas tout maintenant, nous entrerons plus en détail plus tard.

Exemple : Stockage

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract SimpleStorage {
    uint storedData;

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

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

The first line tells you that the source code is licensed under the GPL version 3.0. Machine-readable license specifiers are important in a setting where publishing the source code is the default.

La ligne suivante indique simplement que le code source est écrit pour Solidity version 0.4.0 ou tout ce qui est plus récent qui ne casse pas la fonctionnalité (jusqu'à la version 0.6.0, mais non comprise). Il s'agit de s'assurer que

le contrat n'est pas compilable avec une nouvelle version du compilateur (de rupture), où il pourrait se comporter différemment. Les pré-cités pragmas sont des instructions courantes pour les compilateurs sur la façon de traiter le code source (par exemple pragma once).

Un contrat au sens de Solidity est un ensemble de code (ses *fonctions*) et les données (son *état*) qui résident à une adresse spécifique sur la blockchain Ethereum. La ligne `uint storedData;` déclare une variable d'état appelée `storedData`` de type `uint` (*u*nsigned *int*eger de *256* bits). Vous pouvez le considérer comme une case mémoire dans une base de données qui peut être interrogée et modifiée en appelant les fonctions du code qui gèrent la base de données. Dans le cas d'Ethereum, c'est toujours le contrat propriétaire. Et dans ce cas, les fonctions `set` et `get` peuvent être utilisées pour modifier ou récupérer la valeur de la variable.

Pour accéder à une variable d'état, vous n'avez pas besoin du préfixe `this.` d'autres langues.

Ce contrat ne fait pas encore grand-chose en dehors de (en raison de l'infrastructure construite par Ethereum) permettre à n'importe qui de stocker un numéro unique qui est accessible par n'importe qui dans le monde sans un moyen (faisable) pour vous empêcher de publier ce numéro. Bien sûr, n'importe qui peut simplement appeler `set` à nouveau avec une valeur différente. et écraser votre numéro, mais le numéro sera toujours stocké dans l'historique de la blockchain. Plus tard, nous verrons comment vous pouvez imposer des restrictions d'accès pour que vous seul puissiez modifier le numéro.

Note : Tous les identifiants (noms de contrat, noms de fonctions et noms de variables) sont limités au jeu de caractères ASCII. Il est possible de stocker des données encodées en UTF-8 dans des variables de type string.

Avertissement : Soyez prudent lorsque vous utilisez du texte Unicode, car des caractères d'apparence similaire (ou même identique) peuvent avoir des codages unicode différents et seront donc codés sous la forme d'un tableau d'octets différent.

Exemple de sous-monnaie

Le contrat suivant mettra en œuvre la forme la plus simple d'un contrat de cryptomonnaie. Ce contrat autorise son créateur à générer des pièces à partir de rien (des schéma d'émission différents sont possibles). De plus, n'importe qui peut s'envoyer des pièces sans avoir besoin de s'enregistrer avec un nom d'utilisateur et un mot de passe - tout ce dont vous avez besoin est une paire de clés Ethereum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract Coin {
    // Le mot-clé "public" rend ces variables
    // facilement accessible de l'exterieur.
    address public minter;
    mapping (address => uint) public balances;

    // Les Events authowisent les clients légers à réagir
    // aux changements efficacement.
    event Sent(address from, address to, uint amount);

    // C'est le constructor, code qui n'est exécuté
    // qu'à la création du contrat.
    constructor() public {
        minter = msg.sender;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Ce contrat introduit quelques nouveaux concepts, passons-les en revue un à un.

La ligne `address public minter;` déclare une variable d'état de type *address* qui est accessible au public. Le type `adress` est une valeur de 160 bits qui ne permet aucune opération arithmétique. Il convient pour le stockage des adresses de contrats ou de paires de clés appartenant à des **'comptes externes<accounts>'_** .

Le mot-clé « public » génère automatiquement une fonction qui permet d'accéder à la valeur courante de la variable d'état de l'extérieur du contrat. Sans ce mot-clé, les autres contrats n'ont aucun moyen d'accéder à la variable. Le code de la fonction générée par le compilateur est à peu près équivalent à ce qui suit (ignorez `external'' et ``view` pour l'instant) :

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

Bien sûr, l'ajout d'une fonction exactement comme celle-là ne fonctionnera pas parce que nous aurions une fonction et une variable d'état avec le même nom, mais vous avez l'idée - le compilateur réalisera cela pour vous.

La ligne suivante, `mapping (" adress => uint ") public balances;` crée également une variable d'état publique, mais c'est un type de données plus complexe. Le type *mapping* fait correspondre les adresses aux *entiers non signés*.

Les mappings peuvent être vus comme des tables de hachage qui sont virtuellement initialisées de sorte que toutes les clés possibles existent dès le début et sont mappées à un fichier dont la représentation octale n'est que de zéros. Cette analogie ne va pas trop loin, car il n'est pas non plus possible d'obtenir une liste de toutes les clés d'un mapping, ni une liste de toutes les valeurs. Il faut donc garder à l'esprit (ou bien mieux, gardez une liste ou utilisez un type de données plus avancé) ce que vous avez ajouté à la cartographie ou l'utiliser dans un contexte où cela n'est pas nécessaire. La fonction getter créé par le mot-clé `public` est un peu plus complexe dans ce cas. Ça ressemble grossièrement à ça :

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

Comme vous pouvez le voir, vous pouvez utiliser cette fonction pour interroger facilement le solde d'un seul compte.

La ligne `event Sent(address from, address to, uint amount);` déclare un bien-nommé *« event »* qui est émis dans la dernière ligne de la fonction `send`. Les interfaces utilisateur (ainsi que les applications serveur bien sûr) peuvent écouter les événements qui sont émis sur la blockchain sans trop de frais. Dès qu'elle est émise, l'auditeur reçoit également le message des arguments « from », « to » et « amount », ce qui facilite le suivi des transactions.

Pour écouter cet événement, vous devriez utiliser le code JavaScript suivant (qui suppose que ‘‘Coin‘ est un objet de contrat créé via web3.js ou un module similaire) :

```
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));
    }
})
```

Le *constructor* est une fonction spéciale qui est exécutée pendant la création du contrat et ne peut pas être appelée ultérieurement. Il stocke de façon permanente l'adresse de la personne qui crée le contrat : `msg` (avec `tx` et `block`) est une variable globale spéciale qui contient certaines propriétés qui permettent d'accéder à la blockchain. `msg.sender` est toujours l'adresse d'où vient l'appel de la fonction courante (externe).

Enfin, les fonctions qui finiront avec le contrat et qui peuvent être appelées par les utilisateurs et les contrats sont « mint » et « send ». Si « mint » est appelé par quelqu'un d'autre que le compte qui a créé le contrat, rien ne se passera. Ceci est assuré par la fonction spéciale *require* qui fait que tous les changements sont annulés si son argument est évalué à faux. Le deuxième appel à `require` permet de s'assurer qu'il n'y aura pas trop de pièces, ce qui pourrait causer des erreurs de débordement de buffer plus tard.

D'un autre côté, `send` peut être utilisé par n'importe qui (qui a déjà certaines de ces pièces) pour envoyer des pièces à n'importe qui d'autre. Si vous n'avez pas assez de pièces à envoyer, l'appel `require` échouera et fournira également à l'utilisateur un message d'erreur approprié.

Note : Si vous utilisez ce contrat pour envoyer des pièces à une adresse, vous ne verrez rien lorsque vous regarderez cette adresse sur un explorateur de chaîne de blocs, parce que le fait que vous avez envoyé des pièces et les soldes modifiés sont seulement stockés dans le stockage de données de ce contrat de pièces particulier. Par l'utilisation d'événements, il est relativement facile de créer un « explorateur de chaîne » qui suit les transactions et les soldes de votre nouvelle pièce, mais vous devez inspecter l'adresse du contrat de pièces et non les adresses des propriétaires des pièces.

3.1.2 Blockchain Basics

Les blockchains en tant que concept ne sont pas trop difficiles à comprendre pour les programmeurs. La raison en est que la plupart des complications (mining, hashing, elliptic-curve cryptography, réseaux pair-à-pair, etc.) sont juste là pour fournir un certain nombre de fonctionnalités et de promesses pour la plate-forme. Une fois que vous prenez ces fonctions pour aquises, vous n'avez pas à vous soucier de la technologie sous-jacente - ou devez-vous savoir comment fonctionne le cloud AWS d'Amazon en interne afin de l'utiliser ?

Transactions

Une blockchain est une base de données transactionnelle partagée à l'échelle mondiale. Cela signifie que tout le monde peut lire les entrées de la base de données simplement en participant au réseau. Si vous voulez modifier quelque chose dans la base de données, vous devez créer une transaction qui doit être acceptée par tous les autres. Le mot transaction implique que la modification que vous voulez effectuer (en supposant que vous voulez modifier deux valeurs en même temps) n'est pas effectuée du tout ou est complètement appliquée. De plus, pendant que votre transaction est appliquée à la base de données, aucune autre transaction ne peut la modifier.

Par exemple, imaginez un tableau qui énumère les soldes de tous les comptes dans une devise électronique. Si un transfert d'un compte à un autre est demandé, la nature transactionnelle de la base de données garantit que si le montant est soustrait d'un compte, il est toujours ajouté à l'autre compte. Si, pour quelque raison que ce soit, il n'est pas possible d'ajouter le montant au compte cible, le compte source n'est pas non plus modifié.

De plus, une transaction est toujours signée cryptographiquement par l'expéditeur (créateur). Il est donc facile de garder l'accès à des modifications spécifiques de la base de données. Dans l'exemple de la monnaie électronique, un simple contrôle permet de s'assurer que seule la personne qui détient les clés du compte peut transférer de l'argent à partir de celui-ci.

Blocs

Un obstacle majeur à surmonter est ce que l'on appelle (en termes Bitcoin) une » attaque de double dépense » : Que se passe-t-il si deux transactions existent dans le réseau et que toutes deux veulent vider un compte ? Une seule des transactions peut être valide, généralement celle qui est acceptée en premier. Le problème est que « premier » n'est pas un terme objectif dans un réseau pair-à-pair.

La réponse abstraite à cette question est que vous n'avez pas à vous en soucier. Un ordre des transactions accepté dans le monde entier sera sélectionné pour vous, résolvant ainsi le conflit. Les transactions seront regroupées dans ce que l'on appelle un « bloc », puis elles seront exécutées et réparties entre tous les nœuds participants. Si deux transactions se contredisent, celle qui finit deuxième sera rejetée et ne fera pas partie du bloc.

Ces blocs forment une séquence linéaire dans le temps et c'est de là que vient le mot « blockchain ». Des blocs sont ajoutés à la chaîne à des intervalles assez réguliers - pour Ethereum, c'est à peu près toutes les 17 secondes.

Dans le cadre du mécanisme de sélection d'ordre (qu'on appelle « mining »), il peut arriver que des blocs soient retournés de temps à autre, mais seulement au « sommet » de la chaîne. Plus il y a de blocs ajoutés au-dessus d'un bloc particulier, moins il y a de chances que ce bloc soit retourné. Il se peut donc que vos transactions soient annulées et même supprimées de la blockchain, mais plus vous attendez, moins il est probable qu'elles le soient.

Note : Il n'est pas garanti que les transactions seront incluses dans le bloc suivant ou dans tout bloc futur spécifique, puisque ce n'est pas à l'auteur d'une transaction, mais aux mineurs de déterminer dans quel bloc la transaction est incluse.

Si vous voulez programmer des appels futurs de votre contrat, vous pouvez utiliser le service alarm clock ou un service oracle similaire.

3.1.3 La Machine Virtuelle Ethereum

Définition

La Machine Virtuelle Ethereum ou EVM est l'environnement d'exécution des contrats intelligents dans Ethereum. Il n'est pas seulement cloisonné, il est aussi complètement isolé, ce qui signifie que le code fonctionnant à l'intérieur de l'EVM n'a pas accès au réseau, au système de fichiers ou à d'autres processus. Les Smart Contracts ont même un accès limité à d'autres Smart Contracts.

Comptes De plus, chaque compte a une **balance** en Ether (dans « Wei » pour être exact, *1 ether* est *10**18 wei*) qui peut être modifié en envoyant des transactions qui incluent des Ether.

Transactions

Une transaction est un message envoyé d'un compte à un autre (qui peut être identique ou vide, voir ci-dessous). Il peut inclure des données binaires (ce qu'on appelle charge utile ou « payload ») et de l'éther.

Si le compte cible contient du code, ce code est exécuté et le payload est fourni comme données d'entrée.

Si le compte cible n'est pas défini (la transaction n'a pas de destinataire ou le destinataire est défini sur `null`), la transaction crée un **nouveau contrat**. Comme nous l'avons déjà mentionné, l'adresse de ce contrat n'est pas l'adresse zéro, mais une adresse dérivée de l'adresse de l'expéditeur et de son nombre de transactions envoyées (le « nonce »). Le payload d'une telle transaction de création de contrat est considérée comme étant du bytecode EVM et exécuté. Les données de sortie de cette exécution sont stockées en permanence comme code du contrat. Cela signifie que pour créer un contrat, vous n'envoyez pas le code réel du contrat, mais en fait un code qui retourne ce code lorsqu'il est exécuté.

Note : Pendant la création d'un contrat, son code est toujours vide. Pour cette raison, vous ne devez pas rappeler le contrat en cours de construction tant que son constructeur n'a pas terminé son exécution.

Gas

Lors de la création, chaque transaction est facturée une certaine quantité de **gas**, dont le but est de limiter la quantité de travail nécessaire à l'exécution de la transaction et de payer pour cette exécution en même temps. Pendant que l'EVM exécute la commande le gaz est progressivement épuisé selon des règles spécifiques.

Le **gas price** (prix du gas) est une valeur fixée par le créateur de la transaction, qui doit payer `gas_price * gas` à l'avance à partir du compte émetteur. S'il reste du gaz après l'exécution, il est remboursé au créateur de la même manière.

Si le gaz est épuisé à n'importe quel moment (c'est-à-dire qu'il serait négatif), une exception « à court de gas » est déclenchée, qui annule toutes les modifications apportées à l'état dans la trame d'appel en cours.

Storage, Memory et la Stack

La machine virtuelle Ethereum dispose de trois zones où elle peut stocker les données, stockage (« storage »), la mémoire (« memory ») et la pile (« stack »), qui sont expliquées dans les paragraphes suivants.

Chaque compte possède une zone de données appelée **storage**, qui est persistante entre les appels de fonction et les transactions. Storage est un stockage de valeur clé qui mappe les mots de 256 bits en 256 bits. Il n'est pas possible d'énumérer storage à partir d'un contrat et il est comparativement coûteux à lire, et encore plus à modifier le storage. Un contrat ne peut ni lire ni écrire dans un storage autre que le sien.

La deuxième zone de données est appelée **memory**, dont un contrat obtient une instance fraîchement rapprochée pour chaque appel de message. La mémoire est linéaire et peut être adressée au niveau de l'octet, mais les lectures sont limitées à une largeur de 256 bits, tandis que les écritures peuvent être de 8 bits ou de 256 bits. La mémoire est augmentée d'un mot (256 bits), lors de l'accès (en lecture ou en écriture) à un mot de mémoire qui n'a pas été touché auparavant (c.-à-d. tout décalage dans un mot). Au moment de l'agrandissement, le coût en gaz doit être payé. La mémoire est d'autant plus coûteuse qu'elle s'agrandit (le coût grandit de façon quadratique).

L'EVM n'est pas une machine à registre mais une machine à pile, donc tous les calculs sont effectués sur une zone de données appelée la **stack**. Elle a une taille maximale de 1024 éléments et contient des mots de 256 bits. L'accès à la stack est limitée à l'extrémité supérieure de la façon suivante : Il est possible de copier l'un des 16 éléments les plus hauts au sommet de la stack ou d'inverser l'élément le plus en haut avec l'un des 16 éléments en dessous. Toutes les autres opérations prennent les deux éléments les plus hauts (ou un, ou plus, selon l'opération) de la stack et poussent le résultat sur la stack. Bien sûr, il est possible de déplacer les éléments de la pile vers le stockage ou la mémoire afin d'obtenir un accès plus profond à la stack, mais il n'est pas possible d'accéder à des éléments arbitraires plus profondément dans la stack sans d'abord en enlever le haut.

Jeu d'Instructions

Le jeu d'instructions de l'EVM est maintenu au minimum afin d'éviter des impl'ementations incorrectes ou incohérentes qui pourraient causer des problèmes de consensus. Toutes les instructions fonctionnent sur le type de données de base, les mots de 256 bits ou sur des tranches de mémoire (ou d'autres tableaux d'octets). Les opérations arithmétiques, binaires, logiques et de comparaison habituelles sont présentes. Des sauts conditionnels et inconditionnels sont possibles. En outre, les contrats peuvent accéder aux propriétés pertinentes du bloc actuel comme son numéro et son horodatage.

Pour une liste complète, veuillez consulter la liste :ref :‘ liste des opcodes <opcodes>‘ dans la documentation de l'insertion de langage assembleur.

Les Message Calls

Les contrats peuvent appeler d'autres contrats ou envoyer des Ether sur des comptes non contractuels par le biais d'appels de messages (« message calls »). Les Message Calls sont similaires aux transactions, en ce sens qu'ils ont une source, une cible, une charge utile de données, d'éventuels Ether, le gas et le retour. En fait, chaque transaction consiste en un message call de niveau supérieur qui, à son tour, peut créer d'autres message calls.

Un contrat peut décider de la quantité de **gas** qu'il doit envoyer avec l'appel de message interne et de la quantité qu'il souhaite conserver. Si une exception fin de gas se produit dans l'appel interne (ou toute autre exception), elle sera signalée par une valeur d'erreur placée sur la stack. Dans ce cas, seul le gas envoyé avec l'appel est épuisé. Dans Solidity, le contrat appelant provoque une exception manuelle par défaut dans de telles situations, de sorte que les exceptions « remontent en surface » de la pile d'appels.

Comme déjà dit, le contrat appelé (qui peut être le même que celui de l'appelant) recevra une instance de mémoire fraîchement effacée et aura accès à la charge utile de l'appel - qui sera fournie dans une zone séparée appelée **calldata**. Une fois l'exécution terminée, il peut renvoyer des données qui seront stockées à un emplacement de la mémoire de l'appelant pré-alloué par ce dernier. Tous ces appels sont entièrement synchrones.

Les appels sont **limités** à une profondeur de 1024, ce qui signifie que pour les opérations plus complexes, les boucles doivent être préférées aux appels récursifs. De plus, seul 63/64ème du gaz peut être transféré lors d'un appel de message, ce qui entraîne une limite de profondeur d'un peu moins de 1000 en pratique.

Delegatecall / Callcode et Libraries

Il existe une variante spéciale d'un message call, appelée **delegatecall**, qui est identique à un appel de message sauf que le code à l'adresse cible est exécuté dans le cadre du contrat d'appel et que `msg.sender` et `msg.value` ne changent pas leurs valeurs.

Cela signifie qu'un contrat peut charger dynamiquement du code à partir d'une adresse différente lors de l'exécution. Le stockage, l'adresse actuelle et le solde se réfèrent toujours au contrat d'appel, seul le code est repris de l'adresse appelée.

Cela permet d'implémenter la fonctionnalité « bibliothèque » dans Solidity : Code de bibliothèque réutilisable qui peut être appliqué au stockage d'un contrat, par exemple pour implémenter une structure de données complexe.

Logs / Journalisation

Il est possible de stocker les données dans une structure de données spécialement indexée qui s'étend jusqu'au niveau du bloc. Cette fonction appelée **logs** (journalisation) est utilisé par Solidity pour implémenter les *events*. Les contrats ne peuvent pas accéder aux données du journal une fois qu'elles ont été créées, mais ils peut être accédé efficacement de l'extérieur de la chaîne de blocs. Puisqu'une partie des données du journal est stockée dans des bloom filters, il est possible de rechercher ces données de manière efficace et cryptographique de manière sécurisée, afin que les pairs du réseau qui ne téléchargent pas la totalité de la blockchain (appelés « clients légers ») peuvent encore trouver ces logs.

Création

Les contrats peuvent même créer d'autres contrats à l'aide d'un opcode spécial (càd qu'ils n'appellent pas simplement l'adresse zéro comme le ferait une transaction). La seule différence entre ces **appels de création** et des appels de message normaux est que les données de charge utile sont exécutées, le résultat stocké sous forme de code et l'appelant / créateur reçoit l'adresse du nouveau contrat sur la stack.

Désactivation et Auto-Destruction

La seule façon de supprimer du code de la blockchain est lorsqu'un contrat à cette adresse exécute l'opération d'auto-destruction `selfdestruct`. L'Ether restant stocké à cette adresse est envoyé à une cible désignée, puis le stockage et le code sont retirés de l'état. Supprimer le contrat en théorie semble être une bonne idée, mais c'est potentiellement dangereux, comme en cas d'envoi d'éther à des contrats supprimés, où l'éther est perdu à jamais.

Avertissement : Même si un contrat est supprimé par `selfdestruct`, il fait toujours partie de l'historique de la blockchain et probablement conservé par la plupart des nœuds Ethereum. L'utilisation de l'autodestruction n'est donc pas la même chose que la suppression de données d'un disque dur.

Note : Même si le code d'un contrat ne contient pas d'appel à `selfdestruct`, il peut toujours effectuer cette opération en utilisant le `delegate code` ou le `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

3.2 Installer le Compilateur Solidity

3.2.1 Versionnage

Les versions de Solidity suivent un versionnage sémantique et en plus des versions stables, des versions de développement **nightly development builds** sont également disponibles. Les versions nightly ne sont pas garanties de fonctionner et malgré tous les efforts, elles peuvent contenir des changements non documentés et/ou cassés. Nous vous recommandons d'utiliser la dernière version. Les installateurs de paquets ci-dessous utilisent la dernière version.

3.2.2 Remix

Nous recommandons Remix pour les petits contrats et pour l'apprentissage rapide de Solidity.

Accédez à Remix en ligne, vous n'avez rien à installer. Si vous voulez l'utiliser sans connexion à Internet, allez à https://github.com/ethereum/remix-live/tree/gh-pages et téléchargez le fichier `.zip` tel qu'expliqué sur cette page. Remix is also a convenient option for testing nightly builds without installing multiple Solidity versions.

D'autres options sur cette page détaillent l'installation du compilateur Solidity en ligne de commande sur votre ordinateur. Choisissez un compilateur de ligne de commande si vous travaillez sur un contrat plus important ou si vous avez besoin de plus d'options de compilation.

3.2.3 npm / Node.js

Utilisez *npm" pour un moyen pratique et portable d'installer 'solcjs", un compilateur Solidity. Le programme 'solcjs* a moins de fonctionnalités que le compilateur décrit plus bas sur cette page. La documentation du *Using the Commandline Compiler* suppose que vous utilisez le compilateur complet, *solc*. L'utilisation de *solcjs* est documentée dans son propre dépot.

Note : Le projet solc-js est dérivé du projet C++ *solc* en utilisant Emscripten, ce qui signifie que les deux utilisent le même code source du compilateur. *solc-js* peut être utilisé directement dans les projets JavaScript (comme Remix). Veuillez vous référer au dépôt solc-js pour les instructions.

```
npm install -g solc
```

Note : L'exécutable en ligne de commande est nommé *solcjs*.

Les options de la ligne de commande de *solcjs* ne sont pas compatibles avec *solc" et les outils (tels que 'geth") attendant le comportement de 'solc* ne fonctionneront pas avec *solcjs*.

3.2.4 Docker

Nous fournissons des images dockers à jour pour le compilateur via l'image `solc` distribué par l'organisation `ethereum`. Le label `stable` contient les versions publiées tandis que le label `nightly` contient des changements potentiellement instables dans la branche develop. Docker images of Solidity builds are available using the `solc` image from the `ethereum` organisation. Use the `stable` tag for the latest released version, and `nightly` for potentially unstable changes in the develop branch.

The Docker image runs the compiler executable, so you can pass all compiler arguments to it. For example, the command below pulls the stable version of the `solc` image (if you do not have it already), and runs it in a new container, passing the `--help` argument.

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

You can also specify release build versions in the tag, for example, for the 0.5.4 release.

```
docker run ethereum/solc:0.5.4 --help
```

To use the Docker image to compile Solidity files on the host machine mount a local folder for input and output, and specify the contract to compile. For example.

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --
↪bin /sources/Contract.sol
```

You can also use the standard JSON interface (which is recommended when using the compiler with tooling). When using this interface it is not necessary to mount any directories.

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

Paquets binaires Les binaires de Solidity sont disponibles à solidity/releases.

Nous avons également des PPAs for Ubuntu, vous pouvez obtenir la dernière version via la commande :

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

La version nigthly peut s'installer avec la commande :

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

Nous publions également un package snap, installable dans toutes les distributionss linux supportées. Pour installer la dernière evrsion stable de solc :

```
sudo snap install solc
```

Si vous voulez aider aux tests en utilisant la dernière version de développement, avec les changements l ;es plus récents, merci d'utiliser :

```
sudo snap install solc --edge
```

Note : The `solc` snap uses strict confinement. This is the most secure mode for snap packages but it comes with limitations, like accessing only the files in your `/home` and `/media` directories. For more information, go to Demystifying Snap Confinement.

Arch Linux a aussi des paquets, bien que limités à la dernière version de développement :

```
pacman -S solidity
```

Nous distribuons également le compilateur Solidity via homebrew dans une version compilée à partir des sources. Les « bottles » pré-compilées ne sont pas encore supportées pour l'instant.

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

To install the most recent 0.4.x / 0.5.x version of Solidity you can also use `brew install solidity@4` and `brew install solidity@5`, respectively.

Si vous avec besoin d'une version spécifique, vous pouvez exécuter la formule homebrew correspondante disponible sur GitHub.

Regarder commits de solidity.rb sur Github.

Suivez l'historique des liens jusqu'à avoir un lien de ficher brut (« raw ») d'un commit spécifique de `solidity.rb`.

Installez-le via `brew` :

```
brew unlink solidity
# Install 0.4.8
brew install https://raw.githubusercontent.com/ethereum/homebrew-ethereum/
↪77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo Linux has an Ethereum overlay that contains a solidity package. After the overlay is setup, `solc` can be installed in x86_64 architectures by :

```
emerge dev-lang/solidity
```

3.2.5 Compilation à partir des sources

Prérequis - Linux

Vous aurez besoin des dépendances suivantes pour toutes compilations de Solidity :

Software	Notes
CMake (version 3.9+)	Cross-platform build file generator.
Boost (version 1.65+)	C++ libraries.
Git	Command-line tool for retrieving source code.
z3 (version 4.6+, Optional)	For use with SMT checker.
cvc4 (Optional)	For use with SMT checker.

Note : Solidity versions prior to 0.5.10 can fail to correctly link against Boost versions 1.70+. A possible workaround is to temporarily rename `<Boost install path>/lib/cmake/Boost-1.70.0` prior to running the cmake command to configure solidity.

Starting from 0.5.10 linking against Boost 1.70+ should work without manual intervention.

Minimum compiler versions

The following C++ compilers and their minimum versions can build the Solidity codebase :

- GCC, version 5+
- Clang, version 3.4+
- MSVC, version 2017+

Prérequis - macOS

Pour macOS, assurez-vous d'avoir installer la dernière version de Xcode. Ceci contient le compilateur C++ Clang, l'IDE Xcode et d'autres outils de développement Apple qui sont nécessaires pour construire des applications C++ sous OS X. Si vous installez Xcode pour la première fois, ou si vous venez d'installer une nouvelle version, vous devrez accepter la licence avant de pouvoir compiler en ligne de commande :

```
sudo xcodebuild -license accept
```

Nos versions pour OS X exigent que vous installiez Homebrew <http://brew.sh>‘_http://brew.sh pour l'installation des dépendances externes. Voici comment ‘désinstaller Homebrew, si vous voulez recommencer à zéro.

Prérequis - Windows

Vous aurez besoin des dépendances suivants pour compiler Solidity sous Windows :

Software	Notes
Visual Studio 2017 Build Tools	C++ compiler
Visual Studio 2017 (Optional)	C++ compiler and dev environment.

Si vous avez déjà eu un IDE et que vous n'avez besoin que du compilateur et des bibliothèques, vous pouvez installer Visual Studio 2017 Build Tools.

Visual Studio 2017 fournit à la fois l'IDE et le compilateur et les bibliothèques nécessaires. Donc si vous n'avez pas d'IDE et que vous préférez développer en Solidity, Visual Studio 2017 peut être un choix pour tout installer facilement.

Voici la liste des composants à installer dans Visual Studio 2017 Build Tools ou Visual Studio 2017 :

- Visual Studio C+++ fonctionnalités de base
- VC+++ 2017 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- Support C+++/CLI

Dependencies Helper Script

We have a helper script which you can use to install all required external dependencies on macOS, Windows and on numerous Linux distros.

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

Or, on Windows :

```
scripts\install_deps.bat
```

Clonez le dépot

Pour cloner le code source, exécutez la commande suivante :

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

Si vous voulez aider à développer Solidity, vous devriez forker Solidity et ajouter votre fork comme un second dépot distant :

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

Compilation en ligne de commande

Soyez sûrs d'installer les dépendances externes avant de compiler.

Le projet Solidity utilise CMake pour la configuration de compilation. Vous voulez peut-être installer ccache pour accélérer des compilations successives. CMake l'utilisera automatiquement. Compiler Solidity est similaire sur Linux, macOS et autres systèmes Unix :

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

ou même sous Linux et macOS, vous pouvez :

```
#note: les binaires de solc et les tests seront installés dans usr/local/bin
./scripts/build.sh
```

Avertissement : BSD builds should work, but are untested by the Solidity team.

Et pour Windows :

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

Ce dernier ensemble d'instructions devrait aboutir à la création de **solidity.sln** dans ce répertoire de compilation. Double-cliquer sur ce fichier devrait faire démarrer Visual Studio. Nous suggérons de construire la configuration **RelWithDebugInfo**, mais toutes les autres fonctionnent.

Alternativement, vous pouvez compiler pour Windows en ligne de commande, comme ça :

```
cmake --build . --config Release
```

3.2.6 Options de CMake

La liste des options de Cmake est disponible via la commande : `cmake .. -LH`.

Solveurs SMT

Solidity peut être compilé avec les solveurs SMT et le fera par défaut s'ils sont trouvés dans le système. Chaque solveur peut être désactivé par une option *cmake*.

Remarque : Dans certains cas, cela peut également être une solution de contournement potentielle en cas d'échec de compilation.

Dans le dossier de compilation, vous pouvez les désactiver, car ils sont activés par défaut :

```
# désactive seulement Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# désactive seulement CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF

# désactive Z3 et CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.2.7 La string de version en détail

La string de version de Solidity contient 4 parties :

- le numéro de version
- la balise de pre-version, généralement définie sur `develop.YYYY.MM.DD` ou `nightly.YYYY.MM.DD`.
- commit au format `commit.GITHASH`.
- plate-forme, qui a un nombre arbitraire d'éléments, contenant des détails sur la plate-forme et le compilateur

S'il y a des modifications locales, le commit sera suffixé avec `.mod`.

Ces parties sont combinées comme l'exige Semver, où la balise de pré-version Solidity est identique à la pré-version de Semver. et le commit Solidity et la plate-forme Solidity combinés constituent les métadonnées de la construction Semver.

Un exemple de version : `` `0.4.8+commit.60cc1668.Emscripten.clang ``.

Un exemple de pré-version : `0.4.9-nightly.2017.1.17+commit.6ecb4aaa3.Emscripten.clang`

3.2.8 Informations importantes concernant le versionnage

Après la sortie d'une version, la version de correctif est incrémentée, parce que nous supposons que seulement les changements de niveau patch suivent. Lorsque les modifications sont fusionnées, la version doit être supprimée en fonction des éléments suivants et la gravité du changement. Enfin, une version est toujours basée sur la nigthly actuelle, mais sans le spécificateur `prerelease`.

Exemple :

0. la version 0.4.0 est faite
1. nightly build a une version de 0.4.1 à partir de maintenant
2. des modifications incessantes sont introduites - pas de changement de version
3. un changement de rupture est introduit - la version est augmentée à 0.5.0
4. la version 0.5.0 est faite

Ce comportement fonctionne bien avec le *version pragma*.

3.3 Solidity par l'Exemple

3.3.1 Vote

Le contrat suivant est assez complexe, mais il présente de nombreuses caractéristiques de Solidity. Il implémente un contrat de vote. Bien entendu, le principal problème du vote électronique est de savoir comment attribuer les droits de vote aux bonnes personnes et éviter les manipulations. Nous ne résoudrons pas tous les problèmes ici, mais nous montrerons au moins comment le vote délégué peut être effectué de manière à ce que le dépouillement soit à la fois **automatique et totalement transparent**.

L'idée est de créer un contrat par bulletin de vote, en donnant un nom court à chaque option. Ensuite, le créateur du contrat qui agit à titre de président donnera le droit de vote à chaque adresse individuellement.

Les personnes derrière les adresses peuvent alors choisir de voter elles-mêmes ou de déléguer leur vote à une personne en qui elles ont confiance.

A la fin du temps de vote, la `winningProposal()` (proposition gagnante) retournera la proposition avec le plus grand nombre de votes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

/// @title Vote par délegation.
contract Ballot {
    // Ceci déclare un type complexe, représentant
    // un votant, qui sera utilisé
    // pour les variables plus tard.
    struct Voter {
        uint weight; // weight (poids), qui s'accumule avec les délégations
        bool voted;  // si true, cette personne a déjà voté
        address delegate; // Cette personne a délégué son vote à
        uint vote;   // index la la proposition choisie
    }

    // Type pour une proposition.
    struct Proposal {
        bytes32 name;   // nom court (jusqu'à 32 octets)
        uint voteCount; // nombre de votes cumulés
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    address public chairperson;

    // Ceci déclare une variable d'état qui stocke
    // un élément de structure 'Voters' pour  chaque votant.
    mapping(address => Voter) public voters;

    // Un tableau dynamique de structs `Proposal`.
    Proposal[] public proposals;

    /// Créé un nouveau bulletin pour choisir l'un des `proposalNames`.
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // Pour chacun des noms proposés,
        // crée un nouvel objet proposal
        // à la fin du tableau.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` créé un objet temporaire
            // Proposal et `proposals.push(...)`
            // l'ajoute à la fin du tableau `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Donne à un `voter` un droit de vote pour ce scrutin.
    // Peut seulement être appelé par `chairperson`.
    function giveRightToVote(address voter) public {
        // Si le premier argument passé à `require` s'évalue
        // à `false`, l'exécution s'arrete et tous les changements
        // à l'état et aux soldes sont annulés.
        // Cette opération consommait tout le gas dans
        // d'anciennes versions de l'EVM, plus maintenant.
        // Il est souvent une bonne idée d'appeler `require`
        // pour vérifier si les appels de fonctions
        // s'effectuent correctement.
        // Comme second argument, vous pouvez fournir une
        // phrase explicative de ce qui est allé de travers.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegue son vote au votant `to`.
    function delegate(address to) public {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // assigne les références
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // Relaie la délégation tant que `to`
        // est également en délégation de vote.
        // En général, ce type de boucles est très dangereux,
        // puisque s'il tourne trop longtemps, l'opération
        // pourrait demander plus de gas qu'il n'est possible
        // d'en avoir dans un bloc.
        // Dans ce cas, la délégation ne se ferait pas,
        // mais dans d'autres circonstances, ces boucles
        // peuvent complètement paraliser un contrat.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // On a trouvé une boucle dans la chaine
            // de délégations => interdit.
            require(to != msg.sender, "Found loop in delegation.");
        }

        // Comme `sender` est une référence, ceci
        // modifie `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // Si le délégué a déjà voté,
            // on ajoute directement le vote aux autres
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // Sinon, on l'ajoute au poids de son vote.
            delegate_.weight += sender.weight;
        }
    }

    /// Voter (incluant les procurations par délégation)
    /// pour la proposition `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // Si `proposal` n'est pas un index valide,
        // une erreur sera levée et l'exécution annulée
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Calcule la proposition gagnante
    /// en prenant tous les votes précédents en compte.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Appelle la fonction winningProposal() pour avoir
    // l'index du gagnant dans le tableau de propositions
    // et retourne le nom de la proposition gagnante.
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Améliorations possibles

À l'heure actuelle, de nombreuses opérations sont nécessaires pour attribuer les droits de vote à tous les participants. Pouvez-vous trouver un meilleur moyen ?

3.3.2 Enchères à l'aveugle

Dans cette section, nous allons montrer à quel point il est facile de créer un contrat d'enchères à l'aveugle sur Ethereum. Nous commencerons par une enchère ouverte où tout le monde pourra voir les offres qui sont faites, puis nous prolongerons ce contrat dans une enchère aveugle où il n'est pas possible de voir l'offre réelle avant la fin de la période de soumission.

Enchère ouverte simple

L'idée générale du contrat d'enchère simple suivant est que chacun peut envoyer ses offres pendant une période d'enchère. Les ordres incluent l'envoi d'argent / éther afin de lier les soumissionnaires à leur offre. Si l'enchère est la plus haute, l'enchérisseur qui avait fait l'offre la plus élevée auparavant récupère son argent. Après la fin de la période de soumission, le contrat doit être appelé manuellement pour que le bénéficiaire reçoive son argent - les contrats ne peuvent pas s'activer eux-mêmes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract SimpleAuction {
    // Paramètres de l'enchère
    // temps unix absolus (secondes depuis 01-01-1970)
    // ou des durées en secondes.
    address payable public beneficiary;
    uint public auctionEndTime;

    // État actuel de l'enchère.
    address public highestBidder;
    uint public highestBid;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Remboursements autorisés d'enchères précédentes
    mapping(address => uint) pendingReturns;

    // Mis à true à la fin, interdit tout changement.
    // Par defaut à `false`, comme un grand.
    bool ended;

    // Évènements déclenchés aux changements.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Ce qui suit est appelé commentaire natspec,
    // reconaissable à ses 3 slashes.
    // Ce message sera affiché quand l'utilisateur
    // devra confirmer une transaction.

    /// Créée une enchère simple de `_biddingTime`
    /// secondes au profit de l'addresse
    /// beneficaire address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEndTime = now + _biddingTime;
    }

    /// Faire une offre avec la valeur envoyée
    /// avec cette transaction.
    /// La valeur ne sera remboursée que si
    // l'enchère est perdue.
    function bid() public payable {
        // Aucun argument n'est nécessaire, toute
        // l'information fait déjà partie
        // de la transaction. Le mot-clé payable
        // est requis pour autoriser la fonction
        // à recevoir de l'Ether.

        // Annule l'appel si l'enchère est termminée
        require(
            now <= auctionEndTime,
            "Auction already ended."
        );

        // Rembourse si l'enchère est trop basse
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );

        if (highestBid != 0) {
            // Renvoyer l'argent avec un simple
            // highestBidder.send(highestBid) est un risque de sécurité
            // car ça pourrait déclencher un appel à un contrat.
            // Il est toujours plus sûr de laisser les utilisateurs
            // retirer leur argent eux-mêmes.
            pendingReturns[highestBidder] += highestBid;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Retirer l'argent d'une enchère dépassée
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Il est important de mettre cette valeur à zéro car l'utilisateur
            // pourrait rappeler cette fonction avant le retour de `send`.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // Pas besoin d'avorter avec un throw ici, juste restaurer le montant
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// Met fin à l'enchère et envoie
    /// le montant de l'enchère la plus haute au bénéficiaire.
    function auctionEnd() public {
        // C'est une bonne pratique de structurer les fonctions qui
        // intéragissent avec d'autres contrats (appellent des
        // fonctions ou envoient de l'Ether) en trois phases:
        // 1. Vérifier les conditions
        // 2. éffectuer les actions (potentiellement changeant les conditions)
        // 3. interagir avec les autres contrats
        // Si ces phases sont mélangées, l'autre contrat pourrait rappeler
        // le contrat courant et modifier l'état ou causer des effets
        // (paiements en Ether par ex) qui se produiraient plusieurs fois.
        // Si des fonctions appelées en interne effectuent des appels
        // à des contrats externes, elles doivent aussi êtres considérées
        // comme concernées par cette norme.

        // 1. Conditions
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

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

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```

Enchère aveugle

L'enchère ouverte précédente est étendue en une enchère aveugle dans ce qui suit. L'avantage d'une enchère aveugle est qu'il n'y a pas de pression temporelle vers la fin de la période de soumission. La création d'une enchère aveugle sur

une plate-forme informatique transparente peut sembler une contradiction, mais la cryptographie vient à la rescousse.

Pendant la **période de soumission**, un soumissionnaire n'envoie pas son offre, mais seulement une version hachée de celle-ci. Puisqu'il est actuellement considéré comme pratiquement impossible de trouver deux valeurs (suffisamment longues) dont les valeurs de hachage sont égales, le soumissionnaire s'engage à l'offre par cela. Après la fin de la période de soumission, les soumissionnaires doivent révéler leurs offres : Ils envoient leurs valeurs en clair et le contrat vérifie que la valeur de hachage est la même que celle fournie pendant la période de soumission.

Un autre défi est de savoir comment rendre l'enchère **contraignante et aveugle** en même temps : La seule façon d'éviter que l'enchérisseur n'envoie pas l'argent après avoir gagné l'enchère est de le lui faire envoyer avec l'enchère. Puisque les transferts de valeur ne peuvent pas être aveuglés dans Ethereum, tout le monde peut voir la valeur.

Le contrat suivant résout ce problème en acceptant toute valeur supérieure à l'offre la plus élevée. Comme cela ne peut bien sûr être vérifié que pendant la phase de révélation, certaines offres peuvent être **invalides**, et c'est fait exprès (il fournit même un marqueur explicite pour placer des offres invalides avec des transferts de grande valeur) : Les soumissionnaires peuvent brouiller la concurrence en plaçant plusieurs offres invalides hautes ou basses.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

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

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

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

    address public highestBidder;
    uint public highestBid;

    // Remboursements autorisés d'enchères précédentes
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Les Modifiers sont une façon pratique de valider des entrées.
    /// `onlyBefore` est appliqué à `bid` ci-dessous:
    /// Le corps de la fonction sera placé dans le modifier
    /// où `_` est placé.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// Placer une enchère à l'aveugle avec `_blindedBid` =
    /// keccak256(abi.encodePacked(value, fake, secret)).
    ///  L'éther envoyé n'est remboursé que si l'enchère est correctement
    /// révélée dans la phase de révélation. L'offre est valide si
    /// l'éther envoyé avec l'offre est d'au moins "valeur" et
    /// "fake" n'est pas true. Régler "fake" à true et envoyer
    /// envoyer un montant erroné sont des façons de masquer l'enchère
    /// mais font toujours le dépot requis. La même addresse peut placer
    /// plusieurs ordres
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Révèle vos ench1eres aveugles. Vous serez remboursé pour toutes
    /// les enchères invalides et toutes les autres exceptée la plus haute
    /// le cas échéant.
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // L'enchère n'a pas été révélée.
                // Ne pas rembourser.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Rendre impossible un double remboursement
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    // Cette fonction interne ("internal") ne peut être appelée que
    // que depuis l'intérieur du contrat (ou ses contrats dérivés).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Rembourse la précédent leader.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Se faire rembourser une enchère battue.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Il est important de mettre cette valeur à zéro car l'utilisateur
            // pourrait rappeler cette fonction avant le retour de `send`.
            // (voir remarque sur conditions -> effets -> interaction).
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// Met fin à l'enchère et envoie
    /// le montant de l'enchère la plus haute au bénéficiaire.
    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

3.3.3 Achat distant sécurisé

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment here : There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the

value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() public payable {
        seller = msg.sender;
        value = msg.value / 2;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        require((2 * value) == msg.value, "Value has to be even.");
    }

    /// Annule l'achat et rembourse l'ether du dépot.
    /// Peut seulement être appelé par le vendeur
    /// avant le verrouillage du contrat
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // We use transfer here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        seller.transfer(address(this).balance);
    }

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

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Release;

        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        public
        onlySeller
        inState(State.Release)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}
```

3.3.4 Canaux de micro-paiement

Dans cette section, nous allons apprendre comment construire une implémentation simple d'un canal de paiement. Il utilise des signatures cryptographiques pour effectuer des transferts répétés d'Ether entre les mêmes parties en toute sécurité, instantanément et sans frais de transaction. Pour ce faire, nous devons comprendre comment signer et vérifier les signatures, et configurer le canal de paiement.

Création et vérification des signatures

Imaginez qu'Alice veuille envoyer une quantité d'Ether à Bob, c'est-à-dire qu'Alice est l'expéditeur et Bob est le destinataire. Alice n'a qu'à envoyer des messages cryptographiquement signés hors chaîne (par exemple par e-mail) à Bob et cela sera très similaire à la rédaction de chèques.

Les signatures sont utilisées pour autoriser les transactions et sont un outil généraliste à la disposition des contrats intelligents. Alice construira un simple contrat intelligent qui lui permettra de transmettre des Ether, mais d'une manière inhabituelle, au lieu d'appeler une fonction elle-même pour initier un paiement, elle laissera Bob le faire, et donc payer les frais de transaction.

Le contrat fonctionnera comme suit :

1. Alice déploie le contrat `ReceiverPays` en y attachant suffisamment d'éther pour couvrir les paiements qui seront effectués.
2. Alice autorise un paiement en signant un message avec sa clé privée.
3. Alice envoie le message signé cryptographiquement à Bob. Le message n'a pas besoin d'être gardé secret (vous le comprendrez plus tard), et le mécanisme pour l'envoyer n'a pas d'importance.
4. Bob réclame leur paiement en présentant le message signé au contrat intelligent, il vérifie l'authenticité du message et libère ensuite les fonds.

Création de la signature

Alice n'a pas besoin d'interagir avec le réseau Ethereum pour signer la transaction, le processus est complètement hors ligne. Dans ce tutoriel, nous allons signer les messages dans le navigateur en utilisant web3.js et MetaMask. En particulier, nous utiliserons la méthode standard décrite dans EIP-762, car elle offre un certain nombre d'autres avantages en matière de sécurité.

```
/// Hasher d'abord simplifie un peu les choses
var hash = web3.sha3("message to sign");
web3.personal.sign(hash, web3.eth.defaultAccount, function () {...});
```

Note : Notez que `web3.personal.sign` préfixe les données signées de la longueur du message. Mais comme nous avons hashé en premier, le message sera toujours exactement 32 octets de long, et donc ce préfixe de longueur est toujours le même, ce qui facilite tout.

Que signer

Dans le cas d'un contrat qui effectue des paiements, le message signé doit inclure :

1. Adresse du destinataire
2. le montant à transférer
3. Protection contre les attaques de rediffusion

Une attaque de rediffusion se produit lorsqu'un message signé est réutilisé pour revendiquer l'autorisation pour une deuxième action. Pour éviter les attaques par rediffusion, nous utiliserons la même méthode que pour les transactions Ethereum elles-mêmes, ce qu'on appelle un nonce, qui est le nombre de transactions envoyées par un compte. Le contrat intelligent vérifiera si un nonce est utilisé plusieurs fois.

Il existe un autre type d'attaques de redifussion, il se produit lorsque le propriétaire déploie un smart contract `ReceiverPays`, effectue certains paiements, et ensuite détruit le contrat. Plus tard, il décide de déployer `ReceiverPays` encore une fois, mais le nouveau contrat ne peut pas connaître les nonces utilisés dans le déploiement précédent, donc l'attaquant peut réutiliser les anciens messages.

Alice peut s'en protéger, notamment en incluant l'adresse du contrat dans le message, et seulement les messages contenant l'adresse du contrat lui-même seront acceptés. Cette fonctionnalité se trouve dans les deux premières lignes de la fonction `claimPayment()` du contrat complet à la fin de ce chapitre.

Encoder les arguments

Maintenant que nous avons déterminé quelles informations inclure dans le message signé, nous sommes prêts à assembler le message, à le hacher, et le signer. Par souci de simplicité, nous ne faisons que concaténer les données. La bibliothèque ethereumjs-abi fournit une fonction appelée `solditySHA3` qui imite le comportement de la fonction `keccak256` de Solidity appliquée aux arguments codés en utilisant `abi.encododePacked`. En résumé, voici une fonction JavaScript qui crée la signature appropriée pour l'exemple `ReceiverPays` :

```
// recipient est l'addresse à payer.
// amount, en wei, spécifie combien d'Ether doivent être envoyés.
// nonce peut être n'importe quel nombre unique pour prévenir les attques par␣
↪redifusion
// contractAddress est utilisé pour éviter les attaque par redifusion de messages␣
↪inter-contrats
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + ethereumjs.ABI.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

Récupérer le signataire du message en Solidity

En général, les signatures ECDSA se composent de deux paramètres, `r` et `s`. Les signatures dans Ethereum incluent un troisième paramètre appelé « v », qui peut être utilisé pour récupérer la clé privée du compte qui a été utilisée pour signer le message, l'expéditeur de la transaction. La solidité offre une fonction intégrée ecrecover qui accepte un message avec les paramètres `r`, `s` et `v` et renvoie l'adresse qui a été utilisée pour signer le message.

Extraire les paramètres de signature

Les signatures produites par web3.js sont la concaténation de `r`, `s` et `v`, donc la première étape est de re-séparer ces paramètres. Cela peut être fait sur le client, mais le faire à l'intérieur du smart contract signifie qu'un seul paramètre de signature peut être envoyé au lieu de trois. Diviser un tableau d'octets en plusieurs parties est un peu compliqué. Nous utiliserons l'"assembleur en ligne pour faire le travail dans la fonction `splitSignature` (la troisième fonction dans le contrat complet à la fin du présent chapitre).

Calculer le hash du message

Le smart contract doit savoir exactement quels paramètres ont été signés, et doit donc recréer le message à partir des paramètres et utiliser cette fonction pour la vérification des signatures. Les fonctions `prefixed` et `recoverSigner` s'occupent de cela et leur utilisation peut se trouver dans la fonction `claimPayment`.

Le contrat complet

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.24 <0.7.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() public payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)␣
↪public {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // Cette ligne recrée le message signé par le client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount,␣
↪nonce, this)));

        require(recoverSigner(message, signature) == owner);

        msg.sender.transfer(amount);
    }

    /// détruit le contrat et réclame son solde.
    function shutdown() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// methodes de signature.
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // premiers 32 octets, après le préfixe
            r := mload(add(sig, 32))
            // 32 octets suivants
            s := mload(add(sig, 64))
            // Octet final (premier du prochain lot de 32)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// construit un hash préfixé pour mimer le comportement de eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Écrire un canal de paiement simple

Alice va maintenant construire une implémentation simple mais complète d'un canal de paiement. Les canaux de paiement utilisent des signatures cryptographiques pour effectuer des virements répétés d'Ether en toute sécurité, instantanément et sans frais de transaction.

Qu'est-ce qu'un canal de paiement ?

Les canaux de paiement permettent aux participants d'effectuer des transferts répétés d'Ether sans utiliser de transactions. Cela signifie que les délais et frais associés aux transactions peuvent être évités. Nous allons explorer un canal de paiement unidirectionnel simple entre deux parties (Alice et Bob). Son utilisation implique trois étapes :

1. Alice déploie un smart contract avec de l'Ether. Cela « ouvre » (`opens`) le canal de paiement.
2. Alice signe des messages qui précisent combien d'éther est dû au destinataire. Cette étape est répétée pour chaque paiement.
3. Bob « ferme » (`closes`) le canal de paiement, retirant leur part d'Ether et renvoyant le reste à l'expéditeur.

Note : Non seulement les étapes 1 et 3 exigent des transactions Ethereum, mais l'étape 2 signifie que l'expéditeur transmet un message signé cryptographiquement au destinataire par des moyens hors chaîne (par exemple, par courrier électronique). Cela signifie que seulement deux transactions sont nécessaires pour traiter un nombre quelconque de transferts.

Bob est assuré de recevoir ses fonds parce que le contrat bloque les fonds en Ether et respecte des ordres valides et signés. Le smart contract impose également un délai d'attente, Alice est donc assurée de recouvrer ses fonds même si le bénéficiaire refuse de fermer le canal. C'est aux participants d'un canal de paiement de décider combien de temps il doit rester ouvert. Pour une transaction de courte durée, comme payer un cybercafé pour chaque minute d'accès au réseau, ou dans le cas d'une relation de plus longue durée, comme le versement d'un salaire horaire à un employé, un paiement pourrait durer des mois ou des années.

Ouverture du canal de paiement

Pour ouvrir le canal de paiement, Alice déploie le contrat, y attachant de l'Ether en dépot et spécifiant le destinataire prévu, ainsi qu'une durée de vie maximale du canal. C'est la fonction `SimplePaymentChannel` dans le contrat.

Effectuer des paiements

Alice effectue des paiements en envoyant des messages signés à Bob. Cette étape est entièrement réalisée en dehors du réseau Ethereum. Les messages sont signés cryptographiquement par l'expéditeur puis transmis directement au destinataire.

Chaque message contient les informations suivantes :

— L'adresse du contrat, utilisé pour empêcher les attaques de redifussion par contrats croisés.
— Le montant total d'Ether qui est dû au bénéficiaire jusqu'alors.

Un canal de paiement est fermé une seule fois, à la fin d'une série de virements. De ce fait, un seul des messages envoyés sera échangé. C'est pourquoi chaque message spécifie un montant total cumulatif d'éther dû, plutôt que le montant total d'un micropaiement individuel. Le destinataire réclamera naturellement le message le plus récent parce que c'est celui dont le total est le plus élevé. Le nonce par message n'est plus nécessaire, car le smart contract ne va honorer qu'un seul message. L'adresse du contrat intelligent est toujours utilisée pour éviter qu'un message destiné à un canal de paiement ne soit utilisé pour un autre canal.

Voici le code javascript modifié pour signer cryptographiquement un message du chapitre précédent :

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.solidity.SHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress détectera la rediffusion de messages à d'autres contrats.
// amount, en wei, précise combien d'Ether doivent être envoyés.
```

(suite sur la page suivante)

(suite de la page précédente)

```
function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

Fermeture du canal de paiement

Lorsque Bob est prêt à recevoir leurs ses, il est temps de fermer le canal de paiement en appelant une fonction `close` sur le smart contract. La fermeture du canal paie au destinataire l'Ether qui lui est dû et détruit le contrat, en renvoyant tout Ether restant à Alice. Pour fermer le canal, Bob doit fournir un message signé par Alice.

Le contrat doit vérifier que le message contient une signature valide de l'expéditeur. Le processus de vérification est le même que celui utilisé par le destinataire. Les fonctions Solidity `isValidSignature` et `recoverSigner` fonctionnent de la même manière que leurs fonctions JavaScript dans la section précédente. Ce dernier est emprunté au Le contrat `ReceiverPays` du chapitre précédent.

La fonction `close` ne peut être appelée que par le destinataire du canal de paiement, qui enverra naturellement le message de paiement le plus récent car c'est celui qui comporte le plus haut total dû. Si l'expéditeur était autorisé à appeler cette fonction, il pourrait fournir un message avec un montant inférieur et escroquer le destinataire de ce qui lui est dû.

La fonction vérifie que le message signé correspond aux paramètres donnés. Si tout se passe bien, le destinataire reçoit sa part d'Ether, et l'expéditeur reçoit le reste par `selfdestruct` (autodestruction) du contrat. Vous pouvez voir la fonction `close` dans le contrat complet.

Expiration du canal

Bob peut fermer le canal de paiement à tout moment, mais s'il ne le fait pas, Alice a besoin d'un moyen de récupérer les fonds bloqués. Une durée d'*expiration* a été définie au moment du déploiement du contrat. Une fois cette heure atteinte, Alice peut appeler pour récupérer leurs fonds. Vous pouvez voir la fonction `claimTimeout` dans le contrat complet.

Après l'appel de cette fonction, Bob ne peut plus recevoir d'Ether. Il est donc important que Bob ferme le canal avant que l'expiration ne soit atteinte.

Le contrat complet

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract SimplePaymentChannel {
    address payable public sender;      // Le compte emvoyant les paiements.
    address payable public recipient;   // Le compte destinataire des paiements.
    uint256 public expiration;  // Expitration si le destinataire ne clot pas le␣
↪canal.


    constructor (address payable _recipient, uint256 duration)
        public
        payable
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        sender = msg.sender;
        recipient = _recipient;
        expiration = now + duration;
    }

    /// Le destinataire peut clore le canal à tout moment en présentant le dernier␣
↪montant
    /// signé par l'expéditeur des fonds. Le destinataire se verra verser ce montant,
    /// et le reste sera rendu à l'emetteur des fonds.
    function close(uint256 amount, bytes memory signature) public {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// L'emetteur peut modifier la date d'expiration à tout moment
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// Si l'expiration est atteinte avant cloture par le destinataire,
    /// l'Ether est renvoyé à l'emetteur
    function claimTimeout() public {
        require(now >= expiration);
        selfdestruct(sender);
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// Toutes les fonctions ci-dessous sont tirées
    /// du chapitre 'créer et vérifier les signatures'.

    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
```

(suite sur la page suivante)

(suite de la page précédente)

```
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// construit un hash préfixé pour mimer le comportement de eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Note : La fonction `splitSignature` est très simple et n'utilise pas tous les contrôles de sécurité. Une implémentation réelle devrait utiliser une bibliothèque plus rigoureusement testée de ce code, tel que le fait openzepplin avec version of this code.

Vérification des paiements

Contrairement à notre chapitre précédent, les messages dans un canal de paiement ne sont pas appliqués tout de suite. Le destinataire conserve la trace du dernier message et le fait parvenir au réseau quand il est temps de fermer le canal de paiement. Cela signifie qu'il est essentiel que le destinataire effectue sa propre vérification de chaque message. Sinon, il n'y a aucune garantie que le destinataire sera en mesure d'être payé à la fin.

Le destinataire doit vérifier chaque message à l'aide du processus suivant :

1. Vérifiez que l'adresse du contact dans le message correspond au canal de paiement.
2. Vérifiez que le nouveau total est le montant prévu.
3. Vérifier que le nouveau total ne dépasse pas la quantité d'éther déposée.
4. Vérifiez que la signature est valide et provient de l'expéditeur du canal de paiement.

Nous utiliserons la librairie *ethereumjs-util <https ://github.com/ethereumjs/ethereumjs-util>'_https ://github.com/ethereumjs/ethereumjs-util pour écrire ces vérifications. L'étape finale peut se faire de plusieurs façons, ici en JavaScript, Le code suivant emprunte la fonction 'constructMessage* du **code JavaScript** de signature ci-dessus :

```
// Cette ligne mine le fonctionnement de la méthode JSON-RPC de eth_sign.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

3.3.5 Modular Contracts

A modular approach to building your contracts helps you reduce the complexity and improve the readability which will help to identify bugs and vulnerabilities during development and code review. If you specify and control the behaviour or each module in isolation, the interactions you have to consider are only those between the module specifications and not every other moving part of the contract. In the example below, the contract uses the `move` method of the `Balances` *library* to check that balances sent between addresses match what you expect. In this way, the `Balances` library provides an isolated component that properly tracks balances of accounts. It is easy to verify that the `Balances` library never produces negative balances or overflows and the sum of all balances is an invariant across the lifetime of the contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address␣
↪to, uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) public returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;

    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function transferFrom(address from, address to, uint amount) public returns (bool␣
↪success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) public returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }

    function balanceOf(address tokenOwner) public view returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

3.4 Structure d'un fichier source Solidity

Les fichiers sources peuvent contenir un nombre arbitraire de *définitions de contrats*, directives d'*import* et *directives pragma* et définitions de *struct* et d'*enum*.

SPDX License Identifier

3.4.1 Pragmas

Le mot-clé `pragma` peut être utilisé pour activer certaines fonctions ou vérifications du compilateur. Une directive pragma est toujours locale à un fichier source, vous devez donc ajouter pragma à tous vos fichiers si vous voulez l'activer dans tout votre projet. Si vous *importez* un autre fichier, le pragma de ce fichier ne s'appliquera pas automatiquement au fichier à importer.

Version Pragma

Les fichiers sources peuvent (et devraient) être annotés avec un `version pragma` pour refuser d'être compilés avec de futures versions de compilateurs qui pourraient introduire des changements incompatibles. Nous essayons de limiter ces changements au strict minimum, et en particulier introduire des changements d'une manière telle que les changements de sémantique nécessiteront également des changements de syntaxe, mais ce n'est bien sûr pas toujours possible. Pour cette raison, c'est toujours une bonne idée de lire le fichier des modifications (« changelog ») au moins pour les versions qui contiennent des changements de rupture, ces versions auront toujours des versions de la forme `0.x.0` ou `x.0.0`.

Le `version pragma` est utilisée comme suit :

```
pragma solidity ^0.4.0;
```

Un tel fichier source ne compilera pas avec un compilateur antérieur à la version 0.4.0 et ne fonctionnera pas non plus sur un compilateur à partir de la version 0.5.0 (cette deuxième condition est ajoutée en utilisant `^`). L'idée derrière

cela est la supposition qu'il n'y aura pas de changements de rupture jusqu'à la version `0.5.0`, donc nous pouvons toujours être sûrs que notre code compilera la façon dont nous l'avons prévu. Nous ne précisons pas la version exacte de correctif du compilateur, de sorte que les versions corrigées sont toujours possibles.

Il est possible de spécifier des règles beaucoup plus complexes pour la version du compilateur, la syntaxe suit celle utilisée par npm.

Note : L'utilisation de `version pragma` ne changera pas la version du compilateur. Il n'activera ou désactivera pas non plus les fonctions du compilateur. Il demandera simplement au compilateur de vérifier si sa version correspond à celle requise par le pragma. S'il ne correspond pas, le compilateur affichera une erreur.

Pragma Expérimental

Le deuxième pragma est le `experimental pragma`. Il peut être utilisé pour activer des fonctions du compilateur ou de la langue qui ne sont pas encore activées par défaut. Les pragmas expérimentaux suivants sont actuellement pris en charge :

ABIEncoderV2

Pragma Expérimental

Le deuxième pragma est le `experimental pragma`. Il peut être utilisé pour activer des fonctions du compilateur ou de la langue qui ne sont pas encore activées par défaut. Les pragmas expérimentaux suivants sont actuellement pris en charge :

ABIEncoderV2

Le nouvel encodeur ABI est capable d'encoder et de décoder arbitrairement des tableaux et des structures imbriqués. Il produit un code moins optimal (l'optimiseur pour cette partie du code est encore en développement) et n'a pas reçu autant de tests que l'ancien codeur. Vous pouvez l'activer en utilisant `pragma experimental ABIEncoderV2;` . - we kept the same pragma, even though it is not considered experimental since Solidity 0.6.0 anymore.

SMTChecker

Ce composant doit être activé lors de la compilation du compilateur et n'est par conséquent pas forcément présent dans tous les binaires Solidity. Les *instructions de compilation* expliquent comment activer cette option. Elle est activée pour les versions PPA d'Ubuntu dans la plupart des versions, mais pas pour solc-js, les images Docker, les binaires Windows ni les binaires Linux pré-compilés. It can be activated for solc-js via the smtCallback if you have an SMT solver installed locally and run solc-js via node (not via the browser).

Si vous utilisez `pragma experimental SMTChecker;`, vous aurez des *avertissements de sécuristé* supplémentaires qui sont obtenus en interrogeant un solveur SMT. Le composant ne prend pas encore en charge toutes les fonctionnalités du langage Solidity et émet probablement de nombreux avertissements. Dans le cas où il signale des caractéristiques non prises en charge, l'analyse peut ne pas être cohérente.

3.4.2 Importation d'autres fichiers sources

Syntaxe et sémantique

Solidity supporte les instructions d'importation qui sont très similaires à celles disponibles en JavaScript (à partir de ES6), bien que Solidity ne connaisse pas le concept de default export.

Au niveau global, vous pouvez utiliser les instructions d'importation sous la forme suivante :

```
import "filename";
```

Cette instruction importe tous les symboles globaux de « nom de fichier » (et les symboles qui y sont importés) dans le champ d'application global actuel (différent de celui de ES6 mais rétrocompatible pour Solidity). Cette syntaxe simple n'est pas recommandée car elle pollue l'espace de nommage d'une manière imprévisible : Si vous ajoutez de nouveaux éléments de niveau supérieur dans « nom de fichier », ils apparaîtront automatiquement dans tous les fichiers qui importent ainsi à partir de « nom de fichier ». Il est préférable d'importer explicitement des symboles spécifiques.

L'exemple suivant crée un nouveau symbole global `symbolName` dont les membres sont tous les symboles globaux de `"filename"`.

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

which results in all global symbols being available in the format `symbolName.symbol`.

A variant of this syntax that is not part of ES6, but possibly useful is :

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

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

En cas de collision de noms, vous pouvez également renommer les symboles lors de l'importation. Ce code crée de nouveaux symboles globaux `alias` et `symbole2` qui font référence à `symbole1` et `symbole2` de `"nom de fichier"`, respectivement.

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

Chemins

Ci-dessus, `nom-de-fichier` est toujours traité comme un chemin avec `/` comme séparateur de répertoire, `.` comme le répertoire courant et `..` comme le répertoire parent. Lorsque `.``ou ``..` est suivi d'un caractère autre que `/`, il n'est pas considéré comme le répertoire courant ou parent. Tous les noms de chemins sont traités comme des chemins absolus à moins qu'ils ne commencent par le répertoire courant `.` ou le répertoire parent `..`.

Pour importer un fichier `x` du même répertoire que le fichier courant, utilisez `import "./x" as x;`. Si vous utilisez `import "x" as x;` à la place, un fichier différent pourrait être référencé (d'un plus global « include directory »).

Il repose sur le compilateur (voir *Utilisation dans les compilateurs*) de résoudre les chemins. En général, la hiérarchie des répertoires n'a pas besoin de pointer strictement sur votre système de fichiers local, elle peut aussi pointer vers les ressources en ipfs, http ou git par exemple.

Note : Utilisez toujours des importations relatives comme `import "./filename.sol";` et évitez d'utiliser `..` dans les spécificateurs de chemins. Dans ce dernier cas, il est probablement préférable d'utiliser des chemins globaux et de configurer les remappages comme expliqué ci-dessous.

Utilisation dans les compilateurs

Lorsque vous invoquez le compilateur, vous pouvez spécifier comment découvrir le premier élément d'un chemin, ainsi que les remappages de préfixes de chemins. Par exemple, vous pouvez configurer un remappage de sorte que tout ce qui est importé du répertoire virtuel `github.com/ethereum/dapp-bin/library` soit réellement lu depuis votre répertoire local `/usr/local/dapp-bin/library`. Si plusieurs remappages s'appliquent, celui avec la clé la plus longue est essayé en premier. Un préfixe vide n'est pas autorisé. Les remappages peuvent dépendre d'un contexte, ce qui vous permet de configurer des paquets à importer, par exemple différentes versions d'une bibliothèque du même nom.

solc :

Pour solc (le compilateur de ligne de commande), vous fournissez ces chemins d'accès sous la forme d'arguments `context:prefix=target`, où les parties `context:``et ``target` sont optionnelles (`prefix` est la valeur par défaut de `target` dans ce cas). Toutes les valeurs de remappage qui sont des fichiers réguliers sont compilées (y compris leurs dépendances).

Ce mécanisme est rétrocompatible (tant qu'aucun nom de fichier ne contient = ou :`) et ne constitue donc pas un changement de rupture. Tous les fichiers dans ou sous le répertoire `context` qui importent un fichier commençant par `prefix` sont redirigés en remplaçant `prefix` par `target`.

Par exemple, si vous clonez `github.com/ethereum/dapp-bin/` localement vers `/usr/local/dapp-bin`, vous pouvez utiliser ce qui suit dans votre fichier source :

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

Puis lancer le compilateur :

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

Comme exemple plus complexe, supposons que vous utilisiez un module qui utilise une ancienne version de dapp-bin que vous avez extraite vers `/usr/local/dapp-bin_old`, alors vous pouvez exécuter :

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

Cela signifie que toutes les importations du `module2` pointent vers l'ancienne version mais les importations du `module1` pointent vers la nouvelle version.

Note : `solc` vous permet seulement d'inclure des fichiers de certains répertoires. Ils doivent être dans le répertoire (ou sous-répertoire) d'un des fichiers sources explicitement spécifiés ou dans le répertoire (ou sous-répertoire) d'une cible de remappage. Si vous voulez autoriser les includes absolus directs, ajoutez le remapping `/=//`.

S'il y a plusieurs remappages qui mènent à un fichier valide, le remappage avec le préfixe commun le plus long est choisi.

Remix :

Remix fournit un remappage automatique pour GitHub et récupère automatiquement le fichier en ligne. Vous pouvez importer le mappage itérable comme ci-dessus, par exemple :

: : import « github.com/ethereum/dapp-bin/library/iterable_mapping.sol » as it_mapping ;

Remix may add other source code providers in the future.

3.4.3 Commentaires

Les commentaires sur une seule ligne (`//`) et les commentaires sur plusieurs lignes (`/*...*/`) sont possibles.

```
// Ceci est un commentaire sur une ligne.

/*
Ceci est un commentaire
multi-lignes.
*/
```

Note : Un commentaire d'une seule ligne est terminé par tout terminateur de ligne unicode (LF, VF, FF, CR, NEL, LS ou PS) en codage utf8. Le terminateur fait toujours partie du code source après le commentaire, donc si ce n'est pas un symbole ascii (que sont NEL, LS et PS), il conduira à une erreur d'analyse.

De plus, il existe un autre type de commentaire appelé commentaire natspec, détaillé dans *style guide*. Ils sont écrits avec une triple barre oblique (`///`) ou un double bloc d'astérisque (`/** ... */`) et ils doivent être utilisés directement au-dessus des déclarations ou instructions de fonction. Vous pouvez utiliser les balises de style Doxygen à l'intérieur de ces commentaires pour documenter les fonctions, annoter les conditions de vérification, et fournir un **texte de confirmation** qui est montré aux utilisateurs lorsqu'ils tentent d'appeler une fonction.

Dans l'exemple suivant, nous documentons le titre du contrat, l'explication des deux paramètres d'entrée et les deux valeurs retournées.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.7.0;

/** @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) public pure returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

3.5 Structure d'un contrat

Les contrats Solidity sont similaires à des classes dans des langages orientés objet. Chaque contrat peut contenir des déclarations de *Variables d'état*, structure-fonctions, structure-fonction-modificateurs, structure-événements, *Types Structure* et *Types Enum*. De plus, les contrats peuvent hériter d'autres contrats.

Il existe également des types de contrats spéciaux appelés *libraries* et *interfaces*.

La section sur les contrats contient plus de détails que cette section, qui permet d'avoir une vue d'ensemble rapide.

3.5.1 Variables d'état

Les variables d'état sont des variables dont les valeurs sont stockées en permanence dans le storage du contrat.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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

Voir la section *Types* pour les types de variables d'état valides et *Visibilité et Getters* pour les choix possibles de visibilité.

3.5.2 Fonctions

Les fonctions sont les unités exécutables du code d'un contrat.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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

Les *Appels de fonction* peuvent se faire en interne ou en externe et ont différents niveaux de *visibilité* pour d'autres contrats. *Functions* accept *parameters and return variables* to pass parameters and values between them.

3.5.3 Modificateurs de fonction

Les modificateurs de fonction peuvent être utilisés pour modifier la sémantique des fonctions d'une manière déclarative (voir *Modificateurs de fonctions* dans la section contrats).

Overloading, that is, having the same modifier name with different parameters, is not possible.

Like functions, modifiers can be *overridden*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public view onlySeller { // Modifier usage
        // ...
    }
}
```

3.5.4 Évènements

Les évènements (`event`) sont une interface d'accès aux fonctionnalités de journalisation (logs) de l'EVM.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.7.0;

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

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

Voir *Événements* dans la section contrats pour plus d'informations sur la façon dont les événements sont déclarés et peuvent être utilisés à partir d'une dapp.

3.5.5 Types Structure

Les structures sont des types personnalisés qui peuvent regrouper plusieurs variables (voir *Structs* dans la section types).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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

3.5.6 Types Enum

Les Enumérateurs (`enum`) peuvent être utilisés pour créer des types personnalisés avec un ensemble fini de "valeurs constantes" (voir *Énumérateurs* dans la section Types).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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

3.6 Types

Solidity est un langage à typage statique, ce qui signifie que le type de chaque variable (état et locale) doit être spécifié. Solidity propose plusieurs types élémentaires qui peuvent être combinés pour former des types complexes.

De plus, les types peuvent interagir entre eux dans des expressions contenant des opérateurs. Pour une liste synthétique des différents opérateurs, voir *Order of Precedence of Operators*.

The concept of « undefined » or « null » values does not exist in Solidity, but newly declared variables always have a *default value* dependent on its type. To handle any unexpected values, you should use the *revert function* to revert the whole transaction, or return a tuple with a second `bool` value denoting success.

3.6.1 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.

Entiers

`int` / `uint` : Entiers signés et non-signés de différentes tailles. Les mots-clé `uint8` à `uint256` par pas de `8` (entier non signé de 8 à 256 bits) et `int8` à `int256`. `uint` et `int` sont des alias de `uint256` et `int256`, respectivement.

Opérateurs :
- Comparaisons : `<=`, `<`, `==`, `!=`, `>=`, `>` (retournent un `bool`)
- Opérateurs binaires : `&`, `|`, `^` (ou exclusif binaire), `~` (négation binaire)
- Opérateurs de décalage : `<<` (décalage vers la gauche), `>>` (décalage vers la droite)
- Opérateurs arithmétiques : `+`, `-`, l'' opérateur unaire `-`, `*`, `/`, `%` (modulo), `**` (exponentiation)

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

Avertissement : Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to `2**32 - 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 that you should *be aware of and mitigate against*.

Comparaisons

La valeur d'une comparaison est celle obtenue en comparant la valeur entière.

Opérations binaires

Les opérations binaires sont effectuées sur la représentation du nombre par *complément à deux<https ://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux>*. Cela signifie que, par exemple, `~int256(0) == int256(-1)`.

Décalages

- For positive and negative `x` values, `x << y` is equivalent to `x * 2**y`.
- For positive `x` values, `x >> y` is equivalent to `x / 2**y`.
- For negative `x` values, `x >> y` is equivalent to `(x + 1) / 2**y - 1` (which is the same as dividing `x` by `2**y` while rounding down towards negative infinity).
- In all cases, shifting by a negative `y` throws a runtime exception.

Décaler d'un nombre négatif de bits déclenche une exception.

Avertissement : Avant la version `0.5.0.0`, un décalage vers la droite `x >> y` pour un `x` négatif était équivalent à `x / 2**y`, c'est-à-dire que les décalages vers la droite étaient arrondis vers zéro plutôt que vers l'infini négatif.

Addition, Soustraction et Multiplication

L'addition, la soustraction et la multiplication ont la sémantique habituelle. Ils utilisent également la représentation du complément de deux, ce qui signifie, par exemple, que `uint256(0) - uint256(1) == 2**256 - 1`. Vous devez tenir compte de ces débordements (« overflows ») pour la conception de contrats sûrs.

L'expression `x` équivaut à `(T(0) - x)` où `T` est le type de `x`. Cela signifie que `-x` ne sera pas négatif si le type de `x` est un type entier non signé. De plus, `x` peut être positif si `x` est négatif. Il y a une autre mise en garde qui découle également de la représentation en compléments de deux :

```
int x = -2**255;
assert(-x == x);
```

Cela signifie que même si un nombre est négatif, vous ne pouvez pas supposer que sa négation sera positive.

Division

Puisque le type du résultat d'une opération est toujours le type d'un des opérandes, la division sur les entiers donne toujours un entier. Dans Solidity, la division s'arrondit vers zéro. Cela signifie que `int256(-5) / int256(2) == int256(-2)`.

Notez qu'en revanche, la division sur les littéraux donne des valeurs fractionnaires de précision arbitraire.

Note : La division par zéra cause un échec d"`assert`.

Modulo

L'opération modulo `a % n` donne le reste `r` après la division de l'opérande `a` par l'opérande `n`, où `q = int(a / n)` et `r = a - (n * q)`. Cela signifie que modulo donne le même signe que son opérande gauche (ou zéro) et `a % n == -(abs(a) % n)` est valable pour un `a` négatif :

— `int256(5) % int256(2) == int256(1)`
— `int256(5) % int256(-2) == int256(1)`
— `int256(-5) % int256(2) == int256(-1)`
— `int256(-5) % int256(-2) == int256(-1)`

Note : La division par zéra cause un échec d'`assert`.

Exponentiation

l'exponentiation n'est disponible que p[our les types signés. Veillez à ce que les types que vous utilisez soient suffisamment grands pour conserver le résultat et vous préparer à un éventuel effet d'enroulage (wrapping/int overflow).

Note : `0**0` est défini par l'EVM comme étant `1`.

Nombre à virgule fixe

Avertissement : Les numéros à point fixe ne sont pas encore entièrement pris en charge par Solidity. Ils peuvent être déclarés, mais ne peuvent pas être affectés à ou de.

`fixed` / `ufixed` : Nombre à virgule fixe signés et non-signés de taille variable. Les mots-clés `ufixedMxN` et `fixedMxN`, où `M` représente le nombre de bits pris par le type et `N` représente combien de décimales sont disponibles. `M` doit être divisible par 8 et peut aller de 8 à 256 bits. `N` doit être compris entre 0 et 80, inclusivement. `ufixed` et `fixed` sont des alias pour `ufixed128x18` et `fixed128x18`, respectivement.

Opérateurs :
— Comparaisons : <=, <, ==, !=, >=, > (évalue à `bool`)
— Operateurs arithmétiques : +, −, l'opérateur unaire −, *, /, % (modulo)

Note : La principale différence entre les nombres à virgule flottante (`float``et ``double` dans de nombreux langages, plus précisément les nombres IEEE 754) et les nombres à virgule fixe est que le nombre de bits utilisés pour l'entier et la partie fractionnaire (la partie après le point décimal) est flexible dans le premier, alors qu'il est strictement défini dans le second. Généralement, en virgule flottante, presque tout l'espace est utilisé pour représenter le nombre, alors que seul un petit nombre de bits définit où se trouve le point décimal.

Adresses

Le type d'adresse se décline en deux versions, qui sont en grande partie identiques :
— `address` : Contient une valeur de 20 octets (taille d'une adresse Ethereum).
— `address payable` : Même chose que « adresse », mais avec les membres additionnels `transfert` et `envoi`.
L'idée derrière cette distinction est que l'`address payable` est une adresse à laquelle vous pouvez envoyer de l'éther, alors qu'une simple `address` ne peut être envoyée de l'éther.

Conversions de type :

Les conversions implicites de `address payable` à `address` sont autorisées, tandis que les conversions de `address` à `address payable` ne sont pas possibles.

Note : La seule façon d'effectuer une telle conversion est d'utiliser une conversion intermédiaire en `uint160`.

Les adresses littérales peuvent être implicitement converties en `address payable`.

Les conversions explicites vers et à partir de `address` sont autorisées pour les entiers, les entiers littéraux, les `bytes20` et les types de contrats avec les réserves suivantes : Les conversions sous la forme `address payable(x)` ne sont pas permises. Au lieu de cela, le résultat d'une conversion sous forme `adresse(x)` donne une `address payable` si `x` est un contrat disposant d'une fonction par défaut (`fallback`) `payable`, ou si `x` est de type entier, bytes fixes, ou littéral. Sinon, l'adresse obtenue sera de type `address`. Dans les fonctions de signature externes, `address` est utilisé à la fois pour le type `address``et ``address payable`.

Note :

Il se peut fort bien que vous n'ayez pas à vous soucier de la distinction entre `address` et `address payable` et que vous u
`msg.sender`, qui est une `address payable`.

Opérateurs :
— <=, <, ==, !=, >= and >

Avertissement :

Si vous convertissez un type qui utilise une taille d'octet plus grande en `address`, par exemple `bytes32`, alors l'adresse e
Pour réduire l'ambiguïté de conversion à partir de la version 0.4.24 du compilateur vous force à rendre la troncature explicite dans la conversion. Prenons par exemple l'adresse `0x1111222222323333434444545555566666677777778888999999AAAABBBBBBCCDDDDEEFEFFFFFFCC`.
Vous pouvez utiliser `address(uint160(octets20(b)))`, ce qui donne `0x1111212222323333434444545555566667777888889999aAaaa`, ou vous pouvez utiliser `address(uint160(uint256(b)))`, ce qui donne `0x777777888888999999AaAAbBbbCcccddDdeeeEfFFfCcCcCc`.

Note :

La distinction entre `address``et ``address payable` a été introduite avec la version 0.5.0. À partir de cette version également, les contrats ne dérivent pas du type d'adresse, mais peuvent toujours être convertis explicitement en adresse » ou à » adresse payable « , s'ils ont une fonction par défaut payable.

Membres de Address

Pour une liste des membres de address, voir *Membres du type address*.
— `balance` et `transfer`.
Il est possible d'interroger le solde d'une adresse en utilisant la propriété `balance` et d'envoyer des Ether (en unités de wei) à une adresse payable à l'aide de la fonction `transfert` :

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

La fonction `transfer` échoue si le solde du contrat en cours n'est pas suffisant ou si le transfert d'Ether est rejeté par le compte destinataire. La fonction `transfert` s'inverse en cas d'échec.

Note : Si `x` est une adresse de contrat, son code (plus précisément : sa *Fonction de repli*, si présente) sera exécutée avec l'appel `transfer` (c'est une caractéristique de l'EVM et ne peut être empêché). Si cette exécution échoue ou s'il n'y a plus de gas, le transfert d'Ether sera annulé et le contrat en cours s'arrêtera avec une exception.

— `send`

`send` est la contrepartie de bas niveau du `transfer`. Si l'exécution échoue, le contrat en cours ne s'arrêtera pas avec une exception, mais `send` retournera `false`.

Avertissement : Il y a certains dangers à utiliser la fonction `send` : Le transfert échoue si la profondeur de la stack atteint 1024 (cela peut toujours être forcé par l'appelant) et il échoue également si le destinataire manque de gas. Donc, afin d'effectuer des transferts d'Ether en toute sécurité, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou mieux encore : utilisez un modèle où le destinataire retire l'argent.

— `call`, `delegatecall` et `staticcall`

Afin de s'interfacer avec des contrats qui ne respectent pas l'ABI, ou d'obtenir un contrôle plus direct sur l'encodage, les fonctions `call`, `delegatecall` et `staticcall` sont disponibles. Elles prennent tous pour argument un seul `bytes memory` comme entrée et retournent la condition de succès (en tant que `bool`) et les données (`bytes memory`). Les fonctions `abi.encoder`, `abi.encoderPacked`, `abi.encoderWithSelector` et `abi.encoderWithSignature` peuvent être utilisées pour coder des données structurées.

Exemple :

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

Avertissement :

Toutes ces fonctions sont des fonctions de bas niveau et doivent être utilisées avec précaution. Plus précisément, tout contrat inconnu peut être malveillant et si vous l'appelez, vous transférez le contrôle à ce contrat qui, à son tour, peut revenir dans votre contrat, donc soyez prêt à modifier les variables de votre état. quand l'appel revient. La façon habituelle d'interagir avec d'autres contrats est d'appeler une fonction sur un objet `contract` (`x.f()`)..

: : note : : Les versions précédentes de Solidity permettaient à ces fonctions de recevoir des arguments arbitraires et de traiter différemment un premier argument de type `bytes4`. Ces cas rares ont été supprimés dans la version 0.5.0.

Il est possible de régler le gas fourni avec le modificateur `.gas()` :

```
namReg.call.gas(1000000)(abi.encodeWithSignature("register(string)", "MyName"));
```

De même, la valeur en Ether fournie peut également être contrôlée : : :

```
nameReg.call.value(1 ether)(abi.encodeWithSignature("register(string)", "MyName"));
```

Enfin, ces modificateurs peuvent être combinés. Leur ordre n'a pas d'importance :

```
nameReg.call.gas(1000000).value(1 ether)(abi.encodeWithSignature("register(string)",
↪"MyName"));
```

De la même manière, la fonction `delegatecall` peut être utilisée : la différence est que seul le code de l'adresse donnée est utilisé, tous les autres aspects (stockage, balance,...) sont repris du contrat actuel. Le but de `delegatecall` est d'utiliser du code de bibliothèque qui est stocké dans un autre contrat. L'utilisateur doit s'assurer que la disposition du stockage dans les deux contrats est adaptée à l'utilisation de `delegatecall`.

Note : Avant Homestead, il n'existait qu'une variante limitée appelée `callcode` qui ne donnait pas accès aux valeurs originales `msg.sender` et `msg.value`. Cette fonction a été supprimée dans la version 0.5.0.

Depuis Byzantium, `staticcall` peut aussi être utilisé. C'est fondamentalement la même chose que `call`, mais reviendra en arrière si la fonction appelée modifie l'état d'une manière ou d'une autre.

Les trois fonctions `call`, `delegatecall``et ``staticcall` sont des fonctions de très bas niveau et ne devraient être utilisées qu'en *dernier recours* car elles brisent la sécurité de type de Solidity.

L'option `.gas()` est disponible sur les trois méthodes, tandis que l'option `.value()` n'est pas supportée pour `delegatecall`.

Note : Tous les contrats pouvant être convertis en type `address`, il est possible d'interroger le solde du contrat en cours en utilisant `address(this).balance`.

Types Contrat

Chaque *contrat* définit son propre type. Vous pouvez implicitement convertir des contrats en contrats dont ils héritent. Les contrats peuvent être explicitement convertis de et vers tous les autres types de contrats et le type `address`.

La conversion explicite vers et depuis le type `address payable` n'est possible que si le type de contrat dispose d'une fonction de repli payante. La conversion est toujours effectuée en utilisant `address(x)` et non `address payable(x)`. Vous trouverez plus d'informations dans la section sur le *type address*.

Note : Avant la version 0.5.0, les contrats dérivaient directement du type address et il n'y avait aucune distinction entre `address` et `address payable`.

Si vous déclarez une variable locale de type contrat (*MonContrat c*), vous pouvez appeler des fonctions sur ce contrat. Prenez bien soin de l'assigner à un contrat d'un type correspondant.

Vous pouvez également instancier les contrats (ce qui signifie qu'ils sont nouvellement créés). Vous trouverez plus de détails dans la section "contrats de création".

La représentation des données d'un contrat est identique à celle du type `address` et ce type est également utilisé dans l'*ABI*.

Les contrats ne supportent aucun opérateur.

Les membres du type contrat sont les fonctions externes du contrat, y compris les variables d'état publiques.

For a contract `C` you can use `type(C)` to access *type information* about the contract.

Tableaux d'octets de taille fixe

Les types valeur `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` contiennent une séquence de 1 à 32 octets. `byte` est un alias de `bytes1`.

Opérateurs :

— Comparaisons : <=, <, ==, !=, >=, > (retournent un `bool`)
— Opérateurs binaires : `&`, `|`, `^` (ou exclusif binaire), ~ (négation binaire)
— Opérateurs de décalage : << (décalage vers la gauche), >> (décalage vers la droite)
— Accès par indexage : Si `x` estd e type `bytesI`, alors `x[k]` pour `0 <= k < I` retourne le `k` ème byte (lecture seule).

L'opérateur de décalage travaille avec n'importe quel type d'entier comme opérande droite (mais retourne le type de l'opérande gauche), qui indique le nombre de bits à décaler. Le décalage d'un montant négatif entraîne une exception d'exécution.

Membres :

*`.length` donne la longueur fixe du tableau d'octets (lecture seule).

Note : Le type `byte[]` est un tableau d'octets, mais en raison des règles de bourrage, il gaspille 31 octets d'espace pour chaque élément (sauf en storage). Il est préférable d'utiliser le type « bytes » à la place.

Tableaux dynamiques d'octets

`bytes` : Tableau d'octets de taille dynamique, voir *Tableaux*. Ce n'est pas un type valeur !

`string` : Chaîne codée UTF-8 de taille dynamique, voir *Tableaux*. Ce n'est pas un type valeur !

Adresses Littérales

Les caractères hexadécimaux qui réussissent un test de somme de contrôle d'adresse (« address checksum »), par exemple `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` sont de type `address payable`. Les nombres hexadécimaux qui ont entre 39 et 41 chiffres et qui ne passent pas le test de somme de contrôle produisent un avertissement et sont traités comme des nombres rationnels littéraux réguliers.

Note : Le format de some de contrôle multi-casse est décrit dans EIP-55.

Rationels et entiers littéraux

Les nombres entiers littéraux sont formés à partir d'une séquence de nombres compris entre 0 et 9 interprétés en décimal. Par exemple, `69` signifie soixante-neuf. Les littéraux octaux n'existent pas dans Solidity et les zéros précédant un nombre sont invalides.

Les fractions décimales sont formées par un `.` avec au moins un chiffre sur un côté. Exemples : `1.1`, `.1 `` et ``1.3`.

La notation scientifique est également supportée, où la base peut avoir des fractions, alors que l'exposant ne le peut pas. Exemples : `2e10`, `-2e10`, `2e-10`, `2e-10`, `2.5e1`.

Les soulignements (underscore) peuvent être utilisés pour séparer les chiffres d'un nombre littéral numérique afin d'en faciliter la lecture. Par exemple, la décimale `123_000`, l'hexadécimale `0x2eff_abde`, la notation décimale scientifique `1_2e345_678` sont toutes valables. Les tirets de soulignement ne sont autorisés qu'entre deux chiffres et un seul tiret de soulignement consécutif est autorisé. Il n'y a pas de signification sémantique supplémentaire ajoutée à un nombre contenant des tirets de soulignement, les tirets de soulignement sont ignorés.

Les expressions littérales numériques conservent une précision arbitraire jusqu'à ce qu'elles soient converties en un type non littéral (c'est-à-dire en les utilisant avec une expression non littérale ou par une conversion explicite). Cela

signifie que les calculs ne débordent pas (overflow) et que les divisions ne tronquent pas les expressions littérales des nombres.

Par exemple, `(2**800 + 1) - 2**800` produit la constante `1` (de type `uint8`) bien que les résultats intermédiaires ne rentrent même pas dans la taille d'un mot machine. De plus, `.5 * 8` donne l'entier `4` (bien que des nombres non entiers aient été utilisés entre les deux).

N'importe quel opérateur qui peut être appliqué aux nombres entiers peut également être appliqué aux expressions littérales des nombres tant que les opérandes sont des nombres entiers. Si l'un des deux est fractionnaire, les opérations sur bits sont interdites et l'exponentiation est interdite si l'exposant est fractionnaire (parce que cela pourrait résulter en un nombre non rationnel).

Avertissement : La dvision d'entiers littéraux tronquait dans les versions de Solidity avant la version 0.4.0, mais elle donne maintenant en un nombre rationnel, c'est-à-dire que `5 / 2` n'est pas égal à `2`, mais à `2.5`.

Note :

Solidity a un type de nombre littéral pour chaque nombre rationnel. Les nombres entiers littéraux et les nombres rationnels appartiennent à des types de nombres littéraux. De plus, toutes les expressions numériques littérales (c'est-à-dire les expressions qui ne contiennent que des nombres et des opérateurs) appartiennent à des types littéraux de nombres. Ainsi, les expressions littérales `1 + 2` et `2 + 1` appartiennent toutes deux au même type littéral de nombre pour le nombre rationnel numéro trois.

Note : Les expressions littérales numériques sont converties en caractères non littéraux dès qu'elles sont utilisées avec des expressions non littérales. Indépendamment des types, la valeur de l'expression assignée à `b` ci-dessous est évaluée en entier. Comme `a` est de type `uint128`, l'expression `2,5 + a` doit cependant avoir un type. Puisqu'il n'y a pas de type commun pour les types `2.5` et `uint128`, le compilateur Solidity n'accepte pas ce code.

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

Chaines de caractères littérales

Les chaînes de caractères littérales sont écrites avec des guillemets simples ou doubles (`"foo"` ou `'bar'`). Elles n'impliquent pas de zéro final comme en C ; `foo` représente trois octets, pas quatre. Comme pour les entiers littéraux, leur type peut varier, mais ils sont implicitement convertibles en `bytes1`, ..., `bytes32`, ou s'ils conviennent, en `bytes` et en `string`.

Les chaînes de caractères littérales supportent les caractères d'échappement suivants :

- `\<newline>` (échappe un réel caractère newline)
- `\\` (barre oblique)
- `\'` (guillemet simple)
- `\"` (guillemet double)
- `\b` (backspace)
- `\f` (form feed)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tabulation horizontale)
- `\v` (tabulation verticale)
- `\xNN` (hex escape, see below)

— `\uNNNN` (echapement d'unicode, voir ci-dessous)

`\xNN` prend une valeur hexadécimale et insère l'octet approprié, tandis que `\uNNNNN` prend un codepoint Unicode et insère une séquence UTF-8.

La chaîne de caractères de l'exemple suivant a une longueur de dix octets. Elle commence par un octet de newline, suivi d'une guillemet double, d'une guillemet simple, d'un caractère barre oblique inversée et ensuite (sans séparateur) de la séquence de caractères `abcdef`.

```
"\n\"\'\\abc\
def"
```

Tout terminateur de ligne unicode qui n'est pas une nouvelle ligne (i.e. LF, VF, FF, CR, NEL, LS, PS) est considéré comme terminant la chaîne littérale. Newline ne termine la chaîne littérale que si elle n'est pas précédée d'un `\`.

Hexadécimaux littéraux

Les caractères hexadécimaux sont précédées du mot-clé `hex` et sont entourées de guillemets simples ou doubles (`hex"001122FF"`). Leur contenu doit être une chaîne hexadécimale et leur valeur sera la représentation binaire de ces valeurs.

Les littéraux hexadécimaux se comportent comme *chaînes de caractères littérales* et ont les mêmes restrictions de convertibilité.

Énumérateurs

Les `enum` sont une façon de créer un type défini par l'utilisateur en Solidity. Ils sont explicitement convertibles de et vers tous les types d'entiers mais la conversion implicite n'est pas autorisée. La conversion explicite à partir d'un nombre entier vérifie au moment de l'exécution que la valeur se trouve à l'intérieur de la plage de l'enum et provoque une affirmation d'échec autrement. Un enum a besoin d'au moins un membre.

La représentation des données est la même que pour les énumérations en C : Les options sont représentées par des valeurs entières non signées à partir de `0`.

```
pragma solidity >=0.4.16 <0.6.0;

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

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

    // Comme le type enum ne fait pas partie de l' ABI, la signature de "getChoice"
    // sera automatoquement changée en "getChoice() returns (uint8)"
    // pour ce qui sort de Solidity. Le type entier utilisé est
    // assez grand pour contenir toutes valeurs, par exemple si vous en avez
    // plus de 256, ``uint16`` sera utilisé etc...
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Types Fonction

Les types fonction sont les types des fonctions. Les variables du type fonction peuvent être passés et retournés pour transférer les fonctions vers et renvoyer les fonctions des appels de fonction. Les types de fonctions se déclinent en deux versions : les fonctions *internes* `internal` et les fonctions *externes* `external` :

Les fonctions internes ne peuvent être appelées qu'à l'intérieur du contrat en cours (plus précisément, à l'intérieur de l'unité de code en cours, qui comprend également les fonctions de bibliothèque internes et les fonctions héritées) car elles ne peuvent pas être exécutées en dehors du contexte du contrat actuel. L'appel d'une fonction interne est réalisé en sautant à son label d'entrée, tout comme lors de l'appel interne d'une fonction du contrat en cours.

Les fonctions externes se composent d'une adresse et d'une signature de fonction et peuvent être transférées et renvoyées à partir des appels de fonction externes.

Les types de fonctions sont notés comme suit : :

fonction (<types de paramètres>) {internallexternal} {purelviewlpayable][returns (<types de retour>)]

En contraste avec types de paramètres, les types de retour ne peuvent pas être vides - si le type de fonction ne retourne rien, toute la partie ‘‘returns (<types de retour>)‘‘doit être omise.

Par défaut, les fonctions sont de type `internal`, donc le mot-clé `internal` peut être omis. Notez que ceci ne s'applique qu'aux types de fonctions. La visibilité doit être spécifiée explicitement car les fonctions définies dans les contrats n'ont pas de valeur par défaut.

Conversions :

Une fonction de type `external` peut être explicitement convertie en `address` résultant en l'adresse du contrat de la fonction.

Un type de fonction `A` est implicitement convertible en un type de fonction `B` si et seulement si leurs types de paramètres sont identiques, leurs types de retour sont identiques, leurs propriétés internal/external sont identiques et la mutabilité d'état de `A` n'est pas plus restrictive que la mutabilité de l'état de `B`. En particulier :

- Les fonctions `pure` peuvent être converties en fonctions `view` et `non-payable`.
- Les fonctions `view` peuvent être converties en fonctions `non-payable`.
- les fonctions `payable` peuvent être converties en fonctions `non-payable`.

Aucune autre conversion entre les types de fonction n'est possible.

La règle concernant les fonctions `payable` et `non-payable` peut prêter à confusion, mais essentiellement, si une fonction est `payable`, cela signifie qu'elle accepte aussi un paiement de zéro Ether, donc elle est également `non-payable`. D'autre part, une fonction `non-payable` rejettera l'Ether qui lui est envoyé, de sorte que les fonctions `non-payable` ne peuvent pas être converties en fonctions `payable`.

Si une variable de type fonction n'est pas initialisée, l'appel de celle-ci entraîne l'échec d'une assertion. Il en va de même si vous appelez une fonction après avoir utilisé `delete` dessus.

Si des fonctions de type `external` sont appelées d'en dehors du contexte de Solidity, ils sont traités comme le type `function`, qui code l'adresse suivie de l'identificateur de fonction ensemble dans un seul type `bytes24`.

Notez que les fonctions publiques du contrat actuel peuvent être utilisées à la fois comme une fonction interne et comme une fonction externe. Pour utiliser `f` comme fonction interne, utilisez simplement `f`, si vous voulez utiliser sa forme externe, utilisez `this.f`.

Members :

External (or public) functions have the following members :

- `.address` returns the address of the contract of the function.

— `.selector` returns the *ABI function selector*
— `.gas(uint)` returns a callable function object which, when called, will send the specified amount of gas to the target function. Deprecated - use `{gas: ...}` instead. See *External Function Calls* for more information.
— `.value(uint)` returns a callable function object which, when called, will send the specified amount of wei to the target function. Deprecated - use `{value: ...}` instead. See *External Function Calls* for more information.

Example that shows how to use the members :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;
// This will report a warning

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
        this.f.gas(10).value(800)();
        // New syntax:
        // this.f{gas: 10, value: 800}()
    }
}
```

Exemple d'utilisation des fonctions de type `internal` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

library ArrayUtils {
  // les fonctions internes peuvent être utilisées dams des fonctions
  // de librairies internes car elles partagent le même contexte
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        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, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function range(uint length) internal pure 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) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }

    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

Exemple d" usage de fonction `external` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;


contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST = Oracle(0x1234567); // known contract
    uint private exchangeRate;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
}
```

Note : Les fonctions lambda ou en in-line sont prévues mais pas encore prises en charge.

3.6.2 Types Référence

Les valeurs du type référence peuvent être modifiées par plusieurs noms différents. Comparez ceci avec les catégories de valeurs où vous obtenez une copie indépendante chaque fois qu'une variable de valeur est utilisée. Pour cette raison, les types référence doivent être traités avec plus d'attention que les types de valeur. Actuellement, les types référence comprennent les structures, les tableaux et les mappages. Si vous utilisez un type référence, vous devez toujours indiquer explicitement la zone de données où le type est enregistré : (dont la durée de vie est limitée à un appel de fonction), `storage` (l'emplacement où les variables d'état sont stockées) ou `calldata` (emplacement de données spécial qui contient les arguments de fonction, disponible uniquement pour les paramètres d'appel de fonction externe).

Une affectation ou une conversion de type qui modifie l'emplacement des données entraîne toujours une opération de copie automatique, alors que les affectations à l'intérieur du même emplacement de données ne copient que dans certains cas selon le type de stockage.

Emplacement des données

Chaque type référence, c'est-à-dire *arrays* (tableaux) et *structs*, comporte une annotation supplémentaire, la `localisation des données`, indiquant où elles sont stockées. Il y a trois emplacements de données : `Memory`, `Storage` et `Calldata`. Calldata n'est valable que pour les paramètres des fonctions de contrat externes et n'est nécessaire que pour ce type de paramètre. Calldata est une zone non modifiable, non persistante où les arguments de fonction sont stockés, et se comporte principalement comme memory.

Note : Avant la version 0.5.0, l'emplacement des données pouvait être omis, et était par défaut à des emplacements différents selon le type de variable, le type de fonction, etc.

Data location and assignment behaviour

La localisation des données n'est sont pas seulement pertinente pour la persistance des données, mais aussi pour la sémantique des affectations :

— Les affectations entre le stockage et la mémoire (ou à partir des données de la calldata) créent toujours une copie indépendante.

— Les affectations de mémoire à mémoire ne créent que des références. Cela signifie que les modifications d'une variable mémoire sont également visibles dans toutes les autres variables mémoire qui se réfèrent aux mêmes données.
— Les affectations du stockage à une variable de stockage local n'affectent également qu'une référence.
— En revanche, toutes les autres affectations au stockage sont toujours copiées. Les affectations à des variables d'état ou à des membres de variables locales de type structure de stockage, même si la variable locale elle-même n'est qu'une référence, constituent des exemples dans ce cas.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is␣
↪storage
        y[7]; // fine, returns the 8th element
        y.pop(); // 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) internal pure {}
    function h(uint[] memory) public pure {}
}
```

Tableaux

Les tableaux peuvent avoir une taille fixe à la compilation ou peuvent être dynamiques.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

Un tableau de taille fixe `k` et de type d'élément `T` est écrit `T[k]`, un tableau de taille dynamique `T[]`.

Par exemple, un tableau de 5 tableaux dynamiques de `uint` est `uint[][5]` (notez que la notation est inversée par rapport à certains autres langages). Pour accéder au deuxième uint du troisième tableau dynamique, vous utilisez `x[2][1]` (les indexs commencent à zéro et l'accès fonctionne dans le sens inverse de la déclaration, c'est-à-dire que `x[2]` supprime un niveau dans le type de déclaration à partir de la droite).

Indices are zero-based, and access is in the opposite direction of the declaration.

Il y a peu de restrictions concernant l'élément contenu, il peut aussi être un autre tableau, un mappage ou une structure. Les restrictions générales s'appliquent, cependant, en ce sens que les mappages ne peuvent être utilisés que dans le `storage` et que les fonctions visibles au public nécessitent des paramètres qui sont des types reconnus par l'"*ABI types*.

Il est possible de marquer les tableaux `public` et de demander à Solidity de créer un *getter*. L'index numérique deviendra un paramètre obligatoire pour le getter.

L'accès à un tableau après sa fin provoque un `revert`. Si vous voulez ajouter de nouveaux éléments, vous devez utiliser `.push()` ou augmenter le membre `.length` (voir ci-dessous).

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of the array, where `.push()` appends a zero-initialized element and returns a reference to it.

`bytes` and `strings` as Arrays

Les variables de type `bytes` et `string` sont des tableaux spéciaux. Un `byte` est semblable à un `byte[]`, mais il est condensé en calldata et en mémoire. `string` est égal à `bytes`, mais ne permet pas l'accès à la longueur ou à l'index.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `abi.encodePacked(s1, s2)`.

Il faut donc généralement préférer les `bytes` aux `bytes[]` car c'est moins cher à l'usage. En règle générale, utilisez `bytes` pour les données en octets bruts de longueur arbitraire et `string` pour les données de chaîne de caractères de longueur arbitraire (UTF-8). Si vous pouvez limiter la longueur à un certain nombre d'octets, utilisez toujours un des `bytes1` à `bytes32`, car ils sont beaucoup moins chers également.

Note : Si vous voulez accéder à la représentation en octets d'une chaîne de caractères `s`, utilisez `bytes(s).length` / `bytes(s)[7] ='x';`. Gardez à l'esprit que vous accédez aux octets de bas niveau de la représentation UTF-8, et non aux caractères individuels !

Allouer des tableaux en mémoire

Vous pouvez utiliser le mot-clé `new` pour créer des tableaux dont la longueur dépend de la durée d'exécution en mémoire. Contrairement aux tableaux de stockage, il n'est **pas** possible de redimensionner les tableaux de mémoire (par exemple en les assignant au membre `.length`). Vous devez soit calculer la taille requise à l'avance, soit créer un nouveau tableau de mémoire et copier chaque élément.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

Tableaux littéraux / Inline Arrays

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`). For example `[1, a, f(3)]`. There must be a common type all elements can be implicitly converted to. This is the elementary type of the array.

Array literals are always statically-sized memory arrays.

Le type d'un tableau littéral est un tableau mémoire de taille fixe dont le type de base est le type commun des éléments donnés. Le type de `[1, 2, 3]` est `uint8[3] memory`, car le type de chacune de ces constantes est `uint8`. Pour cette raison, il est nécessaire de convertir le premier élément de l'exemple ci-dessus en `uint`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

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

Les tableaux de taille fixe ne peuvent pas être assignées à des tableaux de taille dynamique, c'est-à-dire que ce qui suit n'est pas possible :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

// Ceci ne compile pas.
contract C {
    function f() public {
        // La ligne suivant provoque une erreur car uint[3] memory
        // ne peut pas être convertit en uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

Il est prévu de supprimer cette restriction à l'avenir, mais crée actuellement certaines complications en raison de la façon dont les tableaux sont transmis dans l'ABI.

Array Members

length :

Les tableaux ont un membre `length` qui contient leur nombre d'éléments. La longueur des tableaux memory est fixe (mais dynamique, c'est-à-dire qu'elle peut dépendre des paramètres d'exécution) une fois qu'ils sont créés.

push() : Les tableaux de stockage dynamique et les `bytes` (et non `string`) ont une fonction membre appelée `push` que vous pouvez utiliser pour ajouter un élément à la fin du tableau. L'élément sera mis à zéro à l'initialisation. La fonction renvoie la nouvelle longueur.

push(x) : Dynamic storage arrays and `bytes` (not `string`) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

pop : Les tableaux de stockage dynamique et les `bytes` (et non `string`) ont une fonction membre appelée `pop` que vous pouvez utiliser pour supprimer un élément à la fin du tableau. Ceci appelle aussi implicitement :ref :`delete` sur l'élément supprimé.

Note : L'augmentation de la longueur d'un tableau en storage a des coûts en gas constants parce qu'on suppose que le stockage est nul, alors que la diminution de la longueur a au moins un coût linéaire (mais dans la plupart des cas pire que linéaire), parce qu'elle inclut explicitement l'élimination des éléments supprimés comme si on appelait :ref :`delete`.

Note : To use arrays of arrays in external (instead of public) functions, you need to activate ABIEncoderV2.

Note : Dans les versions EVM antérieures à Byzantium, il n'était pas possible d'accéder au retour de tableaux dynamique à partir des appels de fonctions. Si vous appelez des fonctions qui retournent des tableaux dynamiques, assurez-vous d'utiliser un EVM qui est configuré en mode Byzantium.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Notez que ce qui suit n'est pas une paire de tableaux dynamiques
    // mais un tableau tableau dynamique de paires (c'est-à-dire de
    // tableaux de taille fixe de longueur deux).
    // Pour cette raison, T[] est toujours un tableau dynamique
    // de T, même si T lui-même est un tableau.
    // L'emplacement des données pour toutes les variables d'état
    // est storage.
    bool[2][] m_pairsOfFlags;

    // newPairs est stocké en memory - seule possibilité
    // pour les arguments de fonction publique
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // l'assignation d' un tableau en storage implique la copie
        // de  ``newPairs`` et remplace l'array ``m_pairsOfFlags``.
        m_pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // stocke un pointeur sur ``s`` dans ``g``
        StructType storage g = s;
        // change aussi ``s.moreInfo``.
        g.moreInfo = 2;
        // assigne une copie car ``g.contents`` n'est
        // pas une variable locale mais un membre
        // d'une variable locale
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // accès à un index inexistant, déclenche une exception
```

(suite sur la page suivante)

(suite de la page précédente)

```
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // using push and pop is the only way to change the
        // length of an array
        if (newSize < m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length > newSize)
                m_pairsOfFlags.pop();
        } else if (newSize > m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length < newSize)
                m_pairsOfFlags.push();
        }
    }

    function clear() public {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags = new bool[2][](0);
    }

    bytes m_byteData;

    function byteArrays(bytes memory data) public {
        // le tableau de byte ("bytes") sont différents car stockés sans
        // padding mais peuvent être traités comme des ``uint8[]``
        m_byteData = data;
        for (uint i = 0; i < 7; i++)
            m_byteData.push();
        m_byteData[3] = 0x08;
        delete m_byteData[2];
    }

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

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Un tableau dynamique est créé via `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Les tableaux littéraux sont toujours de taille statique
        // et en cas d' utilisation de littéraux uniquement, au moins
        // un type doit être spécifié.
        arrayOfPairs[0] = [uint(1), 2];

        // Créée un tableau dynamique de bytes:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(uint8(i));
        return b;
    }
}
```

Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional : `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

Note : As of now, array slices are only implemented for calldata arrays.

Array slices are useful to ABI-decode secondary data passed in function parameters :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Proxy {
    /// Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address _client) public {
        client = _client;
    }

    /// Forward call to "setOwner(address)" that is implemented by client
    /// after doing basic validation on the address argument.
    function forward(bytes calldata _payload) external {
        bytes4 sig = abi.decode(_payload[:4], (bytes4));
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(_payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(_payload);
        require(status, "Forwarded call failed.");
    }
}
```

Structs

Solidity permet de définir de nouveaux types sous forme de structs, comme le montre l'exemple suivant :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    address addr;
    uint amount;
}

contract CrowdFunding {
    // Structs can also be defined inside contracts, which makes them
    // visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

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

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // Creates new struct in memory and copies it to storage.
        // We leave out the mapping type, because it is not valid in memory.
        // If structs are copied (even from storage to storage),
        // types that are not valid outside of storage (ex. mappings and array of␣
↪mappings)
        // are always omitted, because they cannot be enumerated.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage 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) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

Le contrat ne fournit pas toutes les fonctionnalités d'un contrat de crowdfunding, mais il contient les concepts de base nécessaires pour comprendre les `struct`. Les types structs peuvent être utilisés à l'intérieur des `mapping` et des `array` et peuvent eux-mêmes contenir des mappages et des tableaux.

Il n'est pas possible pour une structure de contenir un membre de son propre type, bien que la structure elle-même puisse être le type de valeur d'un membre de mappage ou peut contenir un tableau de taille dynamique de son type. Cette restriction est nécessaire, car la taille de la structure doit être finie.

Notez que dans toutes les fonctions, un type structure est affecté à une variable locale avec l'emplacement de données `storage`. Ceci ne copie pas la structure mais stocke seulement une référence pour que les affectations aux membres de la variable locale écrivent réellement dans l'état.

Bien sûr, vous pouvez aussi accéder directement aux membres de la structure sans l'affecter à une variable locale, comme dans `campaigns[campaignID].amount = 0`.

Mappages

Vous déclarez le objets de type `mapping` avec la syntaxe `mapping(_KeyType => _ValueType)`. `_KeyType` peut être n'importe quel type élémentaire. Cela signifie qu'il peut s'agir de n'importe lequel des types de valeurs intégrés plus les octets et les chaînes de caractères. Les types définis par l'utilisateur ou les types complexes tels que les types de contrat, les énuménations, les mappages, les structs et tout type de tableau, à l'exception des `bytes` et des `string` qui ne sont pas autorisés. `_ValueType` peut être n'importe quel type, y compris les mappages.

Vous pouvez considérer les mappings comme des tables de hashage, qui sont virtuellement initialisées de telle sorte que chaque clé possible existe et est mappée à une valeur dont la représentation binaire est constituée de zéros, de type *valeur par défaut*. La similitude s'arrête là, les données « clés » ne sont pas stockées dans un mappage, seul son hachage `keccak256` est utilisé pour rechercher la valeur.

Pour cette raison, les mappages n'ont pas de longueur ou de concept de clé ou de valeur définie.

Les mappages ne peuvent avoir qu'un emplacement de données en `storage` et sont donc autorisés pour les variables d'état, comme types référence en storage dans les fonctions ou comme paramètres pour les fonctions de librairies. Ils ne peuvent pas être utilisés comme paramètres ou paramètres de retour de fonctions de contrat publiques.

Vous pouvez marquer les variables de type mapping comme `public` et Solidity crée un *getter* pour vous. Le `` `_KeyType `` devient un paramètre pour le getter. Si `_ValueType` est un type de valeur ou une structure, le getter retourne `_ValueType`. Si `_ValueType` est un tableau ou un mappage, le getter a un paramètre pour chaque `_KeyType`, de manière récursive.

In the example below, the `MappingExample` contract defines a public `balances` mapping, with the key type an `address`, and a value type a `uint`, mapping an Ethereum address to an unsigned integer value. As `uint` is a value type, the getter returns a value that matches the type, which you can see in the `MappingUser` contract that returns the value at the specified address.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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

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

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

The example below is a simplified version of an ERC20 token. `_allowances` is an example of a mapping type inside another mapping type. The example below uses `_allowances` to record the amount someone else is allowed to withdraw from your account.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public␣
↪returns (bool) {
        _transfer(sender, recipient, amount);
        approve(sender, msg.sender, amount);
        return true;
    }

    function approve(address owner, address spender, uint256 amount) public returns␣
↪(bool) {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

Iterable Mappings

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that. For example, the code below implements an `IterableMapping` library that the `User` contract then adds data too, and the `sum` function iterates over to sum all the values.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    KeyFlag[] keys;
    uint size;
}

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterate_start(itmap storage self) internal view returns (uint keyIndex) {
        return iterate_next(self, uint(-1));
    }

    function iterate_valid(itmap storage self, uint keyIndex) internal view returns␣
↪(bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint keyIndex) internal view returns␣
↪(uint r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }

    function iterate_get(itmap storage self, uint keyIndex) internal view returns␣
↪(uint key, uint value) {
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```

3.6.3 Opérateurs impliquant des LValues

Si `a` est une LValue (c.-à-d. une variable ou quelque chose qui peut être assigné à), les opérateurs suivants sont disponibles en version raccourcie :

```
``a += e`` équivaut à ``a = a + e``. Les opérateurs ``-=``, ``*=``, ``/=``, ``%=``,␣
↪``|=``, ``&=`` et ``^=`` sont définis de la même manière. ``a++`` et ``a--`` sont␣
↪équivalents à ``a += 1`` / ``a -= 1`` mais l'expression elle-même a toujours la␣
↪valeur précédente ``a``. Par contraste, ``--a`` et ``++a`` changent également ``a``␣
↪de ``1`` , mais retournent la valeur après le changement.
```

delete

`delete a` affecte la valeur initiale du type à `a`. C'est-à-dire que pour les entiers, il est équivalent à `a = 0`, mais il peut aussi être utilisé sur les tableaux, où il assigne un tableau dynamique de longueur zéro ou un tableau statique de la même longueur avec tous les éléments initialisés à leur valeur par défaut. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a *mapping* is probably a better choice.

Pour les structs, il assigne une structure avec tous les membres réinitialisés. En d'autres termes, la valeur de `a` après `delete a` est la même que si `a` était déclaré sans attribution, avec la réserve suivante :

`delete` n'a aucun effet sur les mappages (car les clés des mappages peuvent être arbitraires et sont généralement inconnues). Ainsi, si vous supprimez une structure, elle réinitialisera tous les membres qui ne sont pas des `mappings`

et se propagera récursivement dans les membres à moins qu'ils ne soient des mappings. Toutefois, il est possible de supprimer des clés individuelles et ce à quoi elles correspondent : Si `a` est un mappage, alors `delete a[x]` supprimera la valeur stockée à `x`.

Il est important de noter que `delete a` se comporte vraiment comme une affectation à `a`, c'est-à-dire qu'il stocke un nouvel objet dans `a`. Cette distinction est visible lorsque `a` est une variable par référence : Il ne réinitialisera que `a` lui-même, et non la valeur à laquelle il se référait précédemment.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // met x à 0, n' affecte pas data
        delete data; // met data à 0, n'affecte pas x
        uint[] storage y = dataArray;
        delete dataArray; // ceci met dataArray.length à zéro, mais un uint[]
        // est un objet complexe, donc y est affecté est un alias
        // vers l' objet en storage.
        // D' un autre côté: "delete y" est invalid, car l' assignement à
        // une variable locale pointant vers un objet en storage n' est
        // autorisée que depuis un objet en storage.
        assert(y.length == 0);
    }
}
```

3.6.4 Conversions entre les types élémentaires

Conversions implicites

An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. En général, une conversion implicite entre les types valeur est possible si elle a un sens sémantique et qu'aucune information n'est perdue.

Par exemple, `uint8` est convertible en `uint16` et `int128` en `int256`, mais `uint8` n'est pas convertible en `uint256` (car `uint256` ne peut contenir, par exemple, `-1`)

Si un opérateur est appliqué à différents types, le compilateur essaie de convertir implicitement l'un des opérandes au type de l'autre (c'est la même chose pour les assignations). This means that operations are always performed in the type of one of the operands.

For more details about which implicit conversions are possible, please consult the sections about the types themselves.

In the example below, `y` and `z`, the operands of the addition, do not have the same type, but `uint8` can be implicitly converted to `uint16` and not vice-versa. Because of that, `y` is converted to the type of `z` before the addition is performed in the `uint16` type. The resulting type of the expression `y + z` is `uint16`. Because it is assigned to a variable of type ``uint32` another implicit conversion is performed after the addition.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

Conversions explicites

Si le compilateur ne permet pas la conversion implicite mais que vous savez ce que vous faites, une conversion de type explicite est parfois possible. Notez que cela peut vous donner un comportement inattendu et vous permet de contourner certaines fonctions de sécurité du compilateur, donc assurez-vous de tester que le résultat est ce que vous voulez !

Prenons l'exemple suivant où l'on convertit un `int8` négatif en un `uint` :

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

A la fin de cet extrait de code, `x` aura la valeur `0xfffffff...fd` (64 caractères hexadécimaux), qui est -3 dans la représentation en 256 bits du complément à deux.

Si un entier est explicitement converti en un type plus petit, les bits d'ordre supérieur sont coupés :

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b sera désormais 0x5678
```

Si un entier est explicitement converti en un type plus grand, il est rembourré par la gauche (c'est-à-dire à l'extrémité supérieure de l'ordre). Le résultat de la conversion sera comparé à l'entier original :

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Les types à taille fixe se comportent différemment lors des conversions. Ils peuvent être considérés comme des séquences d'octets individuels et la conversion à un type plus petit coupera la séquence :

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b sera désormais 0x12
```

Si un type à taille fixe est explicitement converti en un type plus grand, il est rembourré à droite. L'accès à l'octet par un index fixe donnera la même valeur avant et après la conversion (si l'index est toujours dans la plage) :

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b sera désormais 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Puisque les entiers et les tableaux d'octets de taille fixe se comportent différemment lorsqu'ils sont tronqués ou rembourrés, les conversions explicites entre entiers et tableaux d'octets de taille fixe ne sont autorisées que si les deux ont la même taille. Si vous voulez convertir entre des entiers et des tableaux d'octets de taille fixe de tailles différentes, vous devez utiliser des conversions intermédiaires qui font la troncature et le remplissage désirés. règles explicites :

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b sera désormais 0x00001234
uint32 c = uint32(bytes4(a)); // c sera désormais 0x12340000
uint8 d = uint8(uint16(a)); // d sera désormais 0x34
uint8 e = uint8(bytes1(a)); // d sera désormais 0x12
```

3.6.5 Conversions entre les types littéraux et élémentaires

Types nombres entiers

Les nombres décimaux et hexadécimaux peuvent être implicitement convertis en n'importe quel type entier suffisamment grand pour le représenter sans troncature :

```
uint8 a = 12; // Bon
uint32 b = 1234; // Bon
uint16 c = 0x123456; // échoue, car devrait tronquer en 0x3456
```

Tableaux d'octets de taille fixe

Les nombres décimaux ne peuvent pas être implicitement convertis en tableaux d'octets de taille fixe. Les nombres hexadécimaux peuvent être littéraux, mais seulement si le nombre de chiffres hexadécimaux correspond exactement à la taille du type de `bytes`. Par exception, les nombres décimaux et hexadécimaux ayant une valeur de zéro peuvent être convertis en n'importe quel type à taille fixe :

```
bytes2 a = 54321; // pas autorisé
bytes2 b = 0x12; // pas autorisé
bytes2 c = 0x123; // pas autorisé
bytes2 d = 0x1234; // bon
bytes2 e = 0x0012; // bon
bytes4 f = 0; // bon
bytes4 g = 0x0; // bon
```

Les littéraux de chaînes de caractères et les littéraux de chaînes hexadécimales peuvent être implicitement convertis en tableaux d'octets de taille fixe, si leur nombre de caractères correspond à la taille du type `bytes` :

```
bytes2 a = hex"1234"; // bon
bytes2 b = "xy"; // bon
bytes2 c = hex"12"; // pas autorisé
bytes2 d = hex"123"; // pas autorisé
bytes2 e = "x"; // pas autorisé
bytes2 f = "xyz"; // débile
```

Adresses

Comme décrit dans *Adresses Littérales*, les chaines de caractères hexadécimaux de la bonne taille qui passent le test de somme de contrôle sont de type `address`. Aucun autre littéral ne peut être implicitement converti au type `address`.

Les conversions explicites de `bytes20` ou de tout type entier en `address` aboutissent en une `address payable`.

3.7 Unités et variables globales

3.7.1 Unités d'Ether

Un nombre littéral peut prendre un suffixe de « wei », « finney », « szabo » ou « ether » pour spécifier une sous-dénomination d'éther, où les nombres d'éther sans postfix sont supposés être Wei.

```
assert(1 wei == 1);
assert(1 szabo == 1e12);
```

(suite sur la page suivante)

(suite de la page précédente)

```
assert(1 finney == 1e15);
assert(1 ether == 1e18);
```

Le seul effet du suffixe de sous-dénomination est une multiplication par une puissance de dix..

3.7.2 Unités de temps

Des suffixes comme `seconds`, `minutes`, `hours`, `days` et `weeks` peuvent être utilisés après les nombres littéraux pour spécifier les unités de temps où les secondes sont l'unité de base et les unités sont considérées naïvement de la façon suivante :

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

Faites attention si vous effectuez des calculs calendaires en utilisant ces unités, parce que chaque année n'est pas égale à 365 jours ni même chaque jour n'a 24 heures à cause des secondes bissextiles. Les secondes intercalaires étant imprévisibles, une bibliothèque de calendrier exacte doit être mise à jour par un oracle externe.

Note : Le suffixe `years` a été supprimé dans la version 0.5.0 pour les raisons ci-dessus.

Ces suffixes ne peuvent pas être appliqués aux variables. Si vous voulez interpréter une variable d'entrée en jours, par exemple, vous pouvez le faire de la manière suivante :

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

3.7.3 Variables spéciales et fonctions

Il y a des variables et des fonctions spéciales qui existent toujours dans l'espace de nommage global et qui sont principalement utilisées pour fournir des informations sur la chaîne de blocs ou sont des fonctions utilitaires générales.

Propriétés du bloc et des transactions

- `blockhash(uint blockNumber) returns (bytes32)` : hash du numéro de bloc passé - mnarche seulement pour les 256 plus récents, excluant le bloc courant
- `block.coinbase` (`address payable`) : addresse du mineur du bloc courant
- `block.difficulty` (`uint`) : difficulté du bloc courant
- `block.gaslimit` (`uint`) : limite de gas actuelle
- `block.number` (`uint`) : numéro du bloc courant
- `block.timestamp` (`uint`) : timestamp du bloc en temps unix (secondes)
- `gasleft() returns (uint256)` : gas restant
- `msg.data` (`bytes calldata`) : calldata complet
- `msg.sender` (`address payable`) : expéditeur du message (appel courant)
- `msg.sig` (`bytes4`) : 4 premiers octets calldata (i.e. identifiant de function)
- `msg.value` (`uint`) : nombre de wei envoyés avec le message
- `now` (`uint`) : alias pour `block.timestamp`

— `tx.gasprice` (`uint`) : prix de la transaction en gas
— `tx.origin` (`address payable`) : expéditeur de la transaction (appel global complet)

Note : Les valeurs de tous les membres de `msg`, y compris `msg.sender``et ``msg.value` peuvent changer pour chaque appel de fonction **external**. Ceci inclut les appels aux fonctions de librairies.

Note : Ne vous basez pas sur `block.timestamp`, `now` ou `blockhash` comme source de hasard, à moins de savoir ce que vous faites.

L'horodatage et le hashage du bloc peuvent tous deux être influencés dans une certaine mesure par les mineurs. Les mauvais acteurs de la communauté minière peuvent par exemple exécuter une fonction de casino sur un hash choisi et simplement réessayer un hash différent s'ils n'ont pas reçu d'argent.

L'horodatage du bloc courant doit être strictement supérieur à celui du dernier bloc, mais la seule garantie est qu'il se situera entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

Note : Les hashs de blocs ne sont pas disponibles pour tous les blocs pour des raisons d'évolutivité/place. Vous ne pouvez accéder qu'aux hachages des 256 blocs les plus récents, toutes les autres valeurs seront nulles.

Note : La fonction `blockhash` était auparavant connue sous le nom `block.blockhash`. Elle a été dépréciée dans la version 0.4.22 et supprimée dans la version 0.5.0.

Note : La fonction `gasleft` était auparavant connue sous le nom de `msg.gas`. Elle a été dépréciée dans la version 0.4.21 et supprimée dans la version 0.5.0.

Fonctions d'encodage et de décodage de l'ABI

— `abi.decode(bytes memory encodedData, (...)) returns (...)` : l'ABI décode les données données, tandis que les types sont donnés entre parenthèses comme second argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
— `abi.encode(...) returns (bytes memory)` : l'ABI encode les arguments passés.
— `abi.encodePacked(...) returns (bytes memory)` : exécute l'*encodage structuré* des arguments donnés
— `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : l'ABI-encode les arguments donnés à partir du second et précède le sélecteur des quatre octets donnés.
— `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : équivalent à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`

Note : Ces fonctions d'encodage peuvent être utilisées pour créer des données pour des appels de fonctions externes sans réellement appeler une fonction externe. De plus, `keccak256(abi.encododePacked(a, b))` est un moyen de calculer le hash des données structurées (bien qu'il soit possible de créer une `collision de hachage` en utilisant différents types d'entrées).

See the documentation about the *ABI* and the *tightly packed encoding* for details about the encoding.

Gestion des erreurs

Voir la section dédiée sur *assert and require* pour plus de détails sur la gestion des erreurs et quand utiliser quelle fonction.

assert(bool condition) : entraîne l'utilisation d'un opcode invalide et donc la réversion du changement d'état si la condition n'est pas remplie - à utiliser pour les erreurs internes.

require(bool condition) : `revert` si la condition n'est pas remplie - à utiliser en cas d'erreurs dans les entrées ou les composants externes.

require(bool condition, string memory message) : `revert` si la condition n'est pas remplie - à utiliser en cas d'erreurs dans les entrées ou les composants externes. Fournit également un message d'erreur.

revert() : annuler l'exécution et annuler les changements d'état

revert(string memory reason) : annuler l'exécution et annuler les changements d'état, fournissant une phrase explicative

Fonctions mathématiques et cryptographiques

addmod(uint x, uint y, uint k) returns (uint) : calcule `(x + y) % k` où l'addition est effectuée avec une précision arbitraire et n'overflow pas à `2**256`. `assert` que `k != 0` à partir de la version 0.5.0.

mulmod(uint x, uint y, uint k) returns (uint) : calcule `(x * y) % k` où la multiplication est effectuée avec une précision arbitraire et n'overflow pas à `2**256`. `assert` que `k != 0` à partir de la version 0.5.0.

keccak256(bytes memory) returns (bytes32) : calcule le hash Keccak-256 du paramètre

---

Note : Il y avait un alias pour `keccak256` appelé `sha3`, qui a été supprimé dans la version 0.5.0. pour éviter la confusion

---

sha256(bytes memory) returns (bytes32) : calcule le hash SHA-256 du paramètre

ripemd160(bytes memory) returns (bytes20) : calcule le hash RIPEMD-160 du paramètre

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address) :

récupérer l'adresse associée à la clé publique à partir de la signature de la courbe elliptique ou retourner zéro sur erreur. The function parameters correspond to ECDSA values of the signature :
— `r` = first 32 bytes of signature
— `s` = second 32 bytes of signature
— `v` = final 1 byte of signature

La fonction `ecrecover` renvoie une `address`, et non une `address payable`. Voir *adresse payable* pour la conversion, au cas où vous auriez besoin de transférer des fonds à l'adresse récupérée.

For further details, read example usage.

Avertissement : If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for _transaction_ signatures (see EIP-2), but the ecrecover function remained unchanged.

This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin have a ECDSA helper library that you can use as a wrapper for `ecrecover` without this issue.

Note : Il se peut que vous rencontriez `out-of-gas` pour `sha256`, `ripemd160` ou `erecover` sur une *blockchain privée*. La raison en est que ces contrats sont mis en œuvre sous la forme de contrats dits précompilés et que ces contrats n'existent réellement qu'après avoir reçu le premier message (bien que leur code contrat soit codé en dur). Les messages à des contrats inexistants sont plus coûteux et l'exécution se heurte donc à une erreur out-of-gas. Une solution de contournement pour ce problème est d'envoyer d'abord, par exemple, 1 Wei à chacun des contrats avant de les utiliser dans vos contrats réels. Le problème n'existe pas sur la cha^ine publique Ethereum ni sur les différents testnets officiels.

Membres du type address

`<address>.balance (uint256)` : balance de l'*Adresses* en Wei

`<address payable>.transfer(uint256 amount)` : envoie la quantité donnée de Wei à adress, `revert` en cas d'échec, envoie 2300 gas (non réglable)

`<address payable>.send(uint256 amount) returns (bool)` : envoie la quantité donnée de Wei à adress, retourne `false` en cas d'échec, envoie 2300 gas (non réglable)

`<address>.call(bytes memory) returns (bool, bytes memory)` : émett un appel de bas niveau `CALL` avec la charge utile donnée, renvoie l'état de réussite et les données de retour, achemine tout le gas disponible ou un montant spécifié

`<address>.delegatecall(bytes memory) returns (bool, bytes memory)` : émet un appel de bas niveau `DELEGATECALL` avec la charge utile donnée, retourne les données de succès et de retour, achemine tout le gas disponible ou un montant spécifié

`<address>.staticcall(bytes memory) returns (bool, bytes memory)` : émettre un appel de bas niveau `STATICCALL` avec la charge utile donnée, retourne les conditions de succès et les données de retour, achemine tout le gas disponible ou un montant spécifié

Pour plus d'informations, voir la section sur adress.

Avertissement : You should avoid using `.call()` whenever possible when executing another contract function as it bypasses type checking, function existence check, and argument packing.

Avertissement : Il y a certains dangers à utiliser l'option `send` : Le transfert échoue si la profondeur de la pile d'appels est à 1024 (cela peut toujours être forcé par l'appelant) et il échoue également si le destinataire manque de gas. Donc, afin d'effectuer des transferts d'éther en toute sécurité, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou mieux encore : Utilisez un modèle où le bénéficiaire retire l'argent.

Note : Avant la version 0.5.0, Solidity permettait aux membres d'adresses d'être accessibles par une instance de contrat, par exemple `this.balance`. Ceci est maintenant interdit et une conversion explicite en adresse doit être faite : `address(this).balance`.

Note :

Si l'accès aux variables d'état s'effectue via un appel de délégation de bas niveau, le plan de stockage des deux contrats doit
Ce n'est bien sûr pas le cas si les pointeurs de stockage sont passés comme arguments de fonction comme dans le cas des fonctions de librairies (bibliothèques) de haut niveau.

Note : Avant la version 0.5.0, `.call`, `.delegatecall` et `staticcall` ne renvoyaient que la condition de succès et non les données de retour.

Note : Avant la version 0.5.0, il y avait un membre appelé `callcode` avec une sémantique similaire mais légèrement différente de celle de `delegatecall`.

Contract Related

`this` (type du contrat courant) : le contrat en cours, explicitement convertible en *Adresses*.

`selfdestruct(address payable destinataire_des_fonds)` : détruire le contrat en cours, en envoyant ses fonds à l'adresse *Adresses* indiquée

Note that `selfdestruct` has some peculiarities inherited from the EVM :
— the receiving contract's receive function is not executed.
— the contract is only really destroyed at the end of the transaction and `revert` s might « undo » the destruction.
En outre, toutes les fonctions du contrat en cours peuvent être appelées directement, y compris la fonction en cours.

Note : Avant la version 0.5.0, il existait une fonction appelée `suicide` avec la même sémantique que `selfdestruct`.

Type Information

The expression `type(X)` can be used to retrieve information about the type `X`. Currently, there is limited support for this feature (`X` can be either a contract or an integer type) but it might be expanded in the future.

The following properties are available for a contract type `C` :

`type(C).name` The name of the contract.

`type(C).creationCode` Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `create2` opcode. This property can **not** be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

`type(C).runtimeCode` Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of `C`. If `C` has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with `.creationCode` also apply for this property.

In addition to the properties above, the following properties are available for an interface type `I` :

`type(I).interfaceId` : A `bytes4` value containing the EIP-165 interface identifier of the given interface `I`. This identifier is defined as the `XOR` of all function selectors defined within the interface itself - excluding all inherited functions.

The following properties are available for an integer type `T` :

`type(T).min` The smallest value representable by type `T`.

`type(T).max` The largest value representable by type `T`.

3.8 Expressions et structures de contrôle

3.8.1 Structures de controle

La plupart des structures de contrôle connues des langages à accolades sont disponibles dans Solidity :

Nous disposons de : `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, avec la syntaxe famillière du C ou du JavaScript.

Solidity also supports exception handling in the form of `try`/`catch`-statements, but only for *external function calls* and contract creation calls.

Les parenthèses ne peuvent *pas* être omises pour les conditions, mais les accolades peuvent être omises autour des déclaration en une opération.

Notez qu'il n'y a pas de conversion de types non booléens vers types booléens comme en C et JavaScript, donc `if (1) {...}` n'est pas valable en Solidity.

3.8.2 Appels de fonction

Appels de fonction internes

Les fonctions du contrat en cours peuvent être appelées directement (`internal`), également de manière récursive, comme le montre cet exemple absurde :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

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

Ces appels de fonction sont traduits en simples sauts (`JUMP`) à l'intérieur de l'EVM. Cela a pour effet que la mémoire actuelle n'est pas effacée, c'est-à-dire qu'il est très efficace de passer des références de mémoire aux fonctions appelées en interne. Seules les fonctions du même contrat peuvent être appelées en interne.

Vous devriez toujours éviter une récursivité excessive, car chaque appel de fonction interne utilise au moins un emplacement de pile et il y a au maximum un peu moins de 1024 emplacements disponibles.

Appels de fonction externes

Les expressions `this.g(8);` et `c.g(2);` (où `c` est une instance de contrat) sont aussi des appels de fonction valides, mais cette fois-ci, la fonction sera appelée `external`, via un appel de message et non directement via des sauts. Veuillez noter que les appels de fonction sur `this` ne peuvent pas être utilisés dans le constructeur, car le contrat actuel n'a pas encore été créé.

Les fonctions d'autres contrats doivent être appelées en externe. Pour un appel externe, tous les arguments de fonction doivent être copiés en mémoire.

Note : A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

Lors de l'appel de fonctions d'autres contrats, le montant de Wei envoyé avec l'appel et le gas peut être spécifié avec les options spéciales `.value()` et `.gas()` : `{value: 10, gas: 10000}`. Note that it is discouraged to specify gas values explicitly, since the gas costs of opcodes can change in the future. Any Wei you send to the contract is added to the total balance of that contract :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

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

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

Vous devez utiliser le modificateur `payable` avec la fonction `info` pour pouvoir appeler `.value()` .

Avertissement : Veillez à ce que `feed.info.value(10).gas(800)` ne définisse que localement la `value` et la quantité de `gas` envoyés avec l'appel de fonction, et que les parenthèses à la fin sont bien présentes pour effectuer l'appel. Ainsi, dans cet exemple, la fonction n'est pas appelée.

Les appels de fonction provoquent des exceptions si le contrat appelé n'existe pas (dans le sens où le compte ne contient pas de code) ou si le contrat appelé lui-même lève une exception ou manque de gas.

Avertissement : Toute interaction avec un autre contrat présente un danger potentiel, surtout si le code source du contrat n'est pas connu à l'avance. Le contrat actuel cède le contrôle au contrat appelé et cela peut potentiellement faire à peu près n'importe quoi. Même si le contrat appelé hérite d'un contrat parent connu, le contrat d'héritage doit seulement avoir une interface correcte. L'exécution du contrat peut cependant être totalement arbitraire et donc représentent un danger. En outre, soyez prêt au cas où il appelle d'autres fonctions de votre contrat ou même de retour dans le contrat d'appel avant le retour du premier appel. Cela signifie que le contrat appelé peut modifier les variables d'état du contrat appelant via ses fonctions. Écrivez vos fonctions de manière à ce que, par exemple, les appels à les fonctions externes se produisent après tout changement de variables d'état dans votre contrat, de sorte que votre contrat n'est pas vulnérable à un exploit de réentrée.

Note : Before Solidity 0.6.2, the recommended way to specify the value and gas was to use `f.value(x).gas(g)()`. This is still possible but deprecated and will be removed with Solidity 0.7.0.

Appels nommés et paramètres de fonction anonymes

Les arguments d'appel de fonction peuvent être donnés par leur nom, dans n'importe quel ordre, s'ils sont inclus dans `{ }` comme on peut le voir dans l'exemple qui suit. La liste d'arguments doit coïncider par son nom avec la liste des paramètres de la déclaration de fonction, mais peut être dans un ordre arbitraire.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}
```

Noms des paramètres de fonction omis

Les noms des paramètres inutilisés (en particulier les paramètres de retour) peuvent être omis. Ces paramètres seront toujours présents sur la pile, mais ils sont inaccessibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

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

3.8.3 Création de contrats via `new`

Un contrat peut créer d'autres contrats en utilisant le mot-clé `new`. Le code complet du contrat en cours de création doit être connu lors de la compilation afin d'éviter les dépendances récursives liées à la création.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

contract D {
    uint public x;
    constructor(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // sera exécuté dans le constructor de C

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

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
```

(suite sur la page suivante)

(suite de la page précédente)

```
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

Comme dans l'exemple, il est possible d'envoyer des Ether en créant une instance de `D` en utilisant l'option `.value()`, mais il n'est pas possible de limiter la quantité de gas. Si la création échoue (à cause d'une rupture de pile, d'un manque de gas ou d'autres problèmes), une exception est levée.

Salted contract creations / create2

When creating a contract, the address of the contract is computed from the address of the creating contract and a counter that is increased with each contract creation.

If you specify the option `salt` (a bytes32 value), then contract creation will use a different mechanism to come up with the address of the new contract :

It will compute the address from the address of the creating contract, the given salt value, the (creation) bytecode of the created contract and the constructor arguments.

In particular, the counter (« nonce ») is not used. This allows for more flexibility in creating contracts : You are able to derive the address of the new contract before it is created. Furthermore, you can rely on this address also in case the creating contracts creates other contracts in the meantime.

The main use-case here is contracts that act as judges for off-chain interactions, which only need to be created if there is a dispute.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

contract D {
    uint public x;
    constructor(uint a) public {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        /// This complicated expression just tells you how the address
        /// can be pre-computed. It is just there for illustration.
        /// You actually only need ``new D{salt: salt}(arg)``.
        address predictedAddress = address(uint(keccak256(abi.encodePacked(
            byte(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                arg
            ))
        ))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Avertissement : There are some peculiarities in relation to salted creation. A contract can be re-created at the same address after having been destroyed. Yet, it is possible for that newly created contract to have a different deployed bytecode even though the creation bytecode has been the same (which is a requirement because otherwise the address would change). This is due to the fact that the compiler can query external state that might have changed between the two creations and incorporate that into the deployed bytecode before it is stored.

3.8.4 Ordre d'évaluation des expressions

L'ordre d'évaluation des expressions est non spécifié (plus formellement, l'ordre dans lequel les enfants d'un noeud de l'arbre des expressions sont évalués n'est pas spécifié, mais ils sont bien sûr évalués avant le noeud lui-même). La seule garantie est que les instructions sont exécutées dans l'ordre et que les expressions booléennes sont court-circuitées correctement.

3.8.5 Assignation

Déstructuration d'assignations et retour de valeurs multiples

Solidity permet en interne les tuples, c'est-à-dire une liste d'objets de types potentiellement différents dont le nombre est une constante au moment de la compilation. Ces tuples peuvent être utilisés pour retourner plusieurs valeurs en même temps. Ceux-ci peuvent ensuite être affectés soit à des variables nouvellement déclarées, soit à des variables préexistantes (ou à des LValues en général).

Les tuples ne sont pas des types propres à Solidity, ils ne peuvent être utilisés que pour former des groupes syntaxiques d'expressions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract C {
    uint index;

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

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // 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).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid : `(x, uint y) = (1, 2);`

Note : Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

Avertissement : Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behaviour.

Complications pour les tableaux et les structures

La sémantique des affectations est un peu plus compliquée pour les types autres que valeurs comme les tableaux et les structs, y compris `bytes` et `string`, voir *Emplacement des donnés et comportements à l'assignation* pour plus de détails.

In the example below the call to `g(x)` has no effect on `x` because it creates an independent copy of the storage value in memory. However, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 Portée et déclarations

Une variable qui est déclarée aura une valeur par défaut initiale dont la représentation octale est égale à une suite de zéros. Les « valeurs par défaut » des variables sont les « états zéro » typiques quel que soit le type. Par exemple, la valeur par défaut d'un `bool` est `false`. La valeur par défaut pour les types `uint` ou `int` est `0`. Pour les tableaux de taille statique et les `bytes1` à `bytes32`, chaque élément individuel sera initialisé à la valeur par défaut correspondant à son type. Enfin, pour les tableaux de taille dynamique, les octets et les chaînes de caractères, la valeur par défaut est un tableau ou une chaîne vide.

La portée en Solidity suit les règles de portée très répandues du C99 (et de nombreux autres languages) : Les variables sont visibles du point situé juste après leur déclaration jusqu'à la fin du plus petit bloc `{ }` qui contient la déclaration. Par exception à cette règle, les variables déclarées dans la partie initialisation d'une boucle `for` ne sont visibles que jusqu'à la fin de la boucle for.

Les variables et autres éléments déclarés en dehors d'un bloc de code, par exemple les fonctions, les contrats, les types définis par l'utilisateur, etc. sont visibles avant même leur déclaration. Cela signifie que vous pouvez utiliser les variables d'état avant qu'elles ne soient déclarées et appeler les fonctions de manière récursive.

Par conséquent, les exemples suivants seront compilés sans avertissement, puisque les deux variables ont le même nom mais des portées disjointes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

À titre d'exemple particulier des règles de détermination de la portée héritées du C99, notons que, dans ce qui suit, la première affectation à `x` affectera en fait la variable externe et non la variable interne. Dans tous les cas, vous obtiendrez un avertissement concernant cette double déclaration.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

Avertissement : Avant la version 0.5.0, Solidity suivait les mêmes règles de scoping que JavaScript, c'est-à-dire qu'une variable déclarée n'importe où dans une fonction était dans le champ d'application pour l'ensemble de la fonction, peu importe où elle était déclarée. L'exemple suivant montre un extrait de code qui compilait, mais conduit aujourd'hui à une erreur à partir de la version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 Gestion d'erreurs : Assert, Require, Revert et Exceptions

Solidity utilise des exceptions qui restaurent l'état pour gérer les erreurs. Une telle exception annule toutes les modifications apportées à l'état de l'appel en cours (et de tous ses sous-appels) et signale également une erreur à l'appelant.

Lorsque des exceptions se produisent dans un sous-appel, elles « remontent à la surface » automatiquement (c'est-à-dire que les exceptions sont déclenchées en casacade). Les exceptions à cette règle sont `send` et les fonctions de bas niveau `call`, `delegatecall` et `staticcall`, qui retournent `false` comme première valeur de retour en cas d'exception au lieu de lancer une chaine d'exceptions.

Avertissement : Les fonctions de bas niveau `call`, `delegatecall` et `staticcall` renvoient `true` comme première valeur de retour si le compte appelé est inexistant, dû à la conception de l'EVM. L'existence doit être vérifiée avant l'appel si désiré.

Exceptions can be caught with the `try`/`catch` statement.

`assert` and `require`

Les fonctions utilitaires `assert` et `require` peuvent être utilisées pour vérifier les conditions et lancer une exception si la condition n'est pas remplie.

La fonction `require` doit être utilisée pour s'assurer que les conditions valides, telles que les entrées ou les variables d'état du contrat, sont remplies, ou pour valider les valeurs de retour des appels aux contrats externes. S'ils sont utilisés correctement, les outils d'analyse peuvent évaluer votre contrat afin d'identifier les conditions et les appels de fonction qui parviendront à un échec d'"`assert`. Un code fonctionnant correctement ne devrait jamais échouer un `assert` ; si cela se produit, il y a un bogue dans votre contrat que vous devriez corriger.

Une exception de type `assert` est générée dans les situations suivantes :

1. Si vous accédez à un tableau avec un index trop grand ou négatif (par ex. `x[i]` où `i >= x.length` ou `i < 0`).
2. Si vous accédez à une variable de longueur fixe `bytesN` à un indice trop grand ou négatif.
3. Si vous divisez ou modulez par zéro (par ex. `5 / 0` ou `23 % 0`).
4. Si vous décalez d'un montant négatif.
5. Si vous convertissez une valeur trop grande ou négative en un type enum.
6. Si vous appelez une variable initialisée nulle de type fonction interne.
7. Si vous appelez `assert` avec un argument qui s'évalue à `false`.

The `require` function should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

Une exception de type `require` est générée dans les situations suivantes :

1. Appeler `require` avec un argument qui s'évalue à `false`.
2. Si vous appelez une fonction via un appel de message mais qu'elle ne se termine pas correctement (c'est-à-dire qu'elle n'a plus de gas, qu'elle n'a pas de fonction correspondante ou qu'elle lance une exception elle-même), sauf lorsqu'une opération de bas niveau `call`, `send`, `staticcall`, `delegatecall` ou `callcode` est utilisée. Les opérations de bas niveau ne lancent jamais d'exceptions mais indiquent les échecs en retournant `false`.
3. Si vous créez un contrat en utilisant le mot-clé `new` mais que la création du contrat ne se termine pas correctement (voir ci-dessus pour la définition de « ne pas terminer correctement »).
4. Si vous effectuez un appel de fonction externe ciblant un contrat qui ne contient aucun code.
5. Si votre contrat reçoit des Ether via une fonction publique sans modificateur `payable` (y compris le constructeur et la fonction par defaut).

6. Si votre contrat reçoit des Ether via une fonction de getter public.
7. Si un `.transfer()` échoue.

Vous pouvez facultativement fournir une chaîne de message pour `require`, mais pas pour `assert`.

Dans l'exemple suivant, vous pouvez voir comment `require` peut être utilisé pour vérifier facilement les conditions sur les entrées et comment `assert` peut être utilisé pour vérifier les erreurs internes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

En interne, Solidity exécute une opération de retour en arrière (instruction `0xfd`) pour une exception de type `require` et exécute une opération invalide (instruction `0xfe`) pour lancer une exception de type `assert`. Dans les deux cas, cela provoque lánnulation toutes les modifications apportées à l'état de l'EVM dans l'appel courant. La raison du retour en arrière est qu'il n'y a pas de moyen sûr de continuer l'exécution, parce qu'un effet attendu ne s'est pas produit. Parce que nous voulons conserver l'atomicité des transactions, la chose la plus sûre à faire est d'annuler tous les changements et de faire toute la transaction (ou au moins l'appel) sans effet.

In both cases, the caller can react on such failures using `try`/`catch` (in the failing `assert`-style exception only if enough gas is left), but the changes in the caller will always be reverted.

Note : Les exceptions de type `assert` consomment tout le gas disponible pour l'appel, alors que les exceptions de type `require` ne consommeront pas de gaz à partir du lancement de Metropolis.

---

revert

The `revert` function is another way to trigger exceptions from within other code blocks to flag an error and revert the current call. The function takes an optional string message containing details about the error that is passed back to the caller.

L'exemple suivant montre comment une chaîne d'erreurs peut être utilisée avec `revert` et `require` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract VendingMachine {
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
```

(suite sur la page suivante)

(suite de la page précédente)

```
                "Not enough Ether provided."
            );
            // Perform the purchase.
        }
}
```

The two syntax options are equivalent, it's developer preference which to use.

La chaîne fournie sera *abi-encoded* comme si c'était un appel à une fonction `Error(string)`. Dans l'exemple ci-dessus, `revert("Not enough Ether provided.");` fera en sorte que les données hexadécimales suivantes soient définies comme données de retour d'erreur :

```
0x08c379a0                                                         // Selecteur de␣
↪fonction pour Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Décalage des␣
↪données
0x000000000000000000000000000000000000000000000000000000000000001a // Taille de la␣
↪string
0x4e6f7420656e6f756768204574686572207072f76696465642e000000000000 // Données de la␣
↪string
```

The provided message can be retrieved by the caller using `try`/`catch` as shown below.

Note : There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

try/catch

A failure in an external call can be caught using a try/catch statement, as follows :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used
            // or there was a failing assertion, division
            // by zero, etc. inside getData.
```

(suite sur la page suivante)

(suite de la page précédente)

```
            errorCount++;
            return (0, false);
        }
    }
}
```

The `try` keyword has to be followed by an expression representing an external function call or a contract creation (`new ContractName()`). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The `returns` part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the `catch` blocks.

Currently, Solidity supports different kinds of catch blocks depending on the type of error. If the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception), then the catch clause of the type `catch Error(string memory reason)` will be executed.

It is planned to support other types of error data in the future. The string `Error` is currently parsed as is and is not treated as an identifier.

The clause `catch (bytes memory lowLevelData)` is executed if the error signature does not match any other clause, there was an error during decoding of the error message, if there was a failing assertion in the external call (for example due to a division by zero or a failing `assert()`) or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.

If you are not interested in the error data, you can just use `catch { ... }` (even as the only catch clause).

In order to catch all error cases, you have to have at least the clause `catch { ...}` or the clause `catch (bytes memory lowLevelData) { ... }`.

The variables declared in the `returns` and the `catch` clause are only in scope in the block that follows.

Note : If an error happens during the decoding of the return data inside a try/catch-statement, this causes an exception in the currently executing contract and because of that, it is not caught in the catch clause. If there is an error during decoding of `catch Error(string memory reason)` and there is a low-level catch clause, this error is caught there.

Note : If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

Note : The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract : The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition : The caller always retains 63/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left.

3.9 Contrats

Les contrats en Solidity sont similaires à des classes dans les langages orientés objets. Ils contiennent des données persistentes dans des variables et des fonctions peuvent les modifier. Appeler la fonction d'un autre contrat (une autre instance) executera un appel de fonction auprès de l'EVM et changera alors le contexte, rendant inaccessibles ces variables. A contract and its functions need to be called for anything to happen. There is no « cron » concept in Ethereum to call a function at a particular event automatically.

3.9.1 Créer des contrats

Les contrats peuvent être créés « en dehors » via des transactions Ethereum ou depuis des contrat en Solidity.

Les EDIs, comme Remix, facilite la tâche via des éléments visuels.

Créer des contrats via du code se fait le plus simplement en utilisant l'API Javascript web3.js. Elle possède une fonction appelée web3.eth.Contract qui facilite cette création

Quand un contrat est créé, son constructeur (une fonction déclarée via le mot-clé `constructor`) est executé, de manière unique.

Un constructeur est optionnel. Aussi, un seul constructeur est autorisé, ce qui signifie que l'overloading n'est pas supporté.

Après que le constructeur ai été exécuté, le code final du contrat est déployé sur la Blockchain. Ce code inclut toutes les fonctions publiques et externes, et toutes les fonctions qui sont atteignables par des appels de fonctions. Le code déployé n'inclut pas le constructeur ou les fonctions internes uniquement appelées depuis le constructeur.

En interne, les arguments du constructeur sont passés ABI encodés après le code du contrat lui-même, mais vous n'avez pas à vous en soucier si vous utilisez `web3.js`.

Si un contrat veut créer un autre contrat, le code source (et le binaire) du contrat créé doit être connu du créateur. Cela signifie que les dépendances cycliques de création sont impossibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;


contract OwnedToken {
    // TokenCreator est un type de contrat défini ci-dessous.
    // Il est possible de le référencer tant qu'il n'est pas utilisé
    // pour créer un nouveau contrat.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // Ceci est le constructeur qui enregistre le
    // le créateur et le nom attribué.
    constructor(bytes32 _name) public {
        // Les variables d'état sont accessibles par leur nom
        // et non par l'intermédiaire de this.owner par exemple. Ceci s'applique␣
↪également
        // aux fonctions et en particulier dans les constructeurs,
        // vous ne pouvez les appeler que comme ça ("en interne"),
        // parce que le contrat lui-même n'existe pas encore.
        owner = msg.sender;
        // Nous effectuons une conversion de type explicite de `address`.
        // vers `TokenCreator` et supposons que le type du
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // contrat appelant est TokenCreator,
        // Il n'y a pas vraiment moyen de vérifier ça.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // Seul le créateur peut modifier le nom --
        // la comparaison est possible puisque les contrats
        // sont explicitement convertibles en adresses.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Seul le propriétaire actuel peut transférer le token.
        if (msg.sender != owner) return;

        // Nous voulons aussi demander au créateur si le transfert
        // est valide. Notez que ceci appelle une fonction de la fonction
        // contrat défini ci-dessous. Si l'appel échoue (p. ex.
        // en raison d'un manque de gas), l'exécution échoue également ici.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name)
       public
       returns (OwnedToken tokenAddress)
    {
        // Créer un nouveau contrat Token et renvoyer son adresse.
        // Du côté JavaScript, le type de retour est simplement
        // `address`, car c'est le type le plus proche disponible dans
        // l'ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // Encore une fois, le type externe de `tokenAddress' est
        // simplement `adresse`.
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Vérifier une condition arbitraire.
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

3.9.2 Visibilité et Getters

Puisque Solidity connaît deux types d'appels de fonction (internes qui ne créent pas d'appel EVM réel (également appelés a « message call ») et externes qui le font), il existe quatre types de visibilités pour les fonctions et les variables d'état.

Les fonctions doivent être spécifiées comme étant `external`, `public`, `internal` ou `private`. Pour les variables d'état, `external` n'est pas possible.

`external` : Les fonctions externes font partie de l'interface du contrat, ce qui signifie qu'elles peuvent être appelées à partir d'autres contrats et via des transactions. Une fonction externe `f` ne peut pas être appelée en interne (c'est-à-dire `f()``ne fonctionne pas, mais ``this.f()` fonctionne). Les fonctions externes sont parfois plus efficaces lorsqu'elles reçoivent de grandes quantités de données.

`public` : Les fonctions publiques font partie de l'interface du contrat et peuvent être appelées en interne ou via des messages. Pour les variables d'état publiques, une fonction getter automatique (voir ci-dessous) est générée.

`internal` : Ces fonctions et variables d'état ne sont accessibles qu'en interne (c'est-à-dire à partir du contrat en cours ou des contrats qui en découlent), sans utiliser `this`.

`private` : Les fonctions privées et les variables d'état ne sont visibles que pour le contrat dans lequel elles sont définies et non dans les contrats dérivés.

Note :

Tout ce qui se trouve à l'intérieur d'un contrat est visible pour tous les observateurs extérieurs à la blockchain. Passer quelque chose en `private`

ne fait qu'empêcher les autres contrats d'accéder à l'information et de la modifier, mais elle sera toujours visible pour le monde entier à l'extérieur de la blockchain.

Le spécificateur de visibilité est donné après le type pour les variables d'état et entre la liste des paramètres et la liste des paramètres de retour pour les fonctions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

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

Dans l'exemple suivant, `D`, peut appeler `c.getData()` pour retrouver la valeur de `data` en mémoire d'état, mais ne peut pas appeler `f`. Le contrat `E` est dérivé du contrat `C` et peut donc appeler `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract C {
    uint private data;

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

(suite sur la page suivante)

(suite de la page précédente)

```
// Ceci ne compile pas
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // Erreur: le membre `f` n'est pas visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // Erreur: le membre `compute` n'est pas visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // accès à un membre interne (du contrat dérivé au
↪contrat parent)
    }
}
```

Fonctions Getter

Le compilateur crée automatiquement des fonctions getter pour toutes les variables d'état **public**. Pour le contrat donné ci-dessous, le compilateur va générer une fonction appelée `data` qui ne prend aucun argument et retourne un `uint`, la valeur de la variable d'état `data`. Les variables d'état peuvent être initialisées lorsqu'elles sont déclarées.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

Les fonctions getter ont une visibilité externe. Si le symbole est accédé en interne (c'est-à-dire sans `this.`), il est évalué à une variable d'état. S'il est accédé de l'extérieur (c'est-à-dire avec `this.`), il évalue à une fonction.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // accès interne
        return this.data(); // accès externe
    }
}
```

Si vous avez une variable d'état `public` de type array, alors vous ne pouvez récupérer que des éléments simples de l'array via la fonction getter générée. Ce mécanisme permet d'éviter des coûts de gas élevés lors du retour d'un tableau

complet. Vous pouvez utiliser des arguments pour spécifier quel élément individuel retourner, par exemple `data(0)`. Si vous voulez retourner un tableau entier en un appel, alors vous devez écrire une fonction, par exemple :

```
  // SPDX-License-Identifier: GPL-3.0
  pragma solidity >=0.4.16 <0.7.0;

contract arrayExample {
  // variable d'état publique
  uint[] public myArray;

  // Fonction getter générée par le compilateur
  /*
  function myArray(uint i) returns (uint) {
      return myArray[i];
  }
  */

  // fonction retournant une array complète
  function getArray() returns (uint[] memory) {
      return myArray;
  }
}
```

Maintenant vous pouvez utiliser `getArray()` pour récupérer le tableau entier, au lieu de `myArray(i)`, qui retourne un seul élément par appel.

L'exemple suivant est plus complexe :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

Il génère une fonction de la forme suivante. Le mappage dans la structure est omis parce qu'il n'y a pas de bonne façon de fournir la clé pour le mappage :

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

3.9.3 Modificateurs de fonctions

Les modificateurs peuvent être utilisés pour modifier facilement le comportement des fonctions. Par exemple, ils peuvent vérifier automatiquement une condition avant d'exécuter la fonction.

Les modificateurs sont des propriétés héritables des contrats et peuvent être redéfinis dans les contrats dérivés, mais seulement s'ils sont indiqués `virtual`. Pour plus de détails, voir *Modifier Overriding*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;

    // Ce contrat ne définit qu'un modificateur mais ne l'utilise pas:
    // il sera utilisé dans les contrats dérivés.
    // Le corps de la fonction est inséré à l'endroit où le symbole spécial
    // `_;` apparaît dans la définition d'un modificateur.
    // Cela signifie que si le propriétaire appelle cette fonction, la fonction
    // est exécutée et dans le cas contraire, une exception est
    // levée.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract mortal is owned {
    // Ce contrat hérite du modificateur `onlyOwner` de `owned`
    // et l'applique à la fonction `close`, qui
    // cause que les appels à `close` n'ont un effet que s'il
    // sont passés par le propriétaire enregistré.
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // Les modificateurs peuvent prendre des arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

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

    constructor(uint initialPrice) public { price = initialPrice; }

    // Il est important de fournir également le
    // mot-clé `payable` ici, sinon la fonction
    // rejettera automatiquement tous les Ethers qui lui sont envoyés.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// Cette fonction est protégée par un mutex, ce qui signifie que
    /// les appels entrants à partir de `msg.sender.call` ne peuvent pas rappeler `f`.
    /// L'instruction `return 7` assigne 7 à la valeur de retour, mais en même temps
    /// exécute l'instruction `locked = false` dans le modificateur.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

Plusieurs modificateurs sont appliqués à une fonction en les spécifiant dans une liste séparée par des espaces et sont évalués dans l'ordre présenté.

Avertissement : Dans une version antérieure de Solidity, les instructions `return` des fonctions ayant des modificateurs se comportaient différemment.

Les retours explicites d'un modificateur ou d'un corps de fonction ne laissent que le modificateur ou le corps de fonction courant. Les variables de retour sont affectées et le flow de contrôle continue après le « _ » dans le modificateur précédent.

Des expressions arbitraires sont autorisées pour les arguments du modificateur et dans ce contexte, tous les symboles visibles depuis la fonction sont visibles dans le modificateur. Les symboles introduits dans le modificateur ne sont pas visibles dans la fonction (car ils peuvent changer en cas de redéfinition).

3.9.4 Variables d'état constantes

Les variables d'état peuvent être déclarées comme `constantes` ou `immutable`. Dans les deux cas, ces variables ne peuvent être modifiées après la construction du contrat. Dans ce cas, elles doivent être assignées à partir d'une expression constante au moment de la compilation. Pour les variables `constant`, la valeur doit être connue à la compilation. Pour les variables `immutable`, les variables peuvent être assognées jusqu'à la construction.

Le compilateur ne réserve pas d'emplacement de stockage pour ces variables, et chaque occurrence est remplacée par l'expression constante correspondante.

Tous les types de constantes ne sont pas implémentés pour le moment. Les seuls types pris en charge sont chaines de caractères (uniquement pour les constantes) et types valeur.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.6.4 <0.7.0;

contract C {
    uint constant X = 32**22 + 8;
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint _decimals, address _reference) public {
        decimals = _decimals;
        // L'assignement à des immutables peut même accéder à l'environnement
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

Constant

Pour les variables `constant`, doivent être assignées à partir d'une expression constante au moment de la compilation et doit 6etre assignée à la déclaration. Toute expression qui accède au stockage, aux données de la blockchain (par exemple `now`, `address(this).balance` ou `block.number`) ou les données d'exécution (`msg.value` ou `gasleft()`) ou les appels vers des contrats externes sont interdits. Les expressions qui peuvent avoir un effet secondaire sur l'allocation de mémoire sont autorisées, mais celles qui peuvent avoir un effet secondaire sur d'autres objets mémoire ne le sont pas. Les fonctions intégrées `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` et `mulmod` sont autorisées (même si des contrats externes sont appelés).

La raison pour laquelle on autorise les effets secondaires sur l'allocateur de mémoire est qu'il devrait être possible de construire des objets complexes comme par exemple des tables de consultation. Cette fonctionnalité n'est pas encore entièrement utilisable.

Immutable

Variables declared as `immutable` are a bit less restricted than those declared as `constant` : Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They cannot be read during construction time and can only be assigned once.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables by the values assigned to the them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

3.9.5 Fonctions

Function Parameters and Return Variables

Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

Function Parameters

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

For example, if you want your contract to accept one kind of external call with two integers, you would use something like the following :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract Simple {
    uint sum;
    function taker(uint _a, uint _b) public {
        sum = _a + _b;
    }
}
```

Function parameters can be used as any other local variable and they can also be assigned to.

Note : An *external function* cannot accept a multi-dimensional array as an input parameter. This functionality is possible if you enable the new `ABIEncoderV2` feature by adding `pragma experimental ABIEncoderV2;` to your source file.

An *internal function* can accept a multi-dimensional array without enabling the feature.

Return Variables

Function return variables are declared with the same syntax after the `returns` keyword.

For example, suppose you want to return two results : the sum and the product of two integers passed as function parameters, then you use something like :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

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

The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their *default value* and have that value until they are (re-)assigned.

You can either explicitly assign to return variables and then leave the function using `return;`, or you can provide return values (either a single or *multiple ones*) directly with the `return` statement :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

```

(suite sur la page suivante)

(suite de la page précédente)

```
contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        return (_a + _b, _a * _b);
    }
}
```

This form is equivalent to first assigning values to the return variables and then using `return;` to leave the function.

Note : You cannot return some types from non-internal functions, notably multi-dimensional dynamic arrays and structs. If you enable the new `ABIEncoderV2` feature by adding `pragma experimental ABIEncoderV2;` to your source file then more types are available, but `mapping` types are still limited to inside a single contract and you cannot transfer them.

Returning Multiple Values

When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an *implicit conversion*.

Fonctions View

Les fonctions peuvent être déclarées `view`, auquel cas elles promettent de ne pas modifier l'état.

Note : Si la cible EVM du compilateur est Byzantium ou plus récent (par défaut), l'opcode `STATICCALL` est utilisé pour les fonctions `view` qui imposent à l'état de rester non modifié lors de l'exécution EVM. Pour les librairies, on utilise les fonctions `view` et `DELEGATECALL` parce qu'il n'y a pas de `DELEGATECALL` et `STATICCALL` combinés. Cela signifie que les fonctions `view` de librairies n'ont pas de contrôles d'exécution qui empêchent les modifications d'état. Cela ne devrait pas avoir d'impact négatif sur la sécurité car le code de librairies est généralement connu au moment de la compilation et le vérificateur statique effectue les vérifications au moment de la compilation.

Les déclarations suivantes sont considérées comme une modification de l'état :

1. Ecrire dans les variables d'état.
2. *Emettre des événements*.
3. *Création d'autres contrats*.
4. Utiliser `selfdestruct`.
5. Envoyer des Ethers par des appels.
6. Appeler une fonction qui n'est pas marquée `view` ou `pure`.
7. Utilisation d'appels bas niveau.
8. Utilisation d'assembleur inline qui contient certains opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

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

Note : `constant` sur les fonctions était un alias de `view`, mais cela a été abandonné dans la version 0.5.0.

Note : Les méthodes Getter sont automatiquement marquées `view`.

Note : Avant la version 0.5.0, le compilateur n'utilisait pas l'opcode `STATICCALL`. pour les fonctions `view`. Cela permettait de modifier l'état des fonctions `view` grâce à l'utilisation de conversions de type explicites non valides. En utilisant `STATICCALL` pour les fonctions `view`, les modifications de la fonction sont évités au niveau de l'EVM.

Fonctions Pure

Les fonctions peuvent être déclarées `pures`, auquel cas elles promettent de ne pas lire ou modifier l'état.

Note : Si la cible EVM du compilateur est Byzantium ou plus récente (par défaut), on utilise l'opcode `STATICCALL`, ce qui ne garantit pas que l'état ne soit pas lu, mais au moins qu'il ne soit pas modifié.

En plus de la liste des modificateurs d'état expliqués ci-dessus, sont considérés comme des lectures de l'état :

1. Lecture des variables d'état.
2. Accéder à `address(this).balance` ou `<address>.balance`.
3. Accéder à l'un des membres de `block`, `tx`, `msg` (à l'exception de `msg.sig` et `msg.data`).
4. Appeler une fonction qui n'est pas marquée `pure`.
5. Utilisation d'assembleur inline qui contient certains opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

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

Note : Avant la version 0.5.0, le compilateur n'utilisait pas l'opcode `STATICCALL` pour les fonctions `pure`. Cela permettait de modifier l'état des fonctions `pures` en utilisant des conversions de type explicites invalides. En utilisant `STATICCALL` pour des fonctions `pures`, les modifications de l'état sont empêchées au niveau de l'EVM.

Avertissement : Il n'est pas possible d'empêcher les fonctions de lire l'état au niveau de l'EVM, il est seulement possible de les empêcher d'écrire dans l'état (c'est-à-dire que seul « view » peut être exécuté au niveau de l'EVM, `pure` ne peut pas).

Avertissement : Avant la version 0.4.17, le compilateur n'appliquait pas le fait que `pure` ne lisait pas l'état. Il s'agit d'un contrôle de type à la compilation, qui peut être contourné en effectuant des conversions explicites invalides entre les types de contrats, parce que le compilateur peut vérifier que le type de contrat ne fait pas d'opérations de changement d'état, mais il ne peut pas vérifier que le contrat qui sera appelé à l'exécution est effectivement de ce type.

Receive Ether Function

A contract can have at most one `receive` function, declared using `receive() external payable { ... }` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility and `payable` state mutability. It is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via `.send()` or `.transfer()`). If no such function exists, but a payable *fallback function* exists, the fallback function will be called on a plain Ether transfer. If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.

In the worst case, the fallback function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging. The following operations will consume more gas than the 2300 gas stipend :

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

Avertissement : Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) but do not define a receive Ether function or a payable 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 receive Ether function (using payable fallback functions for receiving Ether is not recommended, since it would not fail on interface confusions).

Avertissement : A contract without a receive Ether function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

Below you can see an example of a Sink contract that uses function `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fonction de repli

Un contrat peut avoir exactement une fonction sans nom. Cette fonction ne peut pas avoir d'arguments, ne peut rien retourner et doit avoir une visibilité `external`. Elle est exécutée lors d'un appel au contrat si aucune des autres fonctions ne correspond à l'identificateur de fonction donné (ou si aucune donnée n'a été fournie).

En outre, cette fonction est exécutée chaque fois que le contrat reçoit des Ethers bruts (sans données). De plus, pour recevoir des Ethers, la fonction de fallback doit être marquée `payable`. En l'absence d'une telle fonction, le contrat ne peut recevoir d'Ether par des transactions traditionnelles.

Dans le pire des cas, la fonction de fallback ne peut compter que sur la disponibilité de 2 300 gas (par exemple lorsque l'on utilise *send* ou *transfer*), ce qui laisse peu de place pour effectuer d'autres opérations que du log basique. Les opérations suivantes consommeront plus de gaz que le forfait de 2 300 gas alloué :

- Ecrire dans le stockage
- Création d'un contrat
- Appel d'une fonction externe qui consomme une grande quantité de gas
- Envoi d'Ether

Comme toute fonction, la fonction de fallback peut exécuter des opérations complexes tant que suffisamment de gas lui est transmis.

Note : Même si la fonction de fallback ne peut pas avoir d'arguments, on peut toujours utiliser `msg.data` pour récupérer toute charge utile fournie avec l'appel.

Avertissement : La fonction de fallback est également exécutée si l'appelant a l'intention d'appeler une fonction qui n'est pas disponible. Si vous voulez implémenter la fonction de fallback uniquement pour recevoir de l'Ether, vous devez ajouter une vérification comme `require(msg.data.length == 0)` pour éviter les appels invalides.

Avertissement : Les contrats qui reçoivent directement l'Ether (sans appel de fonction, c'est-à-dire en utilisant `send` ou ‘‘ transfer‘‘) mais ne définissent pas de fonction de fallback lèvent une exception, renvoyant l'Ether (c'était différent avant Solidity v0.4.0). Donc si vous voulez que votre contrat reçoive de l'Ether, vous devez implémenter une fonction de fallback `payable`.

Avertissement : Un contrat sans fonction de fallback payable peut recevoir de l'Ether en tant que destinataire d'une *coinbase transaction* (alias *récompense de mineur de bloc*) ou en tant que destination d'un `selfdestruct`.

Un contrat ne peut pas réagir à de tels transferts d'Ether et ne peut donc pas non plus les rejeter. C'est un choix de conception de l'EVM et Solidity ne peut le contourner.

Cela signifie également que `address(this).balance` peut être plus élevé que la somme de certaines comptabilités manuelles implémentées dans un contrat (i.e. avoir un compteur mis à jour dans la fonction fallback).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

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.
    fallback() external { x = 1; }
    uint x;
}

contract TestPayable {
    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable { x = 1; y = msg.value; }

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable { x = 2; y = msg.value; }
    uint x;
    uint y;
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has␣
↪no payable
        // fallback function.
        // It has to be converted to the ``address payable`` type to even allow␣
↪calling ``send`` on it.
        address payable testPayable = payable(address(test));

        // If someone sends Ether to that contract,
        // the transfer will fail, i.e. this returns false here.
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 0.
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
↪"nonExistingFunction()"));
```

(suite sur la page suivante)

(suite de la page précédente)

```
        require(success);
        // results in test.x becoming == 1 and test.y becoming 1.

        // If someone sends Ether to that contract, the receive function in␣
↪TestPayable will be called.
        require(address(test).send(2 ether));
        // results in test.x becoming == 2 and test.y becoming 2 ether.
    }
}
```

Surcharge de fonctions

Un contrat peut avoir plusieurs fonctions du même nom, mais avec des types de paramètres différents. Ce processus est appelé « surcharge » et s'applique également aux fonctions héritées. L'exemple suivant montre la surcharge de la fonction `f` dans le champ d'application du contrat `A`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = _in;
    }

    function f(uint _in, bool _really) public pure returns (int out) {
        if (_really)
            out = int(_in);
    }
}
```

Des fonctions surchargées sont également présentes dans l'interface externe. C'est une erreur si deux fonctions visibles de l'extérieur diffèrent par leur type Solidity (ici *A* et *B*) mais pas par leur type extérieur (ici `address`).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

// Ceci ne compile pas
contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(A _in) public pure returns (B out) {
        out = B(address(_in));
    }
}

contract B {
}
```

Les deux fonctions `f` surchargées ci-dessus acceptent des addresses du point de vue de l'ABI, mais ces adresses sont considérées comme différents types en Solidity.

Résolution des surcharges et concordance des arguments

Les fonctions surchargées sont sélectionnées en faisant correspondre les déclarations de fonction dans le scope actuel aux arguments fournis dans l'appel de fonction. La fonction évaluée est choisie si tous les arguments peuvent être implicitement convertis en types attendus. S'il y a plusieurs fonctions correspondantes, la résolution échoue.

Note : Le type des valeurs retournées par la fonction n'est pas pris en compte dans la résolution des surcharges.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

L'appel de `f(50)` créerait une erreur de type puisque `50` peut être implicitement converti à la fois en type `uint8` et `uint256`. D'un autre côté, `f(256)` se résoudrait à `f(uint256)` car `256` ne peut pas être implicitement converti en `uint8`.

3.9.6 Événements

Les événements Solidity autorisent une abstraction en plus de la fonctionnalité de journalisation de l'EVM. Les applications peuvent souscrire à et écouter ces événements via l'interface RPC d'un client Ethereum.

Les événements sont des membres héritables des contrats. Lorsque vous les appelez, ils font en sorte que les arguments soient stockés dans le journal des transactions - une structure de données spéciale dans la blockchain. Ces logs sont associés à l'adresse du contrat, sont incorporés dans la blockchain et y restent tant qu'un bloc est accessible (pour toujours à partir des versions Frontier et Homestead, mais cela peut changer avec Serenity). Le journal et ses données d'événement ne sont pas accessibles depuis les contrats (pas même depuis le contrat qui les a créés).

Il est possible de demander une simple vérification de paiement (SPV) pour les logs, de sorte que si une entité externe fournit un contrat avec une telle vérification, elle peut vérifier que le log existe réellement dans la blockchain. Vous devez fournir des en-têtes (headers) de bloc car le contrat ne peut voir que les 256 derniers hashs de blocs.

Vous pouvez ajouter l'attribut `indexed` à un maximum de trois paramètres qui les ajoute à une structure de données spéciale appelée *« topics »* au lieu de la partie data du log. Si vous utilisez des tableaux (y compris les `string` et `bytes`) comme arguments indexés, leurs hashs Keccak-256 sont stockés comme topic à la place, car un topic ne peut contenir qu'un seul mot (32 octets).

Tous les paramètres sans l'attribut `indexed` sont *ABI-encoded* dans la partie données du log.

Les topics vous permettent de rechercher des événements, par exemple lors du filtrage d'une séquence de blocs pour certains événements. Vous pouvez également filtrer les événements par l'adresse du contrat qui les a émis.

Par exemple, le code ci-dessous utilise web3.js `subscribe("logs")` method pour filtrer les logs qui correspondent à un sujet avec une certaine valeur d'adresse :

```
var options = {
    fromBlock: 0,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000",␣
↪null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

Le hash de la signature de l'event est l'un des topics, sauf si vous avez déclaré l'événement avec le spécificateur « anonymous ». Cela signifie qu'il n'est pas possible de filtrer des événements anonymes spécifiques par leur nom.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.7.0;

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

    function deposit(bytes32 _id) public payable {
        // Les événements sont émis à l'aide de `emit`, suivi du
        // nom de l'événement et des arguments
        // (le cas échéant) entre parenthèses. Une telle invocation
        // (même profondément imbriquée) peut être détectée à partir de
        // l'API JavaScript en filtrant `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

L'utilisation dans l'API JavaScript est la suivante :

```
var abi = /* abi telle que générée par le compilateur */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* adresse */);

var event = clientReceipt.Deposit();

// inspecter les eventuels changements
event.watch(function(error, result){
    // le résultat contient des arguments et topics non indexés
    // passées à l'appel de `Deposit`.
    if (!error)
        console.log(result);
});


// Ou passez une fonction pour ecouter dès maintenant
var event = clientReceipt.Deposit(function(error, result) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if (!error)
        console.log(result);
});
```

La sortie du code ci-dessus ressemble à (trimmée) :

```
{
   "returnValues": {
       "_from": "0x1111...FFFFCCCC",
       "_id": "0x50...sd5adb20",
       "_value": "0x420042"
   },
   "raw": {
       "data": "0x7f...91385",
       "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

Interface bas-niveau des Logs

Il est également possible d'accéder à l'interface bas niveau du mécanisme de logs via les fonctions `log0`, `log1`, `log2`, `log3` et `log4`. `logi` prend le paramètre `i + 1` paramètre de type `bytes32`, où le premier argument sera utilisé pour la partie données du journal et les autres comme sujets. L'appel d'événement ci-dessus peut être effectué de la même manière que

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.10 <0.7.0;

contract C {
    function f() public payable {
        uint256 _id = 0x420042;
        log3(
            bytes32(msg.value),
            ␣
↪bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20),
            bytes32(uint256(msg.sender)),
            bytes32(_id)
        );
    }
}
```

où le nombre hexadécimal long est égal à `keccak256("Deposit(address,bytes32,uint256)")`, la signature de l'événement.

Ressources complémentaires pour comprendre les Events

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

3.9.7 Héritage

Solidity supporte l'héritage multiple en copiant du code, incluant le polymorphisme.

Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy. This has to be explicitly enabled on each function in the hierarchy using the `virtual` and `override` keywords. See *Function Overriding* for more details.

It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy (see below).

Lorsqu'un contrat hérite d'autres contrats, un seul contrat est créé dans la blockchain et le code de tous les contrats de base est copié dans le contrat créé. This means that all internal calls to functions of base contracts also just use internal function calls (`super.f(..)` will use JUMP and not a message call).

State variable shadowing is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

Le système général d'héritage est très similaire à celui de Python, surtout en ce qui concerne l'héritage multiple, mais il y a aussi quelques *differences*.

Les détails sont donnés dans l'exemple suivant.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;


contract Owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}


// Utilisez `is` pour dériver d'un autre contrat. Les contrats dérivés peuvent␣
↪accéder à tous les membres non privés, y compris les fonctions internes et les␣
↪variables d'état. Il n'est cependant pas possible d'y accéder de l'extérieur via␣
↪`this`.
contract Destructible is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// Ces contrats abstraits ne sont fournis que pour faire connaître l'interface au␣
↪compilateur. Notez la fonction sans corps. Si un contrat n'implémente pas toutes␣
↪les fonctions, il ne peut être utilisé que comme interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}


abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}


// L'héritage multiple est possible. Notez que `owned` est aussi une classe de base␣
↪de `mortal`, pourtant il n'y a qu'une seule instance de `owned` (comme pour l␣
↪'héritage virtuel en C++).
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract Named is Owned, Destructible {
    constructor(bytes32 name) public {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Les fonctions peuvent être remplacées par une autre fonction ayant le même nom
↪et le même nombre/type d'entrées.  Si la fonction de surcharge a différents types
↪de paramètres de sortie, cela provoque une erreur.
    // Les appels de fonction locaux et les appels de fonction basés sur la
↪messagerie tiennent compte de ces dérogations.
    // If you want the function to override, you need to use the
    // `override` keyword. You need to specify the `virtual` keyword again
    // if you want this function to be overridden again.
    function destroy() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
            Destructible.destroy();
        }
    }
}


// Si un constructeur prend un argument, il doit être fourni dans l'en-tête (ou dans
↪le constructeur du contrat dérivé (voir ci-dessous)).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Here, we only specify `override` and not `virtual`.
    // This means that contracts deriving from `PriceFeed`
    // cannot change the behaviour of `destroy` anymore.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Notez que ci-dessus, nous appelons `Destructible.destroy()` pour « transmettre » la demande de destruction. La façon dont cela est fait est problématique, comme vu dans l'exemple suivant :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.
↪destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ Destructible.
↪destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

Un appel à `Final.destroy()` appellera `Base2.destroy` puisque nous le demandons explicitement dans l'override, mais cet appel évitera `Base1.destroy`. La solution à ce problème est d'utiliser `super` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}


contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

Si `Base2` appelle une fonction de `super`, elle n'appelle pas simplement cette fonction sur un de ses contrats de base. Elle appelle plutôt cette fonction sur le prochain contrat de base dans le graph d'héritage final, donc elle appellera `Base1.destroy()` (notez que la séquence d'héritage finale est – en commençant par le contrat le plus dérivé : Final, Base2, Base1, Destructible, owned). La fonction réelle qui est appelée lors de l'utilisation de super n'est pas connue dans le contexte de la classe où elle est utilisée, bien que son type soit connu. Il en va de même pour la recherche de méthodes virtuelles ordinaires.

Function Overriding

Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header as shown in this example :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Base
{
    function foo() virtual public {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() public override {}
}
```

For multiple inheritance, the most derived base contracts that define the same function must be specified explicitly after the `override` keyword. In other words, you have to specify all base contracts that define the same function and have not yet been overridden by another base contract (on some path through the inheritance graph). Additionally, if a contract inherits the same function from multiple (unrelated) bases, it has to explicitly override it :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    function foo() public override(Base1, Base2) {}
}
```

An explicit override specifier is not required if the function is defined in a common base contract or if there is a unique function in a common base contract that already overrides all other functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// No explicit override required
contract D is B, C {}
```

More formally, it is not required to override a function (directly or indirectly) inherited from multiple bases if there

is a base contract that is part of all override paths for the signature, and (1) that base implements the function and no paths from the current contract to the base mentions a function with that signature or (2) that base does not implement the function and there is at most one mention of the function in all paths from the current contract to that base.

In this sense, an override path for a signature is a path through the inheritance graph that starts at the contract under consideration and ends at a contract mentioning a function with that signature that does not override.

If you do not mark a function that overrides as `virtual`, derived contracts can no longer change the behaviour of that function.

Note : Functions with the `private` visibility cannot be `virtual`.

Note : Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract A
{
    function f() external pure virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

Note : While public state variables can override external functions, they themselves cannot be overridden.

Modifier Overriding

Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

In case of multiple inheritance, all direct base contracts must be specified explicitly :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Constructeurs

Un constructeur est une fonction optionnelle déclarée avec le mot-clé `constructeur` qui est exécuté lors de la création du contrat, et où vous pouvez exécuter le code d'initialisation du contrat.

Avant l'exécution du code constructeur, les variables d'état sont initialisées à leur valeur spécifiée si vous les initialisez en ligne, ou à zéro si vous ne le faites pas.

Après l'exécution du constructeur, le code final du contrat est déployé dans la chaîne de blocs. Le déploiement du code coûte du gas supplémentaire linéairement à la longueur du code. Ce code inclut toutes les fonctions qui font partie de l'interface publique et toutes les fonctions qui sont accessibles à partir de là par des appels de fonctions. Il n'inclut pas le code constructeur ni les fonctions internes qui ne sont appelées que par le constructeur.

Les fonctions du constructeur peuvent être `public` ou `internal`. S'il n'y a pas de constructeur, le contrat assumera le constructeur par défaut, ce qui est équivalent à `constructor() public {}`. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract A {
    uint public a;

    constructor(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    constructor() public {}
}
```

Un constructeur déclaré `internal` rend le contrat *abstract*.

Attention : Avant 0.4.22, ont été définis comme des fonctions portant le même nom que le contrat. Cette syntaxe a été dépréciée et n'est plus autorisée dans la version 0.5.0.

Arguments des Constructeurs de Base

Les constructeurs de tous les contrats de base seront appelés selon les règles de linéarisation expliquées ci-dessous. Si les constructeurs de base ont des arguments, les contrats dérivés doivent les spécifier tous. Cela peut se faire de deux façons :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

contract Base {
    uint x;
    constructor(uint _x) public { x = _x; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() public {}
}

// or through a "modifier" of the derived constructor.
contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) public {}
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier is invoked as part 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. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts" constructors, it will be abstract.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses « C3 Linearization » to force a specific order in the directed acyclic graph (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 : You have to list the direct base contracts in the order from « most base-like » to « most derived ». Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error « Linearization of inheritance graph impossible ».

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

contract X {}
contract A is X {}
// This will not compile
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.

Due to the fact that you have to explicitly override a function that is inherited from multiple bases without a unique override, C3 linearization is not too important in practice.

One area where inheritance linearization is especially important and perhaps not as clear is when there are multiple constructors in the inheritance hierarchy. The constructors will always be executed in the linearized order, regardless of the order in which their arguments are provided in the inheriting contract's constructor. For example :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

contract Base1 {
    constructor() public {}
}

contract Base2 {
    constructor() public {}
}

// Constructors are executed in the following order:
//  1 - Base1
//  2 - Base2
//  3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() public Base1() Base2() {}
}

// Constructors are executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() public Base2() Base1() {}
}

// Constructors are still executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() public Base1() Base2() {}
}
```

Inheriting Different Kinds of Members of the Same Name

It is an error when any of the following pairs in a contract have the same name due to inheritance :
- a function and a modifier
- a function and an event
- an event and a modifier

As an exception, a state variable getter can override an external function.

3.9.8 Abstract Contracts

Contracts need to be marked as abstract when at least one of their functions is not implemented. Contracts may be marked as abstract even though all functions are implemented.

This can be done by using the `abstract` keyword as shown in the following example. Note that this contract needs to be defined as abstract, because the function `utterance()` was defined, but no implementation was provided (no implementation body `{ }` was given). :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

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

Such abstract contracts can not be instantiated directly. This is also true, if an abstract contract itself does implement all defined functions. The usage of an abstract contract as a base class is shown in the following example :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}

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

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it needs to be marked as abstract as well.

Note that a function without implementation is different from a *Function Type* even though their syntax looks very similar.

Example of function without implementation (a function declaration) :

```
function foo(address) external returns (address);
```

Example of a declaration of a variable whose type is a function type :

```
function(address) external returns (address) foo;
```

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the Template method and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say « any child of mine must implement this method ».

Note : Abstract contracts cannot override an implemented virtual function with an unimplemented one.

3.9.9 Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions :

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

All functions declared in interfaces are implicitly `virtual`, which means that they can be overridden. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

Interfaces can inherit from other interfaces. This has the same rules as normal inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Types defined inside interfaces and other contract-like structures can be accessed from other contracts : `Token.TokenType` or `Token.Coin`.

3.9.10 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). Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

Note : Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a *mechanism* that disallows state-modifying functions to be called directly (i.e. without

DELEGATECALL).

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 types *stored in memory* will be passed by reference and not copied. To realize this in the EVM, code of internal library functions and all functions called from therein will at compile time be included in the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

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

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;


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

library Set {
    // 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)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // 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, and they 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, be aware 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 *types stored in memory* and internal functions in libraries in order to implement custom types without the overhead of external function calls :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

struct bigint {
    uint[] limbs;
}

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

    function add(bigint memory _a, bigint memory _b) internal pure returns (bigint␣
↪memory 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);
            uint i;
```

(suite sur la page suivante)

(suite de la page précédente)

```
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

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

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

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(uint(-1));
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

It is possible to obtain the address of a library by converting the library type to the `address` type, i.e. using `address(LibraryName)`.

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.

Note : Manually linking libraries on the generated bytecode is discouraged, because in this way, the library name is restricted to 36 characters. You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

In comparison to contracts, libraries are restricted in the following ways :

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

(These might be lifted at a later point.)

Function Signatures and Selectors in Libraries

While external calls to public or external library functions are possible, the calling convention for such calls is considered to be internal to Solidity and not the same as specified for the regular *contract ABI*. External library functions support more argument types than external contract functions, for example recursive structs and storage pointers. For

that reason, the function signatures used to compute the 4-byte selector are computed following an internal naming schema and arguments of types not supported in the contract ABI use an internal encoding.

The following identifiers are used for the types in the signatures :

- Value types, non-storage `string` and non-storage `bytes` use the same identifiers as in the contract ABI.
- Non-storage array types follow the same convention as in the contract ABI, i.e. `<type>[]` for dynamic arrays and `<type>[M]` for fixed-size arrays of `M` elements.
- Non-storage structs are referred to by their fully qualified name, i.e. `C.S` for `contract C { struct S { ... } }`.
- Storage pointer types use the type identifier of their corresponding non-storage type, but append a single space followed by `storage` to it.

The argument encoding is the same as for the regular contract ABI, except for storage pointers, which are encoded as a `uint256` value referring to the storage slot to which they point.

Similarly to the contract ABI, the selector consists of the first four bytes of the Keccak256-hash of the signature. Its value can be obtained from Solidity using the `.selector` member as follows :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.7.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a `CALL` instead of a `DELEGATECALL` or `CALLCODE`, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using `CALL` or not, but a contract can use the `ADDRESS` opcode to find out « where » it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

This means that the actual code stored on chain for a library is different from the code reported by the compiler as `deployedBytecode`.

3.9.11 Using For

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`) in the context of a contract. 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 in the library are attached, even those where the type of the first parameter does not match the type of the object. 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 only within the current contract, including within all of its functions, and has no effect outside of the contract in which it is used. The directive may only be used inside a contract, not inside any of its functions.

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

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;


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

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

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

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


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

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

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

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

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        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) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // 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 external 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 or when internal library functions are called.

3.10 Assembleur en ligne

Vous pouvez entrelacer les instructions en Solidity avec de l'assembleur en ligne, dans un langage proche de celui de la machine virtuelle. Cela vous donne un contrôle plus fin, en particulier lorsque vous améliorez le langage en écrivant des bibliothèques.

Le langage utilisé pour l'assembleur en ligne en Solidity s'appelle *Yul* et est documenté dans sa propre section. Cette section montre comment le code assembleur en ligne peut s'interfacer au code Solidity l'entourant.

Avertissement : L'assembleur en ligne est un moyen d'accéder à la machine virtuelle Ethereum en bas niveau. Ceci permet de contourner plusieurs normes de sécurité importantes et contrôles de Solidity. Vous ne devriez

l'utiliser que pour les tâches qui en ont besoin, et seulement si vous êtes sûr de pourquoi/comment l'utiliser.

Le bloc de code d'assembleur en ligne est indiqué par `assembly { ... }`, où le code entre les accolades est écrit en langage *Yul*.

Le bloc de code assembleur en ligne peut accéder aux variables locales de Solidity comme expliqué ci-dessous.

Différents bloc de coe assembleur ne partagent pas le même espace de noms, c'est à dire qu'il n'est pas possible d'appeler une fonction Yul où d'accéder à une variable Yul variable definie dans un autre bloc.

3.10.1 Exemple

L'exemple suivant fournit le code de bibliothèque pour accéder au code d'un autre contrat et le charger dans une variable `bytes`. Ce n'est pas possible de base avec Solidity et l'idée est que les bibliothèques assembleur seront utilisées pour améliorer le langage Solidity.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

library GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // récupère la taille du code, a besoin d'assembleur
            let size := extcodesize(_addr)
            // allouer le tableau de bytes de sortie - ceci serait fait en Solidity␣
↪via o_code = new bytes(size)
            o_code := mload(0x40)
            // nouvelle "fin de mémoire" en incluant le padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // stocke la taille en mémoire
            mstore(o_code, size)
            // récupère le code lui-même, nécessite de l'assembleur
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

L'assembleur en ligne est également utile dans les cas où l'optimiseur ne parvient pas à produire un code efficace, par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

library VectorSum {
    // Cette fonction est moins efficace car l'optimiseur ne parvient
    // pas à supprimer les contrôles de limites dans l'accès aux tableaux.
    function sumSolidity(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // Nous savons que nous n'accédons au tableau que dans ses
    // limites, ce qui nous permet d'éviter la vérification. 0x20
    // doit être ajouté à un tableau car le premier emplacement
    // contient la longueur du tableau.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function sumAsm(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := add(o_sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Même chose que ci-dessus, mais exécute le code entier en assembleur en ligne.
    function sumPureAsm(uint[] memory _data) public pure returns (uint o_sum) {
        assembly {
            // Charge la taille (premiers 32 bytes)
            let len := mload(_data)

            // Saute le champ de taille.
            //
            // Garde une variable temporaire pour pouvoir l'incrémenter.
            //
            // NOTE: incrémenter _data resulterait en une
            // variable _data inutilisable après ce bloc d'assembleur
            let data := add(_data, 0x20)

            // Itère jusqu'à la limite.
            for
                { let end := add(data, mul(len, 0x20)) }
                lt(data, end)
                { data := add(data, 0x20) }
            {
                o_sum := add(o_sum, mload(data))
            }
        }
    }
}
```

3.10.2 Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name.

Local variables of value type are directly usable in inline assembly.

Local variables that refer to memory or calldata evaluate to the address of the variable in memory, resp. calldata, not the value itself.

For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their « address » is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x_slot`, and to retrieve the byte-offset you use `x_offset`.

Local Solidity variables are available for assignments, for example :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b_slot))
        }
    }
}
```

Avertissement : If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, `bytes16` or `byte`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important : `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode : `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Since Solidity 0.6.0 the name of a inline assembly variable may not end in `_offset` or `_slot` and it may not shadow any declaration visible in the scope of the inline assembly block (including variable, contract and function declarations). Similarly, if the name of a declared variable contains a dot `.`, the prefix up to the `.` may not conflict with any declaration visible in the scope of the inline assembly block.

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.

You can assign to the `_slot` part of a local storage variable pointer. For these (structs, arrays or mappings), the `_offset` part is always zero. It is not possible to assign to the `_slot` or `_offset` part of a state variable, though.

3.10.3 Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs 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, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

3.10.4 Conventions in Solidity

In contrast to EVM assembly, Solidity has types which are narrower than 256 bits, e.g. `uint24`. For efficiency, most arithmetic operations ignore the fact that types can be shorter than 256 bits, and the higher-order bits are cleaned when necessary, i.e., 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 the following way. There is a « free memory pointer » at position `0x40` in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet you can use for allocating memory that follows the process outlined above :

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

The first 64 bytes of memory can be used as « scratch space » for short-term allocation. The 32 bytes after the free memory pointer (i.e., starting at `0x60`) are meant to be zero permanently and is used as the initial value for empty

dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (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 followed by the array elements.

Avertissement : Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically- and dynamically-sized arrays, so do not rely on this.

3.11 Cheatsheet

3.11.1 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	`++`, `--`
	New expression	`new <typename>`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Function-like call	`<func>(<args...>)`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++`, `--`
	Unary 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>`
	Assignment operators	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
15	Comma operator	`,`

3.11.2 Global Variables

— `abi.decode(bytes memory encodedData, (...)) returns (...)` : *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
— `abi.encode(...) returns (bytes memory)` : *ABI*-encodes the given arguments

— `abi.encodePacked(...) returns (bytes memory)` : Performs *packed encoding* of the given arguments. Note that this encoding can be ambiguous !
— `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : *ABI*-encodes the given arguments starting from the second and prepends the given four-byte selector
— `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)` `
— `block.coinbase` (`address payable`) : 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
— `gasleft() returns (uint256)` : remaining gas
— `msg.data` (`bytes`) : complete calldata
— `msg.sender` (`address payable`) : 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 payable`) : sender of the transaction (full call chain)
— `assert(bool condition)` : abort execution and revert state changes if condition is `false` (use for internal error)
— `require(bool condition)` : abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
— `require(bool condition, string memory message)` : abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
— `revert()` : abort execution and revert state changes
— `revert(string memory message)` : abort execution and revert state changes providing an explanatory string
— `blockhash(uint blockNumber) returns (bytes32)` : hash of the given block - only works for 256 most recent blocks
— `keccak256(bytes memory) returns (bytes32)` : compute the Keccak-256 hash of the input
— `sha256(bytes memory) returns (bytes32)` : compute the SHA-256 hash of the input
— `ripemd160(bytes memory) returns (bytes20)` : compute the RIPEMD-160 hash of the input
— `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`. Assert that `k != 0` starting from version 0.5.0.
— `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`. Assert that `k != 0` starting from version 0.5.0.
— `this` (current contract's type) : the current contract, explicitly convertible to `address` or `address payable`
— `super` : the contract one level higher in the inheritance hierarchy
— `selfdestruct(address payable recipient)` : destroy the current contract, sending its funds to the given address
— `<address>.balance` (`uint256`) : balance of the *Adresses* in Wei
— `<address payable>.send(uint256 amount) returns (bool)` : send given amount of Wei to *Adresses*, returns `false` on failure
— `<address payable>.transfer(uint256 amount)` : send given amount of Wei to *Adresses*, throws on failure
— `type(C).name` (`string`) : the name of the contract
— `type(C).creationCode` (`bytes memory`) : creation bytecode of the given contract, see *Type Information*.

— `type(C).runtimeCode` (`bytes memory`) : runtime bytecode of the given contract, see *Type Information*.
— `type(I).interfaceId` (`bytes4`) : value containing the EIP-165 interface identifier of the given interface, see *Type Information*.
— `type(T).min` (`T`) : the minimum value representable by the integer type `T`, see *Type Information*.
— `type(T).max` (`T`) : the maximum value representable by the integer type `T`, see *Type Information*.

Note : Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

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.

Note : In version 0.5.0, the following aliases were removed : `suicide` as alias for `selfdestruct`, `msg.gas` as alias for `gasleft`, `block.blockhash` as alias for `blockhash` and `sha3` as alias for `keccak256`.

3.11.3 Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

— `public` : visible externally and internally (creates a *getter 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

3.11.4 Modifiers

— `pure` for functions : Disallows modification or access of state.
— `view` for functions : Disallows modification of state.
— `payable` for functions : Allows them to receive Ether together with a call.
— `constant` for state variables : Disallows assignment (except initialisation), does not occupy storage slot.
— `immutable` for state variables : Allows exactly one assignment at construction time and is constant afterwards. Is stored in code.
— `anonymous` for events : Does not store event signature as topic.
— `indexed` for event parameters : Stores the parameter as topic.
— `virtual` for functions and modifiers : Allows the function's or modifier's behaviour to be changed in derived contracts.
— `override` : States that this function, modifier or public state variable changes the behaviour of a function or modifier in a base contract.

3.11.5 Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future :

`after`, `alias`, `apply`, `auto`, `case`, `copyof`, `default`, `define`, `final`, `immutable`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `unchecked`.

3.12 Language Grammar

```
// Copyright 2020 Gonçalo Sá <goncalo.sa@consensys.net>
// Copyright 2016-2019 Federico Bond <federicobond@gmail.com>
// Licensed under the MIT license. See LICENSE file in the project root for details.

// This grammar is much less strict than what Solidity currently parses
// to allow this to pass with older versions of Solidity.

grammar Solidity;

sourceUnit
  : (pragmaDirective | importDirective | structDefinition | enumDefinition |␣
↪contractDefinition)* EOF ;

pragmaDirective
  : 'pragma' pragmaName pragmaValue ';' ;

pragmaName
  : identifier ;

pragmaValue
  : version | expression ;

version
  : versionConstraint versionConstraint? ;

versionConstraint
  : versionOperator? VersionLiteral ;

versionOperator
  : '^' | '~' | '>=' | '>' | '<' | '<=' | '=' ;

importDirective
  : 'import' StringLiteralFragment ('as' identifier)? ';'
  | 'import' ('*' | identifier) ('as' identifier)? 'from' StringLiteralFragment ';'
  | 'import' '{' importDeclaration ( ',' importDeclaration )* '}' 'from'␣
↪StringLiteralFragment ';' ;

importDeclaration
  : identifier ('as' identifier)? ;

contractDefinition
  : 'abstract'? ( 'contract' | 'interface' | 'library' ) identifier
    ( 'is' inheritanceSpecifier (',' inheritanceSpecifier )* )?
    '{' contractPart* '}' ;

inheritanceSpecifier
```

(suite sur la page suivante)

(suite de la page précédente)

```
  : userDefinedTypeName ( '(' expressionList? ')' )? ;

contractPart
  : stateVariableDeclaration
  | usingForDeclaration
  | structDefinition
  | modifierDefinition
  | functionDefinition
  | eventDefinition
  | enumDefinition ;

stateVariableDeclaration
  : typeName
    ( PublicKeyword | InternalKeyword | PrivateKeyword | ConstantKeyword |␣
↪ImmutableKeyword | overrideSpecifier )*
    identifier ('=' expression)? ';' ;

overrideSpecifier : 'override' ( '(' userDefinedTypeName (',' userDefinedTypeName)* ')
↪' )? ;

usingForDeclaration
  : 'using' identifier 'for' ('*' | typeName) ';' ;

structDefinition
  : 'struct' identifier
    '{' ( variableDeclaration ';' (variableDeclaration ';')* )? '}' ;

modifierDefinition
  : 'modifier' identifier parameterList? ( VirtualKeyword | overrideSpecifier )* ( ';
↪' | block ) ;

functionDefinition
  : functionDescriptor parameterList modifierList returnParameters? ( ';' | block ) ;

functionDescriptor
  : 'function' ( identifier | ReceiveKeyword | FallbackKeyword )?
  | ConstructorKeyword
  | FallbackKeyword
  | ReceiveKeyword ;

returnParameters
  : 'returns' parameterList ;

modifierList
  : ( modifierInvocation | stateMutability | ExternalKeyword
    | PublicKeyword | InternalKeyword | PrivateKeyword | VirtualKeyword |␣
↪overrideSpecifier )* ;

modifierInvocation
  : identifier ( '(' expressionList? ')' )? ;

eventDefinition
  : 'event' identifier eventParameterList AnonymousKeyword? ';' ;

enumDefinition
  : 'enum' identifier '{' enumValue? (',' enumValue)* '}' ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
enumValue
  : identifier ;

parameterList
  : '(' ( parameter (',' parameter)* )? ')' ;

parameter
  : typeName storageLocation? identifier? ;

eventParameterList
  : '(' ( eventParameter (',' eventParameter)* )? ')' ;

eventParameter
  : typeName IndexedKeyword? identifier? ;

variableDeclaration
  : typeName storageLocation? identifier ;

typeName
  : elementaryTypeName
  | userDefinedTypeName
  | mapping
  | typeName '[' expression? ']'
  | functionTypeName ;

userDefinedTypeName
  : identifier ( '.' identifier )* ;

mapping
  : 'mapping' '(' (elementaryTypeName | userDefinedTypeName) '=>' typeName ')' ;

functionTypeName
  : 'function' parameterList modifierList returnParameters? ;

storageLocation
  : 'memory' | 'storage' | 'calldata';

stateMutability
  : PureKeyword | ConstantKeyword | ViewKeyword | PayableKeyword ;

block
  : '{' statement* '}' ;

statement
  : ifStatement
  | tryStatement
  | whileStatement
  | forStatement
  | block
  | inlineAssemblyStatement
  | doWhileStatement
  | continueStatement
  | breakStatement
  | returnStatement
  | throwStatement
  | emitStatement
  | simpleStatement ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
expressionStatement
  : expression ';' ;

ifStatement
  : 'if' '(' expression ')' statement ( 'else' statement )? ;

tryStatement : 'try' expression returnParameters? block catchClause+ ;

// In reality catch clauses still are not processed as below
// the identifier can only be a set string: "Error". But plans
// of the Solidity team include possible expansion so we'll
// leave this as is, befitting with the Solidity docs.
catchClause : 'catch' ( identifier? parameterList )? block ;

whileStatement
  : 'while' '(' expression ')' statement ;

forStatement
  : 'for' '(' ( simpleStatement | ';' ) ( expressionStatement | ';' ) expression? ')'␣
↪statement ;

simpleStatement
  : ( variableDeclarationStatement | expressionStatement ) ;

inlineAssemblyStatement
  : 'assembly' StringLiteralFragment? assemblyBlock ;

doWhileStatement
  : 'do' statement 'while' '(' expression ')' ';' ;

continueStatement
  : 'continue' ';' ;

breakStatement
  : 'break' ';' ;

returnStatement
  : 'return' expression? ';' ;

// throw is no longer supported by latest Solidity.
throwStatement
  : 'throw' ';' ;

emitStatement
  : 'emit' functionCall ';' ;

// 'var' is no longer supported by latest Solidity.
variableDeclarationStatement
  : ( 'var' identifierList | variableDeclaration | '(' variableDeclarationList ')' )␣
↪( '=' expression )? ';';

variableDeclarationList
  : variableDeclaration? (',' variableDeclaration? )* ;

identifierList
  : '(' ( identifier? ',' )* identifier? ')' ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
elementaryTypeName
  : 'address' PayableKeyword? | 'bool' | 'string' | 'var' | Int | Uint | 'byte' |␣
↪Byte | Fixed | Ufixed ;

Int
  : 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' |
↪'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' |
↪'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184
↪' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' |
↪'int248' | 'int256' ;

Uint
  : 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' | 'uint56
↪' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' | 'uint112' |
↪'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' | 'uint168' |
↪'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' | 'uint224' |
↪'uint232' | 'uint240' | 'uint248' | 'uint256' ;

Byte
  : 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' | 'bytes6' |
↪'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' | 'bytes13' |
↪'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' | 'bytes20' |
↪'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' | 'bytes27' |
↪'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32' ;

Fixed
  : 'fixed' | ( 'fixed' [0-9]+ 'x' [0-9]+ ) ;

Ufixed
  : 'ufixed' | ( 'ufixed' [0-9]+ 'x' [0-9]+ ) ;

expression
  : expression ('++' | '--')
  | 'new' typeName
  | expression '[' expression? ']'
  | expression '[' expression? ':' expression? ']'
  | expression '.' identifier
  | expression '{' nameValueList '}'
  | expression '(' functionCallArguments ')'
  | PayableKeyword '(' expression ')'
  | '(' expression ')'
  | ('++' | '--') expression
  | ('+' | '-') expression
  | ('after' | 'delete') expression
  | '!' expression
  | '~' expression
  | expression '**' expression
  | expression ('*' | '/' | '%') expression
  | expression ('+' | '-') expression
  | expression ('<<' | '>>') expression
  | expression '&' expression
  | expression '^' expression
  | expression '|' expression
  | expression ('<' | '>' | '<=' | '>=') expression
  | expression ('==' | '!=') expression
  | expression '&&' expression
```

(suite sur la page suivante)

(suite de la page précédente)

```
  | expression '||' expression
  | expression '?' expression ':' expression
  | expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/='␣
↪| '%=') expression
  | primaryExpression ;

primaryExpression
  : BooleanLiteral
  | numberLiteral
  | hexLiteral
  | stringLiteral
  | identifier ('[' ']')?
  | TypeKeyword
  | tupleExpression
  | typeNameExpression ('[' ']')? ;

expressionList
  : expression (',' expression)* ;

nameValueList
  : nameValue (',' nameValue)* ','? ;

nameValue
  : identifier ':' expression ;

functionCallArguments
  : '{' nameValueList? '}'
  | expressionList? ;

functionCall
  : expression '(' functionCallArguments ')' ;

tupleExpression
  : '(' ( expression? ( ',' expression? )* ) ')'
  | '[' ( expression ( ',' expression )* )? ']' ;

typeNameExpression
  : elementaryTypeName
  | userDefinedTypeName ;

assemblyItem
  : identifier
  | assemblyBlock
  | assemblyExpression
  | assemblyLocalDefinition
  | assemblyAssignment
  | assemblyStackAssignment
  | labelDefinition
  | assemblySwitch
  | assemblyFunctionDefinition
  | assemblyFor
  | assemblyIf
  | BreakKeyword
  | ContinueKeyword
  | LeaveKeyword
  | subAssembly
  | numberLiteral
```

(suite sur la page suivante)

(suite de la page précédente)

```
  | stringLiteral
  | hexLiteral ;

assemblyBlock
  : '{' assemblyItem* '}' ;

assemblyExpression
  : assemblyCall | assemblyLiteral ;

assemblyCall
  : ( 'return' | 'address' | 'byte' | identifier ) ( '(' assemblyExpression? ( ',' ␣
↪assemblyExpression )* ')' )? ;

assemblyLocalDefinition
  : 'let' assemblyIdentifierList ( ':=' assemblyExpression )? ;

assemblyAssignment
  : assemblyIdentifierList ':=' assemblyExpression ;

assemblyIdentifierList
  : identifier ( ',' identifier )* ;

assemblyStackAssignment
  : '=:' identifier ;

labelDefinition
  : identifier ':' ;

assemblySwitch
  : 'switch' assemblyExpression assemblyCase* ;

assemblyCase
  : 'case' assemblyLiteral assemblyType? assemblyBlock
  | 'default' assemblyBlock ;

assemblyFunctionDefinition
  : 'function' identifier '(' assemblyTypedVariableList? ')'
    assemblyFunctionReturns? assemblyBlock ;

assemblyFunctionReturns
  : ( '-' '>' assemblyTypedVariableList ) ;

assemblyFor
  : 'for' assemblyBlock assemblyExpression assemblyBlock assemblyBlock ;

assemblyIf
  : 'if' assemblyExpression assemblyBlock ;

assemblyLiteral
  : ( stringLiteral | DecimalNumber | HexNumber | hexLiteral | BooleanLiteral ) ␣
↪assemblyType? ;

assemblyTypedVariableList
  : identifier assemblyType? ( ',' assemblyTypedVariableList )? ;

assemblyType
  : ':' identifier ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
subAssembly
  : 'assembly' identifier assemblyBlock ;

numberLiteral
  : (DecimalNumber | HexNumber) NumberUnit? ;

identifier
  : ('from' | 'calldata' | 'address' | Identifier) ;

BooleanLiteral
  : 'true' | 'false' ;

DecimalNumber
  : ( DecimalDigits | (DecimalDigits? '.' DecimalDigits) ) ( [eE] '-'? DecimalDigits␣
↪)? ;

fragment
DecimalDigits
  : [0-9] ( '_'? [0-9] )* ;

HexNumber
  : '0' [xX] HexDigits ;

fragment
HexDigits
  : HexCharacter ( '_'? HexCharacter )* ;

NumberUnit
  : 'wei' | 'szabo' | 'finney' | 'ether'
  | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years' ;

HexLiteralFragment
  : 'hex' (('"' HexDigits? '"') | ('\'' HexDigits? '\'')) ;

hexLiteral : HexLiteralFragment+ ;

fragment
HexPair
  : HexCharacter HexCharacter ;

fragment
HexCharacter
  : [0-9A-Fa-f] ;

ReservedKeyword
  : 'after'
  | 'case'
  | 'default'
  | 'final'
  | 'in'
  | 'inline'
  | 'let'
  | 'match'
  | 'null'
  | 'of'
  | 'relocatable'
```

(suite sur la page suivante)

(suite de la page précédente)

```
  | 'static'
  | 'switch'
  | 'typeof' ;

AnonymousKeyword : 'anonymous' ;
BreakKeyword : 'break' ;
ConstantKeyword : 'constant' ;
ImmutableKeyword : 'immutable' ;
ContinueKeyword : 'continue' ;
LeaveKeyword : 'leave' ;
ExternalKeyword : 'external' ;
IndexedKeyword : 'indexed' ;
InternalKeyword : 'internal' ;
PayableKeyword : 'payable' ;
PrivateKeyword : 'private' ;
PublicKeyword : 'public' ;
VirtualKeyword : 'virtual' ;
PureKeyword : 'pure' ;
TypeKeyword : 'type' ;
ViewKeyword : 'view' ;

ConstructorKeyword : 'constructor' ;
FallbackKeyword : 'fallback' ;
ReceiveKeyword : 'receive' ;

Identifier
  : IdentifierStart IdentifierPart* ;

fragment
IdentifierStart
  : [a-zA-Z$_] ;

fragment
IdentifierPart
  : [a-zA-Z0-9$_] ;

stringLiteral
  : StringLiteralFragment+ ;

StringLiteralFragment
  : '"' DoubleQuotedStringCharacter* '"'
  | '\'' SingleQuotedStringCharacter* '\'' ;

fragment
DoubleQuotedStringCharacter
  : ~["\r\n\\] | ('\\' .) ;

fragment
SingleQuotedStringCharacter
  : ~['\r\n\\] | ('\\' .) ;

VersionLiteral
  : [0-9]+ '.' [0-9]+ ('.' [0-9]+)? ;

WS
  : [ \t\r\n\u000C]+ -> skip ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
COMMENT
  : '/*' .*? '*/' -> channel(HIDDEN) ;

LINE_COMMENT
  : '//' ~[\r\n]* -> channel(HIDDEN) ;
```

3.13 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, contiguous 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 as many bytes as 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).

For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

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

Avertissement : 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.

Note : The layout of state variables in storage is considered to be part of the external interface of Solidity due to the fact that storage pointers can be passed to libraries. This means that any change to the rules outlined in this section is considered a breaking change of the language and due to its critical nature should be considered very carefully before being executed.

3.13.1 Mappings and Dynamic Arrays

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 a slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings of mappings or arrays of arrays). For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see *below*). For mappings, 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)`.

So for the following contract snippet the position of `data[4][9].b` is at `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;


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

`bytes` and `string`

`bytes` and `string` are encoded identically. For short byte arrays, they store their data in the same slot where the length is also stored. 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`. For byte arrays that store data which is `32` or more bytes long, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set : short (not set) and long (set).

Note : Handling invalidly encoded slots is currently not supported but may be added in the future.

3.13.2 JSON Output

The storage layout of a contract can be requested via the *standard JSON interface*. The output is a JSON object containing two keys, `storage` and `types`. The `storage` object is an array where each element has the following form :

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

The example above is the storage layout of `contract A { uint x; }` from source unit `fileA` and

- `astId` is the id of the AST node of the state variable's declaration
- `contract` is the name of the contract including its path as prefix
- `label` is the name of the state variable
- `offset` is the offset in bytes within the storage slot according to the encoding
- `slot` is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- `type` is an identifier used as key to the variable's type information (described in the following)

The given `type`, in this case `t_uint256` represents an element in `types`, which has the form :

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

where

- `encoding` how the data is encoded in storage, where the possible values are :
  - `inplace` : data is laid out contiguously in storage (see *above*).
  - `mapping` : Keccak-256 hash-based method (see *above*).
  - `dynamic_array` : Keccak-256 hash-based method (see *above*).
  - `bytes` : single slot or Keccak-256 hash-based depending on the data size (see *above*).
- `label` is the canonical type name.
- `numberOfBytes` is the number of used bytes (as a decimal string). Note that if `numberOfBytes > 32` this means that more than one slot is used.

Some types have extra information besides the four above. Mappings contain its `key` and `value` types (again referencing an entry in this mapping of types), arrays have its `base` type, and structs list their `members` in the same format as the top-level `storage` (see *above*).

Note : The JSON output format of a contract's storage layout is still considered experimental and is subject to change in non-breaking releases of Solidity.

The following example shows a contract and its storage layout, containing value and reference types, types that are encoded packed, and nested types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping (uint => mapping (address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

```
"storageLayout": {
  "storage": [
    {
      "astId": 14,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
```

(suite sur la page suivante)

(suite de la page précédente)


```
    },
    {
      "astId": 16,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 18,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)12_storage"
    },
    {
      "astId": 20,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 26,
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "astId": 29,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 31,
      "contract": "fileA:A",
      "label": "s1",
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
    },
    {
      "astId": 33,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
```


(suite sur la page suivante)

(suite de la page précédente)

```
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)12_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 2,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
```

(suite sur la page suivante)

(suite de la page précédente)

```
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 4,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 8,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 11,
          "contract": "fileA:A",
          "label": "dynArray",
          "offset": 0,
          "slot": "3",
          "type": "t_array(t_uint256)dyn_storage"
        }
      ],
      "numberOfBytes": "128"
    },
    "t_uint128": {
      "encoding": "inplace",
      "label": "uint128",
      "numberOfBytes": "16"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

3.14 Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows :

- `0x00` - `0x3f` (64 bytes) : scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes) : currently allocated memory size (aka. free memory pointer)
- `0x60` - `0x7f` (32 bytes) : zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to `0x80` initially).

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

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (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 followed by the array elements.

Avertissement : 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 lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one should not expect the free memory to point to zeroed out memory.

While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have unexpected results.

3.15 Layout of Call Data

The input data for a function call is assumed to be in the format defined by the *ABI specification*. Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

3.16 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 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

3.17 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. This mapping also contains other information, like the jump type and the modifier depth (see below).

Both kinds of source mappings use integer identifiers to refer to source files. The identifier of a source file is stored in `output['sources'][sourceName]['id']` where `output` is the output of the standard-json compiler interface parsed as JSON.

Note : In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of `-1`. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

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:m` 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. `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. The last field, `m`, is an integer that denotes the « modifier depth ». This depth is increased whenever the placeholder statement (_) is entered in a modifier and decreased when it is left again. This allows debuggers to track tricky cases like the same modifier being used twice or multiple placeholder statements being used in a single modifier.

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;2:1:2;;`

3.18 The Optimiser

This section discusses the optimiser that was first added to Solidity, which operates on opcode streams. For information on the new Yul-based optimiser, please see the readme on github.

The Solidity optimiser operates on assembly. It splits the sequence of instructions into basic blocks at `JUMP`s and `JUMPDEST`s. Inside these blocks, the optimiser analyses the instructions and records every modification to the stack, memory, or storage as an expression which consists of an instruction and a list of arguments which are pointers to other expressions. The optimiser uses a component called « CommonSubexpressionEliminator » that amongst other tasks, finds expressions that are always equal (on every input) and combines them into an expression class. The optimiser first tries to find each new expression in a list of already known expressions. If this does not work, it simplifies the expression according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is a recursive process, we can also apply the latter rule if the second factor is a more complex expression where we know that it always evaluates 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 do not know what is stored at x after we wrote to y. If 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.

After this process, we know which expressions have to be on the stack at 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 the optimiser finds a `JUMPI` whose condition evaluates to a constant, it transforms it to an unconditional jump.

As the last step, the code in each block is re-generated. The optimiser creates a dependency graph from the expressions on the stack at the end of the block, and it drops every operation that is not part of this graph. It generates code 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). Finally, it 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. Thus code like

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

still simplifies to code which you can compile even though the instructions contained a jump in the beginning of the process :

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

3.19 Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the compiled contract. You can use this file to query the compiler version, the sources used, the ABI and NatSpec documentation to more safely interact with the contract and verify its source code.

The compiler appends by default the IPFS 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. The other available options are the Swarm hash and not appending the metadata hash to the bytecode. These can be configured via the *Standard JSON Interface*.

You have to publish the metadata file to IPFS, Swarm, or another service so that others can access it. You create the file by using the `solc --metadata` command that generates a file called `ContractName_meta.json`. It contains IPFS and 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 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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..." ],
      // Optional: SPDX license identifier as given in the source file
      "license": "MIT"
    },
    "destructible": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract destructible is owned { function destroy() { if (msg.
↪sender == owner) selfdestruct(owner); } }"
    }
  },
  // Required: Compiler settings
  settings:
  {
    // Required for Solidity: Sorted list of remappings
    remappings: [ ":g=/dir" ],
    // Optional: Optimizer settings. The fields "enabled" and "runs" are deprecated
    // and are only given for backwards-compatibility.
    optimizer: {
      enabled: true,
      runs: 500,
      details: {
        // peephole defaults to "true"
        peephole: true,
        // jumpdestRemover defaults to "true"
        jumpdestRemover: true,
        orderLiterals: false,
        deduplicate: false,
        cse: false,
        constantOptimizer: false,
        yul: true,
        // Optional: Only present if "yul" is "true"
        yulDetails: {
          stackAllocation: false,
          optimizerSteps: "dhfoDgvulfnTUtnIf..."
        }
      }
    },
```

(suite sur la page suivante)

(suite de la page précédente)

```
    metadata: {
      // Reflects the setting used in the input json, defaults to false
      useLiteralContent: true,
      // Reflects the setting used in the input json, defaults to "ipfs"
      bytecodeHash: "ipfs"
    }
    // 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: [ ... ],
  }
}
```

Avertissement : Since the bytecode of the resulting contract contains the metadata hash by default, any change to the metadata might result in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

Note : The ABI definition above has no fixed order. It can change with compiler versions. Starting from Solidity version 0.5.12, though, the array maintains a certain order.

3.19.1 Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping `{"ipfs": <IPFS hash>, "solc": <compiler version>}` is stored CBOR-encoded. Since the mapping might contain more keys (see below) and 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 usually adds the following to the end of the deployed bytecode :

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 bytes IPFS hash>
0x64 's' 'o' 'l' 'c' 0x43 <3 byte version encoding>
0x00 0x33
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and use the IPFS hash to retrieve the file.

Whereas release builds of solc use a 3 byte encoding of the version as shown above (one byte each for major, minor and patch version number), prerelease builds will instead use a complete version string including commit hash and build date.

Note : The CBOR mapping can also contain other keys, so it is better to fully decode the data instead of relying on it starting with `0xa264`. For example, if any experimental features that affect code generation are used, the mapping will also contain `"experimental": true`.

Note : The compiler currently uses the IPFS hash of the metadata by default, but it may also use the bzzr1 hash or some other hash in the future, so do not rely on this sequence to start with `0xa2 0x64 'i' 'p' 'f' 's'`. We might also add additional data to this CBOR structure, so the best option is to use a proper CBOR parser.

3.19.2 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 or any wallet) retrieves the code of the contract, from that the IPFS/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, the wallet can use the NatSpec user documentation to display a confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

For additional information, read *Ethereum Natural Language Specification (NatSpec) format*.

3.19.3 Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from IPFS/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.

In the repository source-verify (npm package) you can see example code that shows how to use this feature.

3.20 Contract ABI Specification

3.20.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time.

3.20.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Note : The return type of a function is not part of this signature. In *Solidity's function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

3.20.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

3.20.4 Types

The following elementary types exist :

- `uint<M>` : unsigned integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>` : two's complement signed integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address` : equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int` : synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool` : equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>` : signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M % 8 ==0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>` : unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed` : synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>` : binary type of `M` bytes, `0 < M <= 32`.
- `function` : an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists :

- `<type>[M]` : a fixed-length array of `M` elements, `M >= 0`, of the given type.

The following non-fixed-size types exist :

- `bytes` : dynamic sized byte sequence.
- `string` : dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]` : a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas :

- `(T1,T2,...,Tn)` : tuple consisting of the types `T1`, ..., `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are

not part of the ABI, and on the right column the ABI types that represent them.

Solidity	ABI
address payable	`address`
contract	`address`
enum	smallest `uint` type that is large enough to hold all values For example, an `enum` of 255 values or less is mapped to `uint8` and an `enum` of 256 values is mapped to `uint16`.
struct	`tuple`

3.20.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays :

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.

2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative « addresses ».

3.20.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition : The following types are called « dynamic » :
- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k >= 0`
- `(T1,...,Tk)` if `Ti` is dynamic for some `1 <= i <= k`

All other types are called « static ».

Definition : `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition : For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being
- `(T1,...,Tk)` for `k >= 0` and any types `T1`, ..., `Tk`
  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`
  where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for `Ti` as follows :
  if `Ti` is static :

    `head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

  otherwise, i.e. if `Ti` is dynamic :

    `head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) )) tail(X(i)) = enc(X(i))`

  Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. The value of `head(X(i))` is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:
  `enc(X) = enc((X[0], ..., X[k-1]))`
  i.e. it is encoded as if it were a tuple with `k` elements of the same type.
- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):
  `enc(X) = enc(k) enc([X[0], ..., X[k-1]])`
  i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.
- `bytes`, of length `k` (which is assumed to be of type `uint256`):
  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.
- `string`:
  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.
- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.
- `address`: as in the `uint160` case
- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` bytes for negative `X` and with zero-bytes for non-negative `X` such that the length is 32 bytes.
- `bool`: as in the `uint8` case, where `1` is used for `true` and `0` for `false`
- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.
- `fixed`: as in the `fixed128x18` case
- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.
- `ufixed`: as in the `ufixed128x18` case
- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

3.20.7 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

3.20.8 Examples

Given the contract :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Thus for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into :

- `0xcdcd77c0` : the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.

— `0x0000000000000000000000000000000000000000000000000000000000000045` : the first parameter, a uint32 value `69` padded to 32 bytes
— `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter - boolean `true`, padded to 32 bytes

In total :

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into :

— `0xfce353f6` : the Method ID. This is derived from the signature `bar(bytes3[2])`.
— `0x6162630000000000000000000000000000000000000000000000000000000000` : the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
— `0x6465660000000000000000000000000000000000000000000000000000000000` : the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total :

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000000006465660000000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into :

— `0xa5643bf2` : the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
— `0x0000000000000000000000000000000000000000000000000000000000000060` : the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
— `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter : boolean true.
— `0x00000000000000000000000000000000000000000000000000000000000000a0` : the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
— `0x0000000000000000000000000000000000000000000000000000000000000004` : the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
— `0x6461766500000000000000000000000000000000000000000000000000000000` : the contents of the first argument : the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
— `0x0000000000000000000000000000000000000000000000000000000000000003` : the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
— `0x0000000000000000000000000000000000000000000000000000000000000001` : the first entry of the third parameter.
— `0x0000000000000000000000000000000000000000000000000000000000000002` : the second entry of the third parameter.
— `0x0000000000000000000000000000000000000000000000000000000000000003` : the third entry of the third parameter.

In total :

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000
```

3.20.9 Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way :

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are :

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"` :

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case) : 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity) :

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Let us apply the same principle to encode the data for a function with a signature `g(uint[][],string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding :

First we encode the length and data of the first embedded dynamic array `[1, 2]` of the first root array `[[1, 2], [3]]` :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first array, 2 ; the elements themselves are `1` and `2`)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (second element)

Then we encode the length and data of the second embedded dynamic array `[3]` of the first root array `[[1, 2], [3]]` :

- `0x0000000000000000000000000000000000000000000000000000000000000001` (number of elements in the second array, 1 ; the element is `3`)

— 0x0000000000000000000000000000000000000000000000000000000000000003 (first element)

Then we need to find the offsets a and b for their respective dynamic arrays [1, 2] and [3]. To calculate the offsets we can take a look at the encoded data of the first root array [[1, 2], [3]] enumerating each line in the encoding :

```
0 - a                                                                  - offset of [1,␣
↪2]
1 - b                                                                  - offset of [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1,␣
↪2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
5 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

Offset a points to the start of the content of the array [1, 2] which is line 2 (64 bytes); thus a = 0x0000000000000000000000000000000000000000000000000000000000000040.

Offset b points to the start of the content of the array [3] which is line 5 (160 bytes); thus b = 0x00000000000000000000000000000000000000000000000000000000000000a0.

Then we encode the embedded strings of the second root array :

— 0x0000000000000000000000000000000000000000000000000000000000000003 (number of characters in word "one")

— 0x6f6e650000000000000000000000000000000000000000000000000000000000 (utf8 representation of word "one")

— 0x0000000000000000000000000000000000000000000000000000000000000003 (number of characters in word "two")

— 0x74776f0000000000000000000000000000000000000000000000000000000000 (utf8 representation of word "two")

— 0x0000000000000000000000000000000000000000000000000000000000000005 (number of characters in word "three")

— 0x7468726565000000000000000000000000000000000000000000000000000000 (utf8 representation of word "three")

In parallel to the first root array, since strings are dynamic elements we need to find their offsets c, d and e :

```
0 - c                                                                  - offset for "one
↪"
1 - d                                                                  - offset for "two
↪"
2 - e                                                                  - offset for
↪"three"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of
↪"one"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of
↪"two"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - count for
↪"three"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset c points to the start of the content of the string "one" which is line 3 (96 bytes); thus c = 0x0000000000000000000000000000000000000000000000000000000000000060.

Offset `d` points to the start of the content of the string `"two"` which is line 5 (160 bytes); thus `d = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Offset `e` points to the start of the content of the string `"three"` which is line 7 (224 bytes); thus `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`.

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[],uint[][])`.

Then we encode the length of the first root array :

— `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first root array, 2; the elements themselves are `[1, 2]` and `[3]`)

Then we encode the length of the second root array :

— `0x0000000000000000000000000000000000000000000000000000000000000003` (number of strings in the second root array, 3; the strings themselves are `"one"`, `"two"` and `"three"`)

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order :

```
0x2289b18c                                                          - function␣
↪signature
 0 - f                                                              - offset of [[1,
↪ 2], [3]]
 1 - g                                                              - offset of [
↪"one", "two", "three"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [[1,
↪ 2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - offset of [1,␣
↪2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset of [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1,␣
↪2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
10 - 0000000000000000000000000000000000000000000000000000000000000003 - count for [
↪"one", "two", "three"]
11 - 0000000000000000000000000000000000000000000000000000000000000060 - offset for
↪"one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset for
↪"two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - offset for
↪"three"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one
↪"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of
↪"one"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two
↪"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of
↪"two"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - count for
↪"three"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `f` points to the start of the content of the array `[[1, 2], [3]]` which is line 2 (64 bytes); thus `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `g` points to the start of the content of the array `["one", "two", "three"]` which is line 10 (320 bytes); thus `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

3.20.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series : those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as :

- `address` : the address of the contract (intrinsically provided by Ethereum);
- `topics[0]` : `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated;
- `topics[n]` : `abi_encode(EVENT_INDEXED_ARGS[n - 1])` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data` : ABI encoding of `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_encode` is the ABI encoding function used for returning a series of typed values from a function, as described above).

For all types of length at most 32 bytes, the `EVENT_INDEXED_ARGS` array contains the value directly, padded or sign-extended (for signed integers) to 32 bytes, just as for regular ABI encoding. However, for all « complex » types or types of dynamic length, including all arrays, `string`, `bytes` and structs, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of a special in-place encoded value (see *Encoding of Indexed Event Parameters*), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

3.20.11 JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields :

- `type` : `"function"`, `"constructor"`, `"receive"` (the *« receive Ether » function*) or `"fallback"` (the *« default » function*);
- `name` : the name of the function;
- `inputs` : an array of objects, each of which contains :
  - `name` : the name of the parameter.
  - `type` : the canonical type of the parameter (more below).
  - `components` : used for tuple types (more below).
- `outputs` : an array of objects similar to `inputs`.
- `stateMutability` : a string with one of the following values : `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether - the default) and `payable` (function accepts Ether).

Constructor and fallback function never have `name` or `outputs`. Fallback function doesn't have `inputs` either.

Note : Sending non-zero Ether to non-payable function will revert the transaction.

Note : The state mutability `nonpayable` is reflected in Solidity by not specifying a state mutability modifier at all.

An event description is a JSON object with fairly similar fields :

- `type` : always `"event"`
- `name` : the name of the event.
- `inputs` : an array of objects, each of which contains :
  - `name` : the name of the parameter.
  - `type` : the canonical type of the parameter (more below).
  - `components` : used for tuple types (more below).
  - `indexed` : `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous` : `true` if the event was declared as `anonymous`.

For example,

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;


contract Test {
    constructor() public { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

would result in the JSON :

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Handling tuple types

Despite that names are intentionally not part of the ABI encoding they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way :

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the

word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.19 <0.7.0;
pragma experimental ABIEncoderV2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

would result in the JSON :

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

3.20.12 Strict Encoding Mode

Strict encoding mode is the mode that leads to exactly the same encoding as defined in the formal specification above. This means offsets have to be as small as possible while still not creating overlaps in the data areas and thus no gaps are allowed.

Usually, ABI decoders are written in a straightforward way just following offset pointers, but some decoders might enforce strict mode. The Solidity ABI decoder currently does not enforce strict mode, but the encoder always creates data in strict mode.

3.20.13 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where :

- — types shorter than 32 bytes are neither zero padded nor sign extended and
- — dynamic types are encoded in-place and without the length.
- — array elements are padded, but still encoded in-place

Furthermore, structs as well as nested arrays are not supported.

As an example, the encoding of `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` results in :

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                  int16(-1)
      ^^                                bytes1(0x42)
        ^^^^                            uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^  string("Hello, world!") without a length field
```

More specifically :

- — During the encoding, everything is encoded in-place. This means that there is no distinction between head and tail, as in the ABI encoding, and the length of an array is not encoded.
- — The direct arguments of `abi.encodePacked` are encoded without padding, as long as they are not arrays (or `string` or `bytes`).
- — The encoding of an array is the concatenation of the encoding of its elements **with** padding.
- — Dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field.
- — The encoding of `string` or `bytes` does not apply padding at the end unless it is part of an array or struct (then it is padded to a multiple of 32 bytes).

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used : `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

Avertissement : If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. If you use `abi.encodePacked` for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, `abi.encode` should be preferred.

3.20.14 Encoding of Indexed Event Parameters

Indexed event parameters that are not value types, i.e. arrays and structs are not stored directly but instead a keccak256-hash of an encoding is stored. This encoding is defined as follows :

- the encoding of a `bytes` and `string` value is just the string contents without any padding or length prefix.
- the encoding of a struct is the concatenation of the encoding of its members, always padded to a multiple of 32 bytes (even `bytes` and `string`).
- the encoding of an array (both dynamically- and statically-sized) is the concatenation of the encoding of its elements, always padded to a multiple of 32 bytes (even `bytes` and `string`) and without any length prefix

In the above, as usual, a negative number is padded by sign extension and not zero padded. `bytesNN` types are padded on the right while `uintNN` / `intNN` are padded on the left.

Avertissement : The encoding of a struct is ambiguous if it contains more than one dynamically-sized array. Because of that, always re-check the event data and do not rely on the search result based on the indexed parameters alone.

3.21 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

Note : Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices. See the *Interoperability With Older Contracts* section below.

3.21.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.

— The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
— The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
— Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
— The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
— The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
— Forward all available gas with external function calls starting from Tangerine Whistle.

3.21.2 Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

— The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)")), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
— Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.
— Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

3.21.3 Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

— Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
— Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = m_x` to `uint[] storage x = m_x`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
— Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example : if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
— Explicit conversions between unrelated contract types are now disallowed. You can only convert from a contract type to one of its base or ancestor types. If you are sure that a contract is compatible with the contract type you want to convert to, although it does not inherit from it, you can work around this by converting to `address` first. Example : if `A` and `B` are contract types, `B` does not inherit from `A` and `b` is a contract of type `B`, you can

still convert `b` to type `A` using `A(address(b))`. Note that you still need to watch out for matching payable fallback functions, as explained below.
— The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the *withdraw pattern*, you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
— Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes) by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`.
— Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
— For clarity reasons, the command line interface now requires `-` if the standard input is used as source.

3.21.4 Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

Command Line and JSON Interfaces

— The command line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
— The command line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language `Julia` to `Yul`.
— The `--clone-bin` and `--combined-json clone-bin` command line options were removed.
— Remappings with empty prefix are disallowed.
— The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
— The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value `"constructor"`, `"fallback"` or `"function"`.
— In unlinked binary hex files, library address placeholders are now the first 36 hex characters of the keccak256 hash of the fully qualified library name, surrounded by `$...$`. Previously, just the fully qualified library name was used. This reduces the chances of collisions, especially when long paths are used. Binary files now also contain a list of mappings from these placeholders to the fully qualified names.

Constructors

— Constructors must now be defined using the `constructor` keyword.
— Calling base constructors without parentheses is now disallowed.
— Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
— Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

Functions

— Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
— `suicide` is now disallowed (in favor of `selfdestruct`).
— `sha3` is now disallowed (in favor of `keccak256`).
— `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

Conversions

— Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
— Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

Literals and Suffixes

— The unit denomination `years` is now disallowed due to complications and confusions about leap years.
— Trailing dots that are not followed by a number are now disallowed.
— Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
— The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

Variables

— Declaring empty structs is now disallowed for clarity.
— The `var` keyword is now disallowed to favor explicitness.
— Assignments between tuples with different number of components is now disallowed.
— Values for constants that are not compile-time constants are disallowed.
— Multi-variable declarations with mismatching number of values are now disallowed.
— Uninitialized storage variables are now disallowed.
— Empty tuple components are now disallowed.
— Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
— Fixed-size arrays with a length of zero are now disallowed.

Syntax

— Using `constant` as function state mutability modifier is now disallowed.
— Boolean expressions cannot use arithmetic operations.
— The unary + operator is now disallowed.
— Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
— Empty return statements for functions with one or more return values are now disallowed.
— The « loose assembly » syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.
— Functions without implementation cannot use modifiers anymore.
— Function types with named return values are now disallowed.
— Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
— New keywords : `calldata` and `constructor`.
— New reserved keywords : `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

3.21.5 Interoperability With Older Contracts

It is still possible to interface with contracts written for Solidity versions prior to v0.5.0 (or the other way around) by defining interfaces for them. Consider you have the following pre-0.5.0 contract already deployed :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will report a warning until version 0.4.25 of the compiler
// This will not compile after 0.5.0
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

This will no longer compile with Solidity v0.5.0. However, you can define a compatible interface for it :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Note that we did not declare `anotherOldFunction` to be `view`, despite it being declared `constant` in the original contract. This is due to the fact that starting with Solidity v0.5.0 `staticcall` is used to call `view` functions. Prior to v0.5.0 the `constant` keyword was not enforced, so calling a function declared `constant` with `staticcall` may still revert, since the `constant` function may still attempt to modify storage. Consequently, when defining an interface for older contracts, you should only use `view` in place of `constant` in case you are absolutely sure that the function will work with `staticcall`.

Given the interface defined above, you can now easily use the already deployed pre-0.5.0 contract :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Similarly, pre-0.5.0 libraries can be used by defining the functions of the library without implementation and supplying the address of the pre-0.5.0 library during linking (see *Using the Commandline Compiler* for how to use the commandline compiler for linking) :

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.5.99;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.21.6 Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will not compile after 0.5.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }

    // Function visibility not provided, not an error.
    // Function mutability not provided, not an error.
    function f(uint x) returns (bytes) {
        // Var is fine in this version.
        var z = someInteger();
        x += z;
        // Throw is fine in this version.
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' causes an infinite loop.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        } while (x < 11);
        // Call returns only a Bool.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Local variables could be declared after their use.
            int y;
        }
        return b;
    }

    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

New version :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.5.99;
// This will not compile after 0.6.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using address_make_payable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract,␣
↪address unknownContract) public payable {
        // 'otherContract.transfer' is not provided.
        // Since the code of 'OtherContract' is known and has the fallback
        // function, address(otherContract) has type 'address payable'.
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' is not provided.
        // 'address(unknownContract).transfer' is not provided
        // since 'address(unknownContract)' is not 'address payable'.
        // If the function takes an 'address' which you want to send
        // funds to, you can convert it to 'address payable' via 'uint160'.
        // Note: This is not recommended and the explicit type
        // 'address payable' should be used whenever possible.
        // To increase clarity, we suggest the use of a library for
        // the conversion (provided after the contract in this example).
        address payable addr = unknownContract.make_payable();
        require(addr.send(1 ether));

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the conversion is not allowed.
        // We need to convert to a common size first:
        bytes4 x4 = bytes4(x); // Padding happens on the right
        uint32 y = uint32(x4); // Conversion is consistent
        // 'msg.value' cannot be used in a 'non-payable' function.
        // We need to make the function payable
        myNumber += y + msg.value;
    }
}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library address_make_payable {
    function make_payable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.22 Solidity v0.6.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.6.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

3.22.1 Changes the Compiler Might not Warn About

This section lists changes where the behaviour of your code might change without the compiler telling you about it.

— The resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmentric operations. Additionally, signed types are allowed for the base of the exponetation.

3.22.2 Explicitness Requirements

This section lists changes where the code now needs to be more explicit, but the semantics do not change. For most of the topics the compiler will provide suggestions.

— Functions can now only be overridden when they are either marked with the `virtual` keyword or defined in an interface. Functions without implementation outside an interface have to be marked `virtual`. When overriding a function or modifier, the new keyword `override` must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so : `override(Base1, Base2)`.

— Member-access to `length` of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays assigning a new value to their length. Use `push()`, `push(value)` or `pop()` instead, or assign a full array, which will of course overwrite existing content. The reason behind this is to prevent storage collisions by gigantic storage arrays.

— The new keyword `abstract` can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions.

— Libraries have to implement all their functions, not only the internal ones.

— The names of variables declared in inline assembly may no longer end in `_slot` or `_offset`.

— Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.

— State variable shadowing is now disallowed. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

3.22.3 Semantic and Syntactic Changes

This section lists changes where you have to modify your code and it does something else afterwards.

— Conversions from external function types to `address` are now disallowed. Instead external function types have a member called `address`, similar to the existing `selector` member.

— The function `push(value)` for dynamic storage arrays does not return the new length anymore (it returns nothing).

— The unnamed function commonly referred to as « fallback function » was split up into a new fallback function that is defined using the `fallback` keyword and a receive ether function defined using the `receive` keyword.

  — If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). This function is implicitly `payable`.

  — The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty call data). You can make this function `payable` or not. If it is not `payable` then transactions not matching any other function which send value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

3.22.4 New Features

This section lists things that were not possible prior to Solidity 0.6.0 or at least were more difficult to achieve prior to Solidity 0.6.0.

- The *try/catch statement* allows you to react on failed external calls.
- `struct` and `enum` types can be declared at file level.
- Array slices can be used for calldata arrays, for example `abi.decode(msg.data[4:], (uint, uint))` is a low-level way to decode the function call payload.
- Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as `@param`.
- Yul and Inline Assembly have a new statement called `leave` that exits the current function.
- Conversions from `address` to `address payable` are now possible via `payable(x)`, where `x` must be of type `address`.

3.22.5 Interface Changes

This section lists changes that are unrelated to the language itself, but that have an effect on the interfaces of the compiler. These may change the way how you use the compiler on the command line, how you use its programmable interface or how you analyze the output produced by it.

New Error Reporter

A new error reporter was introduced, which aims at producing more accessible error messages on the command line. It is enabled by default, but passing `--old-reporter` falls back to the the deprecated old error reporter.

Metadata Hash Options

The compiler now appends the IPFS hash of the metadata file to the end of the bytecode by default (for details, see documentation on *contract metadata*). Before 0.6.0, the compiler appended the Swarm hash by default, and in order to still support this behaviour, the new command line option `--metadata-hash` was introduced. It allows you to select the hash to be produced and appended, by passing either `ipfs` or `swarm` as value to the `--metadata-hash` command line option. Passing the value `none` completely removes the hash.

These changes can also be used via the *Standard JSON Interface* and effect the metadata JSON generated by the compiler.

The recommended way to read the metadata is to read the last two bytes to determine the length of the CBOR encoding and perform a proper decoding on that data block as explained in the *metadata section*.

Yul Optimizer

Together with the legacy bytecode optimizer, the *Yul* optimizer is now enabled by default when you call the compiler with `--optimize`. It can be disabled by calling the compiler with `--no-optimize-yul`. This mostly affects code that uses ABIEncoderV2.

C API Changes

The client code that uses the C API of `libsolc` is now in control of the memory used by the compiler. To make this change consistent, `solidity_free` was renamed to `solidity_reset`, the functions `solidity_alloc` and `solidity_free` were added and `solidity_compile` now returns a string that must be explicitly freed via `solidity_free()`.

3.22.6 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

— Change `address(f)` to `f.address` for `f` being of external function type.
— Replace `function () external [payable] { ... }` by either `receive() external payable { ... }`, `fallback() external [payable] { ... }` or both. Prefer using a `receive` function only, whenever possible.
— Change `uint length = array.push(value)` to `array.push(value);`. The new length can be accessed via `array.length`.
— Change `array.length++` to `array.push()` to increase, and use `pop()` to decrease the length of a storage array.
— For every named return parameter in a function's `@dev` documentation define a `@return` entry which contains the parameter's name as the first word. E.g. if you have function `f()` defined like `function f() public returns (uint value)` and a `@dev` annotating it, document its return parameters like so : `@return value The return value.`. You can mix named and un-named return parameters documentation so long as the notices are in the order they appear in the tuple return type.
— Choose unique identifiers for variable declarations in inline assembly that do not conflict with declartions outside the inline assembly block.
— Add `virtual` to every non-interface function you intend to override. Add `virtual` to all functions without implementation outside interfaces. For single inheritance, add `override` to every overriding function. For multiple inheritance, add `override(A, B, ..)`, where you list all contracts that define the overridden function in the parentheses. When multiple bases define the same function, the inheriting contract must override all conflicting functions.

3.23 NatSpec Format

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler. Also detailed below is output of the Solidity compiler, which extracts these comments into a machine-readable format.

3.23.1 Documentation Example

Documentation is inserted above each `class`, `interface` and `function` using the doxygen notation format.

— For Solidity you may choose `///` for single or multi-line comments, or `/**` and ending with `*/`.
— For Vyper, use `"""` indented to the inner contents with bare comments. See Vyper documentation.

The following example shows a contract and a function using all available tags.

Note : NatSpec currently does NOT apply to public state variables (see solidity#3418), even if they are declared public and therefore do affect the ABI.

The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
contract Tree {
    /// @author Mary A. Botanist
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return age in years, rounded up for partial years
    function age(uint256 rings) external pure returns (uint256) {
        return rings + 1;
    }
}
```

3.23.2 Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used. As a special case, if no tags are used then the Solidity compiler will interpret a `///` or `/**` comment in the same way as if it were tagged with `@notice`.

Tag		Context
`@title`	A title that should describe the contract/interface	contract, interface
`@author`	The name of the author	contract, interface, function
`@notice`	Explain to an end user what this does	contract, interface, function
`@dev`	Explain to a developer any extra details	contract, interface, function
`@param`	Documents a parameter just like in doxygen (must be followed by parameter name)	function
`@return`	Documents the return variables of a contract's function	function

If your function returns multiple values, like `(int quotient, int remainder)` then use multiple `@return` statements in the same format as the `@param` statements.

Dynamic expressions

The Solidity compiler will pass through NatSpec documentation from your Solidity source code to the JSON output as described in this guide. The consumer of this JSON output, for example the end-user client software, may present this to the end-user directly or it may apply some pre-processing.

For example, some client software will render :

```
/// @notice This function will multiply `a` by 7
```

to the end-user as :

```
This function will multiply 10 by 7
```

if a function is being called and the input `a` is assigned a value of 10.

Specifying these dynamic expressions is outside the scope of the Solidity documentation and you may read more at the radspec project.

Inheritance Notes

Currently it is undefined whether a contract with a function having no NatSpec will inherit the NatSpec of a parent contract/interface for that same function.

3.23.3 Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON files. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `ex1.sol` then you can generate the documentation using :

```
solc --userdoc --devdoc ex1.sol
```

And the output is below.

User Documentation

The above documentation will produce the following user documentation JSON file as output :

```
{
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function's canonical signature as defined in the Contract ABI and not simply the function's name.

Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this :

```
{
  "author" : "Larry A. Gardner",
  "details" : "All function calls are currently implemented without side effects",
  "methods" :
  {
    "age(uint256)" :
    {
      "author" : "Mary A. Botanist",
```

(suite sur la page suivante)

(suite de la page précédente)

```
      "details" : "The Alexandr N. Tetearing algorithm could increase precision",
      "params" :
      {
        "rings" : "The number of rings from dendrochronological sample"
      },
      "return" : "age in years, rounded up for partial years"
    }
  },
  "title" : "A simulator for trees"
}
```

3.24 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 malicious 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. A list of some publicly known security-relevant bugs of the compiler can be found in the *list of known bugs*, which is also machine-readable. Note that there is a bug bounty program that covers the code generator of the Solidity compiler.

As always, with open source documentation, please help us extend this section (especially, some examples would not hurt) !

NOTE : In addition to the list below, you can find more security recommendations and best practices in Guy Lando's knowledge list and the Consensys GitHub repo.

3.24.1 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) :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

// 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() public {
        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 can always include 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. In particular, the following contract will allow an attacker to refund multiple times as it uses `call` which forwards all remaining gas by default :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.7.0;

// 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() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

To avoid re-entrancy, you can use the Checks-Effects-Interactions pattern as outlined further below :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.11 <0.7.0;

contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

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 `view` 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), either the *receive Ether* or the *fallback* function is executed. If it does not have a receive nor a fallback function, the Ether will be rejected (by throwing an exception). During the execution of one of these functions, the contract can only rely on the « gas stipend » it is passed (2300 gas) being available to it at that time. This stipend is not enough to modify storage (do not take this for granted though, the stipend might change with future hard forks). To be sure that your contract can receive Ether in that way, check the gas requirements of the receive and fallback functions (for example in the « details » section in Remix).
- There is a way to forward more gas to the receiving contract using `addr.call{value: x}("")`. This is essentially the same as `addr.transfer(x)`, only that it forwards all remaining gas and opens up the ability for the recipient to perform more expensive actions (and it returns a failure code instead of automatically propagating the error). 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.
- Use the most precise units to represent the wei amount as possible, as you lose any that is rounded due to a lack of precision.
- If you want to send Ether using `address.transfer`, there are certain details to be aware of :
  1. If the recipient is a contract, it causes its receive or 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 ; take this possibility into account or use `send` and make sure to always check its return value. 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 *require*, *assert*, *revert* or because the operation is too expensive) - it « runs out of gas » (OOG). If you use `transfer` or `send` with a return value check, 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()`, `.delegatecall()` and `.staticcall()` behave in the same way.

tx.origin

Never use tx.origin for authorization. Let's say you have a wallet contract like this :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Now someone tricks you into sending Ether to the address of this attack wallet :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(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.

Two's Complement / Underflows / Overflows

As in many programming languages, Solidity's integer types are not actually integers. They resemble integers when the values are small, but behave differently if the numbers are larger. For example, the following is true : `uint8(255) + uint8(1) == 0`. This situation is called an *overflow*. It occurs when an operation is performed that requires a fixed size variable to store a number (or piece of data) that is outside the range of the variable's data type. An *underflow* is the converse situation : `uint8(0) - uint8(1) == 255`.

In general, read about the limits of two's complement representation, which even has some more special edge cases for signed numbers.

Try to use `require` to limit the size of inputs to a reasonable range and use the *SMT checker* to find potential overflows, or use a library like SafeMath if you want all overflows to cause a revert.

Code such as `require((balanceOf[_to] + _value) >= balanceOf[_to])` can also help you check if values are what you expect.

Clearing Mappings

The Solidity type `mapping` (see *Mappages*) is a storage-only key-value data structure that does not keep track of the keys that were assigned a non-zero value. Because of that, cleaning a mapping without extra information about the written keys is not possible. If a `mapping` is used as the base type of a dynamic storage array, deleting or popping the array will have no effect over the `mapping` elements. The same happens, for example, if a `mapping` is used as the type of a member field of a `struct` that is the base type of a dynamic storage array. The `mapping` is also ignored in assignments of structs or arrays containing a `mapping`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

contract Map {
    mapping (uint => uint)[] array;

    function allocate(uint _newMaps) public {
        for (uint i = 0; i < _newMaps; i++)
            array.push();
    }

    function writeMap(uint _map, uint _key, uint _value) public {
        array[_map][_key] = _value;
    }

    function readMap(uint _map, uint _key) public view returns (uint) {
        return array[_map][_key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

Consider the example above and the following sequence of calls : `allocate(10)`, `writeMap(4, 128, 256)`. At this point, calling `readMap(4, 128)` returns 256. If we call `eraseMaps`, the length of state variable `array` is zeroed, but since its `mapping` elements cannot be zeroed, their information stays alive in the contract's storage. After deleting `array`, calling `allocate(5)` allows us to access `array[4]` again, and calling `readMap(4, 128)` returns 256 even without another call to `writeMap`.

If your `mapping` information must be deleted, consider using a library similar to iterable mapping, allowing you to traverse the keys and delete their values in the appropriate `mapping`.

Minor Details

— 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.

3.24.2 Recommendations

Take Warnings Seriously

If the compiler warns you about something, you should better change it. Even if you do not think that this particular warning has security implications, there might be another issue buried beneath it. Any compiler warning we issue can be silenced by slight changes to the code.

Always use the latest version of the compiler to be notified about all recently introduced warnings.

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.

Ask for Peer Review

The more people examine a piece of code, the more issues are found. Asking people to review your code also helps as a cross-check to find out whether your code is easy to understand - a very important criterion for good smart contracts.

3.24.3 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.

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.

Solidity implements a formal verification approach based on SMT solving. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require/assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true. If an assertion failure is found, a counterexample is given to the user, showing how the assertion can be violated.

The SMTChecker also checks automatically for arithmetic underflow/overflow, trivial conditions and unreachable code. It is currently an experimental feature, therefore in order to use it you need to enable it via *a pragma directive*.

The SMTChecker traverses the Solidity AST creating and collecting program constraints. When it encounters a verification target, an SMT solver is invoked to determine the outcome. If a check fails, the SMTChecker provides specific input values that lead to the failure.

While the SMTChecker encodes Solidity code into SMT constraints, it contains two reasoning engines that use that encoding in different ways.

SMT Encoding

The SMT encoding tries to be as precise as possible, mapping Solidity types and expressions to their closest SMT-LIB representation, as shown in the table below.

Solidity type	SMT sort	Theories (quantifier-free)
Boolean	Bool	Bool
intN, uintN, address, bytesN, enum	Integer	LIA, NIA
array, mapping, bytes, string	Array	Arrays
other types	Integer	LIA

Types that are not yet supported are abstracted by a single 256-bit unsigned integer, where their unsupported operations are ignored.

For more details on how the SMT encoding works internally, see the paper SMT-based Verification of Solidity Smart Contracts.

Model Checking Engines

The SMTChecker module implements two different reasoning engines that use the SMT encoding above, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics.

Bounded Model Checker (BMC)

The BMC engine analyzes functions in isolation, that is, it does not take the overall behavior of the contract throughout many transactions into account when analyzing each function. Loops are also ignored in this engine at the moment. Internal function calls are inlined as long as they are not recursive, direct or indirectly. External function calls are inlined if possible, and knowledge that is potentially affected by reentrancy is erased.

The characteristics above make BMC easily prone to reporting false positives, but it is also lightweight and should be able to quickly find small local bugs.

Constrained Horn Clauses (CHC)

The Solidity contract's Control Flow Graph (CFG) is modelled as a system of Horn clauses, where the lifecycle of the contract is represented by a loop that can visit every public/external function non-deterministically. This way, the behavior of the entire contract over an unbounded number of transactions is taken into account when analyzing any function. Loops are fully supported by this engine. Internal function calls are supported, but external function calls are currently unsupported.

The CHC engine is much more powerful than BMC in terms of what it can prove, and might require more computing resources.

Abstraction and False Positives

The SMTChecker implements abstractions in an incomplete and sound way : If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type). If it determines that a verification target is safe, it is indeed safe, that is, there are no false negatives (unless there is a bug in the SMTChecker).

In the BMC engine, function calls to the same contract (or base contracts) are inlined when possible, that is, when their implementation is available. Calls to functions in other contracts are not inlined even if their code is available, since we cannot guarantee that the actual deployed code is the same.

The CHC engine creates nonlinear Horn clauses that use summaries of the called functions to support internal function calls. The same approach can and will be used for external function calls, but the latter requires more work regarding the entire state of the blockchain and is still unimplemented.

Complex pure functions are abstracted by an uninterpreted function (UF) over the arguments.

Functions	SMT behavior
`assert`	Verification target
`require`	Assumption
internal	BMC : Inline function call CHC : Function summaries
external	BMC : Inline function call or erase knowledge about state variables and local storage references. CHC : Function summaries and erase state knowledge.
`gasleft`, `blockhash`, `keccak256`, `ecrecover` `ripemd160`, `addmod`, `mulmod`	Abstracted with UF
pure functions without implementation (external or complex)	Abstracted with UF
external functions without implementation	BMC : Unsupported CHC : Nondeterministic summary
others	Currently unsupported

Using abstraction means loss of precise knowledge, but in many cases it does not mean loss of proving power.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0;
pragma experimental SMTChecker;
// This may report a warning if no SMT solver available.
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract Recover
{
    function f(
        bytes32 hash,
        uint8 _v1, uint8 _v2,
        bytes32 _r1, bytes32 _r2,
        bytes32 _s1, bytes32 _s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, _v1, _r1, _s1);
        require(_v1 == _v2);
        require(_r1 == _r2);
        require(_s1 == _s2);
        address a2 = ecrecover(hash, _v2, _r2, _s2);
        assert(a1 == a2);
        return a1;
    }
}
```

In the example above, the SMTChecker is not expressive enough to actually compute `ecrecover`, but by modelling the function calls as uninterpreted functions we know that the return value is the same when called on equivalent parameters. This is enough to prove that the assertion above is always true.

Abstracting a function call with an UF can be done for functions known to be deterministic, and can be easily done for pure functions. It is however difficult to do this with general external functions, since they might depend on state variables.

External function calls also imply that any current knowledge that the SMTChecker might have regarding mutable state variables needs to be erased to guarantee no false negatives, since the called external function might direct or indirectly call a function in the analyzed contract that changes state variables.

Reference Types and Aliasing

Solidity implements aliasing for reference types with the same *data location*. That means one variable may be modified through a reference to the same data area. The SMTChecker does not keep track of which references refer to the same data. This implies that whenever a local reference or state variable of reference type is assigned, all knowledge regarding variables of the same type and data location is erased. If the type is nested, the knowledge removal also includes all the prefix base types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0;
pragma experimental SMTChecker;
// This will report a warning

contract Aliasing
{
    uint[] array;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal view {
        require(array[0] == 42);
        require(a[0] == 2);
        require(c[0][0] == 2);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        require(d[0] == 2);
        b[0] = 1;
        // Erasing knowledge about memory references should not
        // erase knowledge about state variables.
        assert(array[0] == 42);
        // Fails because `a == b` is possible.
        assert(a[0] == 2);
        // Fails because `c[i] == b` is possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
}
```

After the assignment to `b[0]`, we need to clear knowledge about `a` since it has the same type (`uint[]`) and data location (memory). We also need to clear knowledge about `c`, since its base type is also a `uint[]` located in memory. This implies that some `c[i]` could refer to the same data as `b` or `a`.

Notice that we do not clear knowledge about `array` and `d` because they are located in storage, even though they also have type `uint[]`. However, if `d` was assigned, we would need to clear knowledge about `array` and vice-versa.

3.25 Resources

3.25.1 General

- Ethereum
- Changelog
- Source Code
- Ethereum Stackexchange
- Language Users Chat
- Compiler Developers Chat

3.25.2 Solidity Integrations

- Generic :
  - **EthFiddle** Solidity IDE in the Browser. Write and share your Solidity code. Uses server-side components.
  - **Remix** Browser-based IDE with integrated compiler and Solidity runtime environment without server-side components.
  - **Solhint** Solidity linter that provides security, style guide and best practice rules for smart contract validation.
  - **Solidity IDE** Browser-based IDE with integrated compiler, Ganache and local file system support.
  - **Ethlint** Linter to identify and fix style and security issues in Solidity.
  - **Superblocks Lab** Browser-based IDE. Built-in browser-based VM and Metamask integration (one click deployment to Testnet/Mainnet).
- Atom :
  - **Etheratom** Plugin for the Atom editor that features syntax highlighting, compilation and a runtime environment (Backend node & VM compatible).
  - **Atom Solidity Linter** Plugin for the Atom editor that provides Solidity linting.
  - **Atom Solium Linter** Configurable Solidity linter for Atom using Solium (now Ethlint) as a base.
- Eclipse :

  - **YAKINDU Solidity Tools** Eclipse based IDE. Features context sensitive code completion and help, code navigation, syntax coloring, built in compiler, quick fixes and templates.
- Emacs :
  - **Emacs Solidity** Plugin for the Emacs editor providing syntax highlighting and compilation error reporting.
- IntelliJ :
  - **IntelliJ IDEA plugin** Solidity plugin for IntelliJ IDEA (and all other JetBrains IDEs)
- Sublime :
  - **Package for SublimeText - Solidity language syntax** Solidity syntax highlighting for SublimeText editor.
- Vim :
  - **Vim Solidity** Plugin for the Vim editor providing syntax highlighting.
  - **Vim Syntastic** Plugin for the Vim editor providing compile checking.
- Visual Studio Code :
  - **Visual Studio Code extension** Solidity plugin for Microsoft Visual Studio Code that includes syntax highlighting and the Solidity compiler.

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

3.25.3 Solidity Tools

- **ABI to Solidity interface converter** A script for generating contract interfaces from the ABI of a smart contract.
- **Dapp** Build tool, package manager, and deployment assistant for Solidity.
- **Doxity** Documentation Generator for Solidity.
- **evmdis** EVM Disassembler that performs static analysis on the bytecode to provide a higher level of abstraction than raw EVM operations.
- **EVM Lab** Rich tool package to interact with the EVM. Includes a VM, Etherchain API, and a trace-viewer with gas cost display.
- **leafleth** A documentation generator for Solidity smart-contracts.
- **PIET** A tool to develop, audit and use Solidity smart contracts through a simple graphical interface.
- **solc-select** A script to quickly switch between Solidity compiler versions.
- **Solidity prettier plugin** A Prettier Plugin for Solidity.
- **Solidity REPL** Try Solidity instantly with a command-line Solidity console.
- **solgraph** Visualize Solidity control flow and highlight potential security vulnerabilities.
- **Securify** Fully automated online static analyzer for smart contracts, providing a security report based on vulnerability patterns.
- **Sūrya** Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts" structure. Also supports querying the function call graph.
- **Universal Mutator** A tool for mutation generation, with configurable rules and support for Solidity and Vyper.

3.25.4 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

3.26 Using the compiler

3.26.1 Using the Commandline Compiler

Note : This section does not apply to *solcjs*, not even if it is used in commandline mode.

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. 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-json --asm sourceFile.sol`.

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime (more specifically, it assumes each opcode is executed around 200 times). If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number. This parameter has effects on the following (this might change in the future) :

— the size of the binary search in the function dispatch routine
— the way constants like large numbers or strings are stored

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide path redirects using `prefix=path` in the following way :

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

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`. `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.

An empty remapping prefix is not allowed.

If there are multiple matches due to remappings, the one with the longest common prefix is selected.

For security reasons the compiler has restrictions what directories it can access. Paths (and their subdirectories) of source files specified on the commandline and paths defined by remappings are allowed for import statements, but everything else is rejected. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch.

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__$53aea86b7d70b31448b230b20ae141a537$__`. These are placeholders for the actual library addresses. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name. The bytecode file will also contain lines of the form `// <placeholder> -> <fq library name>` at the end to help identify which libraries the placeholders represent. Note that the fully qualified library name is the path of its source file and the library name separated by `:`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points :

Either add `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD5678901234567890123456789012345678890"` 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 `__$53aea86b7d70b31448b230b20ae141a537$__`-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.

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses. The process will always terminate in a « success » state and report any errors via the JSON output.

Note : The library placeholder used to be the fully qualified name of the library itself instead of the hash of it. This format is still supported by `solc --link` but the compiler will no longer output it. This change was made to reduce the likelihood of a collision between libraries, since only the first 36 characters of the fully qualified library name could be used.

3.26.2 Setting the EVM version to target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

Avertissement : Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command line, you can select the EVM version as follows :

```
solc --evm-version <VERSION> contract.sol
```

In the *standard JSON interface*, use the `"evmVersion"` key in the `"settings"` field :

```
{
  "sources": { ... },
  "settings": {
    "optimizer": { ... },
    "evmVersion": "<VERSION>"
  }
}
```

Target options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- **`homestead`**
  - (oldest version)
- **`tangerineWhistle`**
  - Gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
  - All gas sent by default for external calls, previously a certain amount had to be retained.
- **`spuriousDragon`**
  - Gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- **`byzantium`**
  - Opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
  - The `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
  - It is possible to access dynamic data returned from function calls.

        — `revert` opcode introduced, which means that `revert()` will not waste gas.
    — **`constantinople`**
        — Opcodes `create2`, ``extcodehash`, `shl`, `shr` and `sar` are available in assembly.
        — Shifting operators use shifting opcodes and thus need less gas.
    — **`petersburg`**
        — The compiler behaves the same way as with constantinople.
    — **`istanbul` (default)**
        — Opcodes `chainid` and `selfbalance` are available in assembly.
    — `berlin` (**experimental**)

3.26.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The standard error output is not used and the process will always terminate in a « success » state, even if there were errors. Errors are always reported as part of the JSON output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

Input Description

```
{
  // Required: Source code language. Currently supported are "Solidity" and "Yul".
  "language": "Solidity",
  // Required
  "sources":
  {
    // The keys here are the "global" names of the source files,
    // imports can use other files via remappings (see below).
    "myFile.sol":
    {
      // Optional: keccak256 hash of the source file
      // It is used to verify the retrieved content if imported via URLs.
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): URL(s) to the source file.
      // URL(s) should be imported in this order and the result checked against the
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL(s) result in success, an error should be raised.
      // Using the commandline interface only filesystem paths are supported.
      // With the JavaScript interface the URL will be passed to the user-supplied
      // read callback, so any URL supported by the callback can be used.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // If files are used, their directories should be added to the command line␣
↪via
        // `--allow-paths <path>`.
```

(suite sur la page suivante)

(suite de la page précédente)

```
      ]
    },
    "destructible":
    {
      // Optional: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "urls" is used): literal contents of the source file
      "content": "contract destructible is owned { function shutdown() { if (msg.
↪sender == owner) selfdestruct(owner); } }"
    }
  },
  // Optional
  "settings":
  {
    // Optional: Sorted list of remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Optimizer settings
    "optimizer": {
      // disabled by default
      "enabled": true,
      // Optimize for how many times you intend to run the code.
      // Lower values will optimize more for initial deployment cost, higher
      // values will optimize more for high-frequency usage.
      "runs": 200,
      // Switch optimizer components on or off in detail.
      // The "enabled" switch above provides two defaults which can be
      // tweaked here. If "details" is given, "enabled" can be omitted.
      "details": {
        // The peephole optimizer is always on if no details are given,
        // use details to switch it off.
        "peephole": true,
        // The unused jumpdest remover is always on if no details are given,
        // use details to switch it off.
        "jumpdestRemover": true,
        // Sometimes re-orders literals in commutative operations.
        "orderLiterals": false,
        // Removes duplicate code blocks
        "deduplicate": false,
        // Common subexpression elimination, this is the most complicated step but
        // can also provide the largest gain.
        "cse": false,
        // Optimize representation of literal numbers and strings in code.
        "constantOptimizer": false,
        // The new Yul optimizer. Mostly operates on the code of ABIEncoderV2
        // and inline assembly.
        // It is activated together with the global optimizer setting
        // and can be deactivated here.
        // Before Solidity 0.6.0 it had to be activated through this switch.
        "yul": false,
        // Tuning options for the Yul optimizer.
        "yulDetails": {
          // Improve allocation of stack slots for variables, can free up stack slots␣
↪early.
          // Activated by default if the Yul optimizer is activated.
          "stackAllocation": true,
          // Select optimization steps to be applied.
          // Optional, the optimizer will use the default sequence if omitted.
```

(suite sur la page suivante)

(suite de la page précédente)

```
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    // Version of the EVM to compile for.
    // Affects type checking and code generation. Can be homestead,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg,␣
↪istanbul or berlin
    "evmVersion": "byzantium",
    // Optional: Debugging settings
    "debug": {
      // How to treat revert (and require) reason strings. Settings are
      // "default", "strip", "debug" and "verboseDebug".
      // "default" does not inject compiler-generated revert strings and keeps user-
↪supplied ones.
      // "strip" removes all revert strings (if possible, i.e. if literals are used)␣
↪keeping side-effects
      // "debug" injects strings for compiler-generated internal reverts, implemented␣
↪for ABI encoders V1 and V2 for now.
      // "verboseDebug" even appends further information to user-supplied revert␣
↪strings (not yet implemented)
      "revertStrings": "default"
    }
    // Metadata settings (optional)
    "metadata": {
      // Use only literal content and not URLs (false by default)
      "useLiteralContent": true,
      // Use the given hash method for the metadata hash that is appended to the␣
↪bytecode.
      // The metadata hash can be removed from the bytecode via option "none".
      // The other options are "ipfs" and "bzzr1".
      // If the option is omitted, "ipfs" is used by default.
      "bytecodeHash": "ipfs"
    },
    // Addresses of the libraries. If not all libraries are given here,
    // it can result in unlinked objects whose output data is different.
    "libraries": {
      // The top level key is the the name of the source file where the library is␣
↪used.
      // If remappings are used, this source file should match the global path
      // after remappings were applied.
      // If this key is an empty string, that refers to a global level.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    }
    // The following can be used to select desired outputs based
    // on file and contract names.
    // If this field is omitted, then the compiler loads and does type checking,
    // but will not generate any outputs apart from errors.
    // The first level key is the file name and the second level key is the contract␣
↪name.
    // An empty contract name is used for outputs that are not tied to a contract
    // but to the whole source file like the AST.
    // A star as contract name refers to all contracts in the file.
    // Similarly, a star as a file name matches all files.
    // To select all outputs the compiler can possibly generate, use
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // "outputSelection: { "*": { "*": [ "*" ], "": [ "*" ] } }"
    // but note that this might slow down the compilation process needlessly.
    //
    // The available output types are as follows:
    //
    // File level (needs empty string as contract name):
    //   ast - AST of all source files
    //   legacyAST - legacy AST of all source files
    //
    // Contract level (needs the contract name or "*"):
    //   abi - ABI
    //   devdoc - Developer documentation (natspec)
    //   userdoc - User documentation (natspec)
    //   metadata - Metadata
    //   ir - Yul intermediate representation of the code before optimization
    //   irOptimized - Intermediate representation after optimization
    //   storageLayout - Slots, offsets and types of the contract's state variables.
    //   evm.assembly - New assembly format
    //   evm.legacyAssembly - Old-style assembly format in JSON
    //   evm.bytecode.object - Bytecode object
    //   evm.bytecode.opcodes - Opcodes list
    //   evm.bytecode.sourceMap - Source mapping (useful for debugging)
    //   evm.bytecode.linkReferences - Link references (if unlinked object)
    //   evm.deployedBytecode* - Deployed bytecode (has all the options that evm.
↪bytecode has)
    //   evm.deployedBytecode.immutableReferences - Map from AST ids to bytecode␣
↪ranges that reference immutables
    //   evm.methodIdentifiers - The list of function hashes
    //   evm.gasEstimates - Function gas estimates
    //   ewasm.wast - eWASM S-expressions format (not supported at the moment)
    //   ewasm.wasm - eWASM binary format (not supported at the moment)
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to␣
↪request everything.
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Enable the metadata and bytecode outputs of␣
↪every single contract.
          , "evm.bytecode.sourceMap" // Enable the source map output of every single␣
↪contract.
        ],
        "": [
          "ast" // Enable the AST output of every single file.
        ]
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    }
  }
}
```

Output Description

```
{
  // Optional: not present if no errors/warnings were encountered
  "errors": [
    {
      // Optional: Location within the source file.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // Optional: Further locations (e.g. places of conflicting declarations)
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError",
↪"Exception", etc.
      // See below for complete list of types.
      "type": "TypeError",
      // Mandatory: Component where the error originated, such as "general", "ewasm",␣
↪etc.
      "component": "general",
      // Mandatory ("error" or "warning")
      "severity": "error",
      // Mandatory
      "message": "Invalid keyword"
      // Optional: the message formatted with source location
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // This contains the file-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "sources": {
    "sourceFile.sol": {
      // Identifier of the source (used in source maps)
      "id": 1,
      // The AST object
      "ast": {},
      // The legacy AST object
      "legacyAST": {}
    }
  },
  // This contains the contract-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "contracts": {
    "sourceFile.sol": {
      // If the language used has no contract names, this field should equal to an␣
↪empty string.
      "ContractName": {
        // The Ethereum Contract ABI. If empty, it is represented as an empty array.
        // See https://solidity.readthedocs.io/en/develop/abi-spec.html
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "abi": [],
        // See the Metadata Output documentation (serialised JSON string)
        "metadata": "{...}",
        // User documentation (natspec)
        "userdoc": {},
        // Developer documentation (natspec)
        "devdoc": {},
        // Intermediate representation (string)
        "ir": "",
        // See the Storage Layout documentation.
        "storageLayout": {"storage": [...], "types": {...} },
        // EVM-related outputs
        "evm": {
          // Assembly (string)
          "assembly": "",
          // Old-style assembly (object)
          "legacyAssembly": {},
          // Bytecode and related details.
          "bytecode": {
            // The bytecode as a hex string.
            "object": "00fe",
            // Opcodes list (string)
            "opcodes": "",
            // The source mapping as a string. See the source mapping definition.
            "sourceMap": "",
            // If given, this is an unlinked object.
            "linkReferences": {
              "libraryFile.sol": {
                // Byte offsets into the bytecode.
                // Linking replaces the 20 bytes located there.
                "Library1": [
                  { "start": 0, "length": 20 },
                  { "start": 200, "length": 20 }
                ]
              }
            }
          },
          "deployedBytecode": {
            ..., // The same layout as above.
            "immutableReferences": [
              // There are two references to the immutable with AST ID 3, both 32␣
↪bytes long. One is
              // at bytecode offset 42, the other at bytecode offset 80.
              "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
            ]
          },
          // The list of function hashes
          "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
          },
          // Function gas estimates
          "gasEstimates": {
            "creation": {
              "codeDepositCost": "420000",
              "executionCost": "infinite",
              "totalCost": "infinite"
            },
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "external": {
              "delegate(address)": "25000"
            },
            "internal": {
              "heavyLifting()": "infinite"
            }
          }
        },
        // eWASM related outputs
        "ewasm": {
          // S-expressions format
          "wast": "",
          // Binary format (hex string)
          "wasm": ""
        }
      }
    }
  }
}
```

Error types

1. `JSONError` : JSON input doesn't conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
2. `IOError` : IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
3. `ParserError` : Source code doesn't conform to the language rules.
4. `DocstringParsingError` : The NatSpec tags in the comment block cannot be parsed.
5. `SyntaxError` : Syntactical error, such as `continue` is used outside of a `for` loop.
6. `DeclarationError` : Invalid, unresolvable or clashing identifier names. e.g. `Identifier not found`
7. `TypeError` : Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. `UnimplementedFeatureError` : Feature is not supported by the compiler, but is expected to be supported in future versions.
9. `InternalCompilerError` : Internal bug triggered in the compiler - this should be reported as an issue.
10. `Exception` : Unknown failure during compilation - this should be reported as an issue.
11. `CompilerError` : Invalid use of the compiler stack - this should be reported as an issue.
12. `FatalError` : Fatal error not processed correctly - this should be reported as an issue.
13. `Warning` : A warning, which didn't stop the compilation, but should be addressed if possible.

3.26.4 Compiler tools

solidity-upgrade

`solidity-upgrade` can help you to semi-automatically upgrade your contracts to breaking language changes. While it does not and cannot implement all required changes for every breaking release, it still supports the ones, that would need plenty of repetitive manual adjustments otherwise.

Note : `solidity-upgrade` carries out a large part of the work, but your contracts will most likely need further manual adjustments. We recommend using a version control system for your files. This helps reviewing and eventually rolling back the changes made.

Avertissement : `solidity-upgrade` is not considered to be complete or free from bugs, so please use with care.

How it works

You can pass (a) Solidity source file(s) to `solidity-upgrade [files]`. If these make use of `import` statement which refer to files outside the current source file's directory, you need to specify directories that are allowed to read and import files from, by passing `--allow-paths [directory]`. You can ignore missing files by passing `--ignore-missing`.

`solidity-upgrade` is based on `libsolidity` and can parse, compile and analyse your source files, and might find applicable source upgrades in them.

Source upgrades are considered to be small textual changes to your source code. They are applied to an in-memory representation of the source files given. The corresponding source file is updated by default, but you can pass `--dry-run` to simulate to whole upgrade process without writing to any file.

The upgrade process itself has two phases. In the first phase source files are parsed, and since it is not possible to upgrade source code on that level, errors are collected and can be logged by passing `--verbose`. No source upgrades available at this point.

In the second phase, all sources are compiled and all activated upgrade analysis modules are run alongside compilation. By default, all available modules are activated. Please read the documentation on *available modules* for further details.

This can result in compilation errors that may be fixed by source upgrades. If no errors occur, no source upgrades are being reported and you're done. If errors occur and some upgrade module reported a source upgrade, the first reported one gets applied and compilation is triggered again for all given source files. The previous step is repeated as long as source upgrades are reported. If errors still occur, you can log them by passing `--verbose`. If no errors occur, your contracts are up to date and can be compiled with the latest version of the compiler.

Available upgrade modules

Module	Version	Description
`constructor`	0.5.0	Constructors must now be defined using the `constructor` keyword.
`visibility`	0.5.0	Explicit function visibility is now mandatory, defaults to `public`.
`abstract`	0.6.0	The keyword `abstract` has to be used if a contract does not implement all its functions.
`virtual`	0.6.0	Functions without implementation outside an interface have to be marked `virtual`.
`override`	0.6.0	When overriding a function or modifier, the new keyword `override` must be used.

Please read *0.5.0 release notes* and *0.6.0 release notes* for further details.

Synopsis

```
Usage: solidity-upgrade [options] contract.sol

Allowed options:
    --help               Show help message and exit.
    --version            Show version and exit.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    --allow-paths path(s)
                          Allow a given path for imports. A list of paths can be
                          supplied by separating them with a comma.
    --ignore-missing      Ignore missing files.
    --modules module(s)   Only activate a specific upgrade module. A list of
                          modules can be supplied by separating them with a comma.
    --dry-run             Apply changes in-memory only and don't write to input
                          file.
    --verbose             Print logs, errors and changes. Shortens output of
                          upgrade patches.
    --unsafe              Accept *unsafe* changes.
```

Bug Reports / Feature requests

If you found a bug or if you have a feature request, please file an issue on Github.

Example

Assume you have the following contracts you want to update declared in `Source.sol` :

```
// This will not compile after 0.5.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.4.23 <0.5.0;

contract Updateable {
    function run() public view returns (bool);
    function update() public;
}

contract Upgradable {
    function run() public view returns (bool);
    function upgrade();
}

contract Source is Updateable, Upgradable {
    function Source() public {}

    function run()
        public
        view
        returns (bool) {}

    function update() {}
    function upgrade() {}
}
```

Required changes

To bring the contracts up to date with the current Solidity version, the following upgrade modules have to be executed : `constructor`, `visibility`, `abstract`, `override` and `virtual`. Please read the documentation on *available modules* for further details.

Running the upgrade

In this example, all modules needed to upgrade the contracts above, are available and all of them are activated by default. Therefore you do not need to specify the `--modules` option.

```
$ solidity-upgrade Source.sol --dry-run
```

```
Running analysis (and upgrade) on given source files.
..............

After upgrade:

Found 0 errors.
Found 0 upgrades.
```

The above performs a dry-ran upgrade on the given file and logs statistics after all. In this case, the upgrade was successful and no further adjustments are needed.

Finally, you can run the upgrade and also write to the source file.

```
$ solidity-upgrade Source.sol
```

```
Running analysis (and upgrade) on given source files.
..............

After upgrade:

Found 0 errors.
Found 0 upgrades.
```

Review changes

The command above applies all changes as shown below. Please review them carefully.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

abstract contract Updateable {
    function run() public view virtual returns (bool);
    function update() public virtual;
}

abstract contract Upgradable {
    function run() public view virtual returns (bool);
    function upgrade() public virtual;
}

contract Source is Updateable, Upgradable {
    constructor() public {}

    function run()
        public
        view
        override(Updateable,Upgradable)
        returns (bool) {}
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function update() public override {}
    function upgrade() public override {}
}
```

3.27 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can be compiled to bytecode for different backends.

Support for EVM 1.0, EVM 1.5 and eWASM is planned, and it is designed to be a usable common denominator of all three platforms. It can already be used in stand-alone mode and for « inline assembly » inside Solidity and there is an experimental implementation of the Solidity compiler that uses Yul as an intermediate language. Yul is a good target for high-level optimisation stages that can benefit all target platforms equally.

3.27.1 Motivation and High-level Description

The design of Yul tries to achieve several goals :

1. Programs written in Yul should be readable, even if the code is generated by a compiler from Solidity or another high-level language.
2. Control flow should be easy to understand to help in manual inspection, formal verification and optimization.
3. The translation from Yul to bytecode should be as straightforward as possible.
4. Yul should be suitable for whole-program optimization.

In order to achieve the first and second goal, Yul provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. These should be sufficient for adequately representing the control flow for assembly programs. Therefore, no explicit statements for `SWAP`, `DUP`, `JUMP` and `JUMPI` are provided, 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.

Even though it was designed for stack machines, Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack.

The third goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...) and cleanup of local variables from the stack.

To avoid confusions between concepts like values and references, Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability.

To keep the language simple and flexible, Yul does not have any built-in operations, functions or types in its pure form. These are added together with their semantics when specifying a dialect of Yul, which allows to specialize Yul to the requirements of different target platforms and feature sets.

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as builtin functions (see below) and defines only the type `u256`, which is the native 256-bit type of the EVM. Because of that, we will not provide types in the examples below.

3.27.2 Simple Example

The following example program is written in the EVM dialect and computes exponentiation. It can be compiled using `solc --strict-assembly`. The builtin functions `mul` and `div` compute product and division, respectively.

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

It is also possible to implement the same function using a for-loop instead of with recursion. Here, `lt(a, b)` computes whether `a` is less than `b`. less-than comparison.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

At the *end of the section*, a complete implementation of the ERC-20 standard can be found.

3.27.3 Stand-Alone Usage

You can use Yul in its stand-alone form in the EVM dialect using the Solidity compiler. This will use the *Yul object notation* so that it is possible to refer to code as data to deploy contracts. This Yul mode is available for the commandline compiler (use `--strict-assembly`) and for the *standard-json interface* :

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

Avertissement : Yul is in active development and bytecode generation is only fully implemented for the EVM dialect of Yul with EVM 1.0 as target.

3.27.4 Informal Description of Yul

In the following, we will talk about each individual aspect of the Yul language. In examples, we will use the default EVM dialect.

Syntax

Yul parses comments, literals and identifiers in the same way as Solidity, so you can e.g. use `//` and `/* */` to denote comments. There is one exception : Identifiers in Yul can contain dots : `..`

Yul can specify « objects » that consist of code, data and sub-objects. Please see *Yul Objects* below for details on that. In this section, we are only concerned with the code part of such an object. This code part always consists of a curly-braces delimited block. Most tools support specifying just a code block where an object is expected.

Inside a code block, the following elements can be used (see the later sections for more details) :

— literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)
— calls to builtin functions, e.g. `add(1, mload(0))`
— variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of 0 is assigned)
— identifiers (variables), e.g. `add(3, x)`
— assignments, e.g. `x := add(y, 3)`
— blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`
— if statements, e.g. `if lt(a, b) { sstore(0, 1) }`
— switch statements, e.g. `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
— for loops, e.g. `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
— function definitions, e.g. `function f(a, b) -> c { c := add(a, b) }`

Multiple syntactical elements can follow each other simply separated by whitespace, i.e. there is no terminating `;` or newline required.

Literals

You can use integer constants in decimal or hexadecimal notation. When compiling for the EVM, this will be translated into an appropriate `PUSHi` instruction. In the following example, `3` and `2` are added resulting in 5 and then the bitwise `and` with the string « abc » is computed. The final value is assigned to a local variable called `x`. Strings are stored left-aligned and cannot be longer than 32 bytes.

```
let x := and("abc", add(3, 2))
```

Unless it is the default type, the type of a literal has to be specified after a colon :

```
let x := and("abc":uint32, add(3:uint256, 2:uint256))
```

Function Calls

Both built-in and user-defined functions (see below) can be called in the same way as shown in the previous example. If the function returns a single value, it can be directly used inside an expression again. If it returns multiple values, they have to be assigned to local variables.

```
mstore(0x80, add(mload(0x80), 3))
// Here, the user-defined function `f` returns
// two values. The definition of the function
// is missing from the example.
let x, y := f(1, mload(0))
```

For built-in functions of the EVM, functional expressions can be directly translated to a stream of opcodes : You just read the expression from right to left to obtain the opcodes. In the case of the first line in the example, this is `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`.

For calls to user-defined functions, the arguments are also put on the stack from right to left and this is the order in which argument lists are evaluated. The return values, though, are expected on the stack from left to right, i.e. in this example, `y` is on top of the stack and `x` is below it.

Variable Declarations

You can use the `let` keyword to declare variables. A variable is only visible inside the `{...}`-block it was defined in. When compiling to the EVM, a new stack slot is created that is reserved for the variable and automatically removed again when the end of the block is reached. You can provide an initial value for the variable. If you do not provide a value, the variable will be initialized to zero.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions `mstore`, `mload`, `sstore` and `sload`. Future dialects migh introduce specific types for such pointers.

When a variable is referenced, its current value is copied. For the EVM, this translates to a `DUP` instruction.

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y is "deallocated" here
    sstore(v, zero)
} // v and zero are "deallocated" here
```

If the declared variable should have a type different from the default type, you denote that following a colon. You can also declare multiple variables in one statement when you assign from a function call that returns multiple values.

```
{
    let zero:uint32 := 0:uint32
    let v:uint256, t:uint32 := f()
    let x, y := g()
}
```

Depending on the optimiser settings, the compiler can free the stack slots already after the variable has been used for the last time, even though it is still in scope.

Assignments

Variables can be assigned to after their definition using the `:=` operator. It is possible to assign multiple variables at the same time. For this, the number and types of the values have to match. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables.

```
let v := 0
// re-assign v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assign multiple values
v, t := f()
```

If

The if statement can be used for conditionally executing code. No « else » block can be defined. Consider using « switch » instead (see below) if you need multiple alternatives.

```
if eq(value, 0) { revert(0, 0) }
```

The curly braces for the body are required.

Switch

You can use a switch statement as an extended version of the if statement. It takes the value of an expression and compares it to several literal constants. The branch corresponding to the matching constant is taken. Contrary to other programming languages, for safety reasons, control flow does not continue from one case to the next. There can be a fallback or default case called `default` which is taken if none of the literal constants matches.

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

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

Loops

Yul supports for-loops which consist of a header containing an initializing part, a condition, a post-iteration part and a body. The condition has to be an expression, while the other three are blocks. If the initializing part declares any variables at the top level, the scope of these variables extends to all other parts of the loop.

The `break` and `continue` statements can be used in the body to exit the loop or skip to the post-part, respectively.

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

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

For loops can also be used as a replacement for while loops : Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
```

(suite sur la page suivante)

(suite de la page précédente)

```
        i := add(i, 0x20)
    }
}
```

Function Declarations

Yul allows the definition of functions. These should not be confused with functions in Solidity since they are never part of an external interface of a contract and are part of a namespace separate from the one for Solidity functions.

For the EVM, Yul functions take their arguments (and a return PC) from the stack and also put the results onto the stack. User-defined functions and built-in functions are called in exactly the same way.

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.

Functions declare parameters and return variables, similar to Solidity. To return a value, you assign it to the return variable(s).

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

The `leave` statement can be used to exit the current function. It works like the `return` statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

Note that the EVM dialect has a built-in function called `return` that quits the full execution context (internal message call) and not just the current yul function.

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

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

3.27.5 Specification of Yul

This chapter describes Yul code formally. Yul code is usually placed inside Yul objects, which are explained in their own chapter.

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )?␣
↪)*
Literal =
    (NumberLiteral | StringLiteral | HexLiteral | TrueLiteral | FalseLiteral) ( ':'␣
↪TypeName )?
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Restrictions on the Grammar

Apart from those directly imposed by the grammar, the following restrictions apply :

Switches must have at least one case (including the default case). All case values need to have the same type and distinct values. If all possible values of the expression type are covered, a default case is not allowed (i.e. a switch with a `bool` expression that has both a true and a false case do not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return variables of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of

values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

The `continue` and `break` statements can only be used inside loop bodies and have to be in the same function as the loop (or both have to be at the top level). The `continue` and `break` statements cannot be used in other parts of a loop, not even when it is scoped inside a second loop's body.

The condition part of the for-loop has to evaluate to exactly one value.

The `leave` statement can only be used inside a function.

Functions cannot be defined anywhere inside for loop init blocks.

Literals cannot be larger than the their type. The largest type defined is 256-bit wide.

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (`FunctionDefinition`, `VariableDeclaration`) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks).

As an exception, the scope of the « init » part of the or-loop (the first block) extends across all other parts of the for loop. This means that variables declared in the init part (but not inside a block inside the init part) are visible in all other parts of the for-loop.

Identifiers declared in the other parts of the for loop respect the regular syntactical scoping rules.

This means a for-loop of the form `for { I... } C { P... } { B... }` is equivalent to `{ I... for {} C { P... } { B... } }`.

The parameters and return parameters of functions are visible in the function body and their names have to be distinct.

Variables can only be referenced after their declaration. In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not accessible.

Inside functions, it is not possible to access a variable that was declared outside of that function.

Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. As builtin functions can have side effects, E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM).

If the AST node is a statement, E returns the two state objects and a « mode », which is used for the `break`, `continue` and `leave` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state `L` is a mapping of identifiers `i` to values `v`, denoted as `L[i] = v`.

For an identifier `v`, let `$v` be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L, mode = E(G, L, i1, ..., in)
        // mode has to be regular or leave due to the syntactic restrictions
        if mode is leave then
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G2, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
```

(suite sur la page suivante)

(suite de la page précédente)

```
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluate literals, context doesn't matter
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: HexLiteral) = G, L, hexString(l),
    where hexString decodes l from hex and left-aligns it into 32 bytes
E(G, L, l: StringLiteral) = G, L, utf8EncodeLeftAligned(l),
    where utf8EncodeLeftAligned performs a utf8 encoding of l
    and aligns it left into 32 bytes
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal decoding function
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal decoding function
```

EVM Dialect

The default dialect of Yul currently is the EVM dialect for the currently selected version of the EVM. with a version of the EVM. The only type available in this dialect is `u256`, the 256-bit native type of the Ethereum Virtual Machine. Since it is the default type of this dialect, it can be omitted.

The following table lists all builtin functions (depending on the EVM version) and provides a short description of the semantics of the function / opcode. This document does not want to be a full description of the Ethereum virtual machine. Please refer to a different document if you are interested in the precise semantics.

Opcodes marked with − do not return a result and all others return exactly one value. Opcodes marked with `F`, `H`, `B`, `C` or `I` are present since Frontier, Homestead, Byzantium, Constantinople or Istanbul, respectively.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to but not including position `b` and `storage[p]` signifies the storage contents at slot `p`.

Since Yul manages local variables and control-flow, opcodes that interfere with these features are not available. This includes the `dup` and `swap` instructions as well as `jump` instructions, labels and the `push` instructions.

Instruction			Explanation
stop()	-	F	stop execution, identical to return(0, 0)
add(x, y)		F	x + y
sub(x, y)		F	x - y
mul(x, y)		F	x * y
div(x, y)		F	x / y or 0 if y == 0
sdiv(x, y)		F	x / y, for signed numbers in two's complement, 0 if y == 0
mod(x, y)		F	x % y, 0 if y == 0
smod(x, y)		F	x % y, for signed numbers in two's complement, 0 if y == 0
exp(x, y)		F	x to the power of y
not(x)		F	bitwise « not » of x (every bit of x is negated)
lt(x, y)		F	1 if x < y, 0 otherwise
gt(x, y)		F	1 if x > y, 0 otherwise
slt(x, y)		F	1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		F	1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)		F	1 if x == y, 0 otherwise
iszero(x)		F	1 if x == 0, 0 otherwise
and(x, y)		F	bitwise « and » of x and y
or(x, y)		F	bitwise « or » of x and y
xor(x, y)		F	bitwise « xor » of x and y
byte(n, x)		F	nth byte of x, where the most significant byte is the 0th byte
shl(x, y)		C	logical shift left y by x bits
shr(x, y)		C	logical shift right y by x bits
sar(x, y)		C	signed arithmetic shift right y by x bits
addmod(x, y, m)		F	(x + y) % m with arbitrary precision arithmetic, 0 if m == 0
mulmod(x, y, m)		F	(x * y) % m with arbitrary precision arithmetic, 0 if m == 0
signextend(i, x)		F	sign extend from (i*8+7)th bit counting from least significant
keccak256(p, n)		F	keccak(mem[p...(p+n)))
pc()		F	current position in code
pop(x)	-	F	discard value x
mload(p)		F	mem[p...(p+32))
mstore(p, v)	-	F	mem[p...(p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (only modifies a single byte)
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	size of memory, i.e. largest accessed memory index
gas()		F	gas still available to execution
address()		F	address of the current contract / execution context
balance(a)		F	wei balance at address a
selfbalance()		I	equivalent to balance(address()), but cheaper
caller()		F	call sender (excluding `delegatecall`)
callvalue()		F	wei sent together with the current call

Instruction			Explanation
calldataload(p)		F	call data starting from position p (32 bytes)
calldatasize()		F	size of call data in bytes
calldatacopy(t, f, s)	-	F	copy s bytes from calldata at position f to mem at position t
codesize()		F	size of the code of the current contract / execution context
codecopy(t, f, s)	-	F	copy s bytes from code at position f to mem at position t
extcodesize(a)		F	size of the code at address a
extcodecopy(a, t, f, s)	-	F	like codecopy(t, f, s) but take code at address a
returndatasize()		B	size of the last returndata
returndatacopy(t, f, s)	-	B	copy s bytes from returndata at position f to mem at position t
extcodehash(a)		C	code hash of address a
create(v, p, n)		F	create new contract with code mem[p... (p+n)) and send v wei and return the new
create2(v, p, n, s)		C	create new contract with code mem[p... (p+n)) at address keccak256(0xff . this . s
call(g, a, v, in, insize, out, outsize)		F	call contract at address a with input mem[in... (in+insize)) providing g gas and v
callcode(g, a, v, in, insize, out, outsize)		F	identical to `call` but only use the code from a and stay in the context of the curre
delegatecall(g, a, in, insize, out, outsize)		H	identical to `callcode` but also keep `caller` and `callvalue` *See more*
staticcall(g, a, in, insize, out, outsize)		B	identical to `call(g, a, 0, in, insize, out, outsize)` but do not
return(p, s)	-	F	end execution, return data mem[p... (p+s))
revert(p, s)	-	B	end execution, revert state changes, return data mem[p... (p+s))
selfdestruct(a)	-	F	end execution, destroy current contract and send funds to a
invalid()	-	F	end execution with invalid instruction
log0(p, s)	-	F	log without topics and data mem[p... (p+s))
log1(p, s, t1)	-	F	log with topic t1 and data mem[p... (p+s))
log2(p, s, t1, t2)	-	F	log with topics t1, t2 and data mem[p... (p+s))
log3(p, s, t1, t2, t3)	-	F	log with topics t1, t2, t3 and data mem[p... (p+s))
log4(p, s, t1, t2, t3, t4)	-	F	log with topics t1, t2, t3, t4 and data mem[p... (p+s))
chainid()		I	ID of the executing chain (EIP 1344)
origin()		F	transaction sender
gasprice()		F	gas price of the transaction
blockhash(b)		F	hash of block nr b - only for last 256 blocks excluding current
coinbase()		F	current mining beneficiary
timestamp()		F	timestamp of the current block in seconds since the epoch
number()		F	current block number
difficulty()		F	difficulty of the current block
gaslimit()		F	block gas limit of the current block

Note : The `call*` instructions use the `out` and `outsize` parameters to define an area in memory where the return data is placed. This area is written to depending on how many bytes the called contract returns. If it returns more data, only the first `outsize` bytes are written. You can access the rest of the data using the `returndatacopy` opcode. If it returns less data, then the remaining bytes are not touched at all. You need to use the `returndatasize` opcode to check which part of this memory area contains the return data. The remaining bytes will retain their values as of before the call. If the call fails (it returns `0`), nothing is written to that area, but you can still retrieve the failure data using `returndatacopy`.

In some internal dialects, there are additional functions :

datasize, dataoffset, datacopy

The functions `datasize(x)`, `dataoffset(x)` and `datacopy(t, f, l)`, are used to access other parts of a Yul object.

`datasize` and `dataoffset` can only take string literals (the names of other objects) as arguments and return the size and offset in the data area, respectively. For the EVM, the `datacopy` function is equivalent to `codecopy`.

setimmutable, loadimmutable

The functions `setimmutable("name", value)` and `loadimmutable("name")` are used for the immutable mechanism in Solidity and do not nicely map to pur Yul. The function `setimmutable` assumes that the runtime code of a contract is currently copied to memory at offsot zero. The call to `setimmutable("name", value)` will store `value` at all points in memory that contain a call to `loadimmutable("name")`.

3.27.6 Specification of Yul Object

Yul objects are used to group named code and data sections. The functions `datasize`, `dataoffset` and `datacopy` can be used to access these sections from within code. Hex strings can be used to specify data in hex encoding, regular strings in native encoding. For code, `datacopy` will access its assembled binary representation.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

An example Yul Object is shown below :

```
// A contract consists of a single object with sub-objects representing
// the code to be deployed or other contracts it can create.
// The single "code" node is the executable code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset / datasize
// The current object, sub-objects and data items inside the current object
// are in scope.
object "Contract1" {
    // This is the constructor code of the contract.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // first create "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // This will turn into codecopy for EVM
        datacopy(offset, dataoffset("Contract2"), size)
        // constructor parameter is a single number 0x1234
        mstore(add(offset, size), 0x1234)
        pop(create(offset, add(size, 32), 0))
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // now return the runtime object (the currently
        // executing code is the constructor code)
        size := datasize("runtime")
        offset := allocate(size)
        // This will turn into a memory->memory copy for eWASM and
        // a codecopy for EVM
        datacopy(offset, dataoffset("runtime"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "runtime" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // runtime code

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // Embedded object. Use case is that the outside is a factory contract,
    // and Contract2 is the code to be created by the factory
    object "Contract2" {
        code {
            // code here ...
        }

        object "runtime" {
            code {
                // code here ...
            }
        }

        data "Table1" hex"4123"
    }
}
```

3.27.7 Yul Optimizer

The Yul optimizer operates on Yul code and uses the same language for input, output and intermediate states. This allows for easy debugging and verification of the optimizer.

Please see the documentation in the source code for more details about its internals.

If you want to use Solidity in stand-alone Yul mode, you activate the optimizer using `--optimize` :

```
solc --strict-assembly --optimize
```

In Solidity mode, the Yul optimizer is activated together with the regular optimizer.

Optimization step sequence

By default the Yul optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option :

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new optimization opportunities for others that were already applied so repeating steps is often beneficial. By enclosing part of the sequence in square brackets (`[]`) you tell the optimizer to repeatedly apply that part until it no longer improves the size of the resulting assembly. You can use brackets multiple times in a single sequence but they cannot be nested.

The following optimization steps are available :

Abbreviation	Full name
f	BlockFlattener
l	CircularReferencesPruner
c	CommonSubexpressionEliminator
C	ConditionalSimplifier
U	ConditionalUnsimplifier
n	ControlFlowSimplifier
D	DeadCodeEliminator
v	EquivalentFunctionCombiner
e	ExpressionInliner
j	ExpressionJoiner
s	ExpressionSimplifier
x	ExpressionSplitter
I	ForLoopConditionIntoBody
O	ForLoopConditionOutOfBody
o	ForLoopInitRewriter
i	FullInliner
g	FunctionGrouper
h	FunctionHoister
T	LiteralRematerialiser
L	LoadResolver
M	LoopInvariantCodeMotion
r	RedundantAssignEliminator
m	Rematerialiser
V	SSAReverser
a	SSATransform
t	StructuralSimplifier
u	UnusedPruner
d	VarDeclInitializer

Some steps depend on properties ensured by `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter`. For this reason the Yul optimizer always applies them before applying any steps supplied by the user.

3.27.8 Complete ERC20 Example

```
object "Token" {
    code {
        // Store the creator in slot zero.
        sstore(0, caller())

        // Deploy the contract
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Protection against sending Ether
            require(iszero(callvalue()))

            // Dispatcher
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ {
                returnUint(balanceOf(decodeAsAddress(0)))
            }
            case 0x18160ddd /* "totalSupply()" */ {
                returnUint(totalSupply())
            }
            case 0xa9059cbb /* "transfer(address,uint256)" */ {
                transfer(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
                transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
                returnTrue()
            }
            case 0x095ea7b3 /* "approve(address,uint256)" */ {
                approve(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0xdd62ed3e /* "allowance(address,address)" */ {
                returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
            }
            case 0x40c10f19 /* "mint(address,uint256)" */ {
                mint(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            default {
                revert(0, 0)
            }

            function mint(account, amount) {
                require(calledByOwner())

                mintTokens(amount)
                addToBalance(account, amount)
                emitTransfer(0, account, amount)
            }
            function transfer(to, amount) {
                executeTransfer(caller(), to, amount)
            }
```

(suite sur la page suivante)

(suite de la page précédente)

```
            function approve(spender, amount) {
                revertIfZeroAddress(spender)
                setAllowance(caller(), spender, amount)
                emitApproval(caller(), spender, amount)
            }
            function transferFrom(from, to, amount) {
                decreaseAllowanceBy(from, caller(), amount)
                executeTransfer(from, to, amount)
            }

            function executeTransfer(from, to, amount) {
                revertIfZeroAddress(to)
                deductFromBalance(from, amount)
                addToBalance(to, amount)
                emitTransfer(from, to, amount)
            }


            /* ---------- calldata decoding functions ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
                if iszero(iszero(and(v,␣
↪not(0xffffffffffffffffffffffffffffffffffffffff)))) {
                    revert(0, 0)
                }
            }
            function decodeAsUint(offset) -> v {
                let pos := add(4, mul(offset, 0x20))
                if lt(calldatasize(), add(pos, 0x20)) {
                    revert(0, 0)
                }
                v := calldataload(pos)
            }
            /* ---------- calldata encoding functions ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
            function returnTrue() {
                returnUint(1)
            }

            /* -------- events ---------- */
            function emitTransfer(from, to, amount) {
                let signatureHash :=␣
↪0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
                emitEvent(signatureHash, from, to, amount)
            }
            function emitApproval(from, spender, amount) {
                let signatureHash :=␣
↪0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
                emitEvent(signatureHash, from, spender, amount)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }
            function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
                mstore(0, nonIndexed)
                log3(0, 0x20, signatureHash, indexed1, indexed2)
            }

            /* -------- storage layout ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            function accountToStorageOffset(account) -> offset {
                offset := add(0x1000, account)
            }
            function allowanceStorageOffset(account, spender) -> offset {
                offset := accountToStorageOffset(account)
                mstore(0, offset)
                mstore(0x20, spender)
                offset := keccak256(0, 0x40)
            }

            /* -------- storage access ---------- */
            function owner() -> o {
                o := sload(ownerPos())
            }
            function totalSupply() -> supply {
                supply := sload(totalSupplyPos())
            }
            function mintTokens(amount) {
                sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
            }
            function balanceOf(account) -> bal {
                bal := sload(accountToStorageOffset(account))
            }
            function addToBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                sstore(offset, safeAdd(sload(offset), amount))
            }
            function deductFromBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                let bal := sload(offset)
                require(lte(amount, bal))
                sstore(offset, sub(bal, amount))
            }
            function allowance(account, spender) -> amount {
                amount := sload(allowanceStorageOffset(account, spender))
            }
            function setAllowance(account, spender, amount) {
                sstore(allowanceStorageOffset(account, spender), amount)
            }
            function decreaseAllowanceBy(account, spender, amount) {
                let offset := allowanceStorageOffset(account, spender)
                let currentAllowance := sload(offset)
                require(lte(amount, currentAllowance))
                sstore(offset, sub(currentAllowance, amount))
            }

            /* ---------- utility functions ---------- */
            function lte(a, b) -> r {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                r := iszero(gt(a, b))
            }
            function gte(a, b) -> r {
                r := iszero(lt(a, b))
            }
            function safeAdd(a, b) -> r {
                r := add(a, b)
                if or(lt(r, a), lt(r, b)) { revert(0, 0) }
            }
            function calledByOwner() -> cbo {
                cbo := eq(owner(), caller())
            }
            function revertIfZeroAddress(addr) {
                require(addr)
            }
            function require(condition) {
                if iszero(condition) { revert(0, 0) }
            }
        }
    }
}
```

3.28 Style Guide

3.28.1 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.

Note : 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 judgement. Look at other examples and decide what looks best. And don't hesitate to ask !

3.28.2 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 :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

contract A {
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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 :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.7.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

Maximum Line Length

Keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `);`, should be placed on the final line by itself.

Function Calls

Yes :

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No :

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

Event Definitions and Event Emitters

Yes :

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    recipient,
    publicKey,
    amount,
    options
);
```

No :

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

contract A {
    // ...
}


import "./Owned.sol";
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

contract A {
    constructor() public {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // ...
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

contract A {

    // External functions
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() public {
        // ...
    }

    // 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() public { spam(); }
```

Immediately before a comma, semicolon :

Yes :

```
function spam(uint i, Coin coin) public;
```

No :

```
function spam(uint i , Coin coin) public ;
```

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 receive and fallback functions :

Yes :

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

No :

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

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 preceded by a single space.

Yes :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

contract Coin {
    struct Bank {
        address owner;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        uint balance;
    }
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;

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 preceded by a single space.

Yes :

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyowner returns (uint) {
    return x + 1;
}
```

No :

```
function increment(uint x) public pure returns (uint)
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

The modifier order for a function should be :

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yes :

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function shutdown() public onlyowner {
    selfdestruct(owner);
}
```

No :

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function shutdown() 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
)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public
{
    doSomething();
}
```

No :

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its 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();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the *Maximum Line Length* section.

Yes :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

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 :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}
contract C {
    constructor(uint, uint) public {
    }
}
contract D {
    constructor(uint) public {
    }
}

contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public
    {
        // do something with param5
        x = param5;
    }
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;


// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract C {
    constructor(uint, uint) public {
    }
}


contract D {
    constructor(uint) public {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    public {
        x = param5;
    }
}


contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public {
            x = param5;
        }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible :

```
function shortFunction() public { 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

In variable declarations, do not separate the keyword `mapping` from its type by a space. Do not separate any nested `mapping` keyword from its type by whitespace.

Yes :

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
```

(suite sur la page suivante)

(suite de la page précédente)

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

No :

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

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;
```

3.28.3 Order of Layout

Layout contract elements in the following order :

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order :

1. Type declarations
2. State variables
3. Events
4. Functions

Note : It might be clearer to declare types close to their use in events or state variables.

3.28.4 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 supersede 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 initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms in mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

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. Examples : `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
— Contract and library names should also match their filenames.
— If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is `Congress` and the library name is `Owned`, then their associated filenames should be `Congress.sol` and `Owned.sol`.

Yes :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;


// Owned.sol
contract Owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.7.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

No :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;


// owned.sol
contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol` :

```
import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

Struct Names

Structs should be named using the CapWords style. Examples : `MyCoin`, `Position`, `PositionXY`.

Event Names

Events should be named using the CapWords style. Examples : `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Function Names

Functions other than constructors should use mixedCase. Examples : `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Function Argument Names

Function arguments should use mixedCase. Examples : `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

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 Variable Names

Use mixedCase. Examples : `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constants

Constants should be named with all capital letters with underscores separating words. Examples : `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Names

Use mixedCase. Examples : `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples : `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Avoiding Naming Collisions

— `single_trailing_underscore_`

This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

3.28.5 NatSpec

Solidity contracts can have a form of comments that are the basis of the Ethereum Natural Language Specification Format.

Add comments above functions or contracts following doxygen notation of one or multiple lines starting with `///` or a multiline comment starting with `/**` and ending with `*/`.

For example, the contract from a simple smart contract with the comments added looks like the one below :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.7.0;


/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
    function get() public view returns (uint) {
        return storedData;
    }
}
```

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

Please see the section about NatSpec for a detailed explanation.

3.29 Common Patterns

3.29.1 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 `transfer` call, this is not recommended as it introduces a potential security risk. You may read more about this on the *Security Considerations* page.

The following 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 no longer the richest, you receive the funds of the person who is now the richest.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        require(msg.value > mostSent, "Not enough money sent.");
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

This is as opposed to the more intuitive sending pattern :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.7.0;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        require(msg.value > mostSent, "Not enough money sent.");
        // This line can cause problems (explained below).
        richest.transfer(msg.value);
        richest = msg.sender;
        mostSent = msg.value;
    }
}
```

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 receive or fallback function which fails (e.g. by using `revert()` or by just consuming more than the 2300 gas stipend transferred to them). That way, whenever `transfer` is called to deliver funds to the « poisoned » contract, it will fail and thus also `becomeRichest` will fail, with the contract being stuck forever.

In contrast, if you use the « withdraw » pattern from the first example, the attacker can only cause his or her own withdraw to fail and not the rest of the contract's workings.

3.29.2 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 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 section is about.

The use of **function modifiers** makes these restrictions highly readable.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
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)
    {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
        // 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)
        public
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(
            now >= _time,
            "Function called too early."
        );
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        public
        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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // This was dangerous before Solidity version 0.4.0,
    // where it was possible to skip the part after `_;`.
    modifier costs(uint _amount) {
        require(
            msg.value >= _amount,
            "Not enough Ether provided."
        );
        _;
        if (msg.value > _amount)
            msg.sender.transfer(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
        public
        payable
        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.

3.29.3 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 outcome ».

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.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.7.0;

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) {
        require(
            stage == _stage,
            "Function cannot be called at this time."
        );
        _;
    }

    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()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

3.30 List of Known Bugs

Below, you can find a JSON-formatted list of some of the known security-relevant bugs in the Solidity compiler. The file itself is hosted in the Github repository. The list stretches back as far as version 0.3.0, bugs known to be present only in versions preceding that are not listed.

There is another file called bugs_by_version.json, which can be used to check which bugs affect a specific version of the compiler.

Contract source verification tools and also other tools interacting with contracts should consult this list according to the following criteria :

— It is mildly suspicious if a contract was compiled with a nightly compiler version instead of a released version. This list does not keep track of unreleased or nightly versions.

— It is also mildly suspicious if a contract was compiled with a version that was not the most recent at the time the contract was created. For contracts created from other contracts, you have to follow the creation chain back to a transaction and use the date of that transaction as creation date.

— It is highly suspicious if a contract was compiled with a compiler that contains a known bug and the contract was created at a time where a newer compiler version containing a fix was already released.

The JSON file of known bugs below is an array of objects, one for each bug, with the following keys :

name Unique name given to the bug

summary Short description of the bug

description Detailed description of the bug

link URL of a website with more detailed information, optional

introduced The first published compiler version that contained the bug, optional

fixed The first published compiler version that did not contain the bug anymore

publish The date at which the bug became known publicly, optional

severity Severity of the bug : very low, low, medium, high. Takes into account discoverability in contract tests, likelihood of occurrence and potential damage by exploits.

conditions Conditions that have to be met to trigger the bug. The following keys can be used : `optimizer`, Boolean value which means that the optimizer has to be switched on to enable the bug. `evmVersion`, a string that indicates which EVM version compiler settings trigger the bug. The string can contain comparison operators. For example, `">=constantinople"` means that the bug is present when the EVM version is set to `constantinople` or later. If no conditions are given, assume that the bug is present.

check This field contains different checks that report whether the smart contract contains the bug or not. The first type of check are Javascript regular expressions that are to be matched against the source code (« source-regex ») if the bug is present. If there is no match, then the bug is very likely not present. If there is a match, the bug might be present. For improved accuracy, the checks should be applied to the source code after stripping comments. The second type of check are patterns to be checked on the compact AST of the Solidity program (« ast-compact-json-path »). The specified search query is a JsonPath expression. If at least one path of the Solidity AST matches the query, the bug is likely present.

```
[
    {
        "name": "MissingEscapingInFormatting",
        "summary": "String literals containing double backslash characters passed␣
↪directly to external or encoding function calls can lead to a different string␣
↪being used when ABIEncoderV2 is enabled.",
        "description": "When ABIEncoderV2 is enabled, string literals passed directly␣
↪to encoding functions or external function calls are stored as strings in the␣
↪intemediate code. Characters outside the printable range are handled correctly, but␣
↪backslashes are not escaped in this procedure. This leads to double backslashes␣
↪being reduced to single backslashes and consequently re-interpreted as escapes␣
↪potentially resulting in a different string being encoded.",
        "introduced": "0.5.14",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "ArraySliceDynamicallyEncodedBaseType",
        "summary": "Accessing array slices of arrays with dynamically encoded base␣
↪types (e.g. multi-dimensional arrays) can result in invalid data being read.",
        "description": "For arrays with dynamically sized base types, index range␣
↪accesses that use a start expression that is non-zero will result in invalid array␣
↪slices. Any index access to such array slices will result in data being read from␣
↪incorrect calldata offsets. Array slices are only supported for dynamic calldata␣
↪types and all problematic type require ABIEncoderV2 to be enabled.",
        "introduced": "0.6.0",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "ImplicitConstructorCallvalueCheck",
        "summary": "The creation code of a contract that does not define a␣
↪constructor but has a base that does define a constructor did not revert for calls␣
↪with non-zero value.",
        "description": "Starting from Solidity 0.4.5 the creation code of contracts␣
↪without explicit payable constructor is supposed to contain a callvalue check that␣
↪results in contract creation reverting, if non-zero value is passed. However, this␣
↪check was missing in case no explicit constructor was defined in a contract at all,␣
↪but the contract has a base that does define a constructor. In these cases it is␣
↪possible to send value in a contract creation transaction or using inline assembly␣
↪without revert, even though the creation code is supposed to be non-payable.",
        "introduced": "0.4.5",
        "fixed": "0.6.8",
        "severity": "very low"
    },
    {
        "name": "TupleAssignmentMultiStackSlotComponents",
        "summary": "Tuple assignments with components that occupy several stack slots,
↪ i.e. nested tuples, pointers to external functions or references to dynamically␣
↪sized calldata arrays, can result in invalid values.",
        "description": "Tuple assignments did not correctly account for tuple␣
↪components that occupy multiple stack slots in case the number of stack slots␣
↪differs between left-hand-side and right-hand-side. This can either happen in the␣
↪presence of nested tuples or if the right-hand-side contains external function␣
↪pointers or references to dynamic calldata arrays, while the left-hand-side␣
↪contains an omission.",
        "introduced": "0.1.6",
        "fixed": "0.6.6",
        "severity": "very low"
    },
    {
        "name": "MemoryArrayCreationOverflow",
        "summary": "The creation of very large memory arrays can result in␣
↪overlapping memory regions and thus memory corruption.",
        "description": "No runtime overflow checks were performed for the length of␣
↪memory arrays during creation. In cases for which the memory size of an array in␣
↪bytes, i.e. the array length times 32, is larger than 2^256-1, the memory␣
↪allocation will overflow, potentially resulting in overlapping memory areas. The␣
↪length of the array is still stored correctly, so copying or iterating over such an␣
↪array will result in out-of-gas.",
        "link": "https://solidity.ethereum.org/2020/04/06/memory-creation-overflow-
↪bug/",
        "introduced": "0.2.0",
        "fixed": "0.6.5",
        "severity": "low"
    },
    {
        "name": "YulOptimizerRedundantAssignmentBreakContinue",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and␣
↪continue and break statements.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared␣
↪inside a for loop if they can be removed in a control-flow branch that ends with␣
↪``break`` or ``continue`` even though they cannot be removed in other control-flow␣
↪branches. Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.6.0",
        "fixed": "0.6.1",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "name": "privateCanBeOverridden",
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and␣
↪cannot be called directly from the derived contract, it is still possible to␣
↪declare a function of the same name and type and thus change the behaviour of the␣
↪base contract's function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },
    {
        "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and␣
↪continue and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared␣
↪inside a for loop if they can be removed in a control-flow branch that ends with␣
↪``break`` or ``continue`` even though they cannot be removed in other control-flow␣
↪branches. Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.5.8",
        "fixed": "0.5.16",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "name": "ABIEncoderV2LoopYulOptimizer",
        "summary": "If both the experimental ABIEncoderV2 and the experimental Yul␣
↪optimizer are activated, one component of the Yul optimizer may reuse data in␣
↪memory that has been changed in the meantime.",
        "description": "The Yul optimizer incorrectly replaces ``mload`` and␣
↪``sload`` calls with values that have been previously written to the load location␣
↪(and potentially changed in the meantime) if all of the following conditions are␣
↪met: (1) there is a matching ``mstore`` or ``sstore`` call before; (2) the contents␣
↪of memory or storage is only changed in a function that is called (directly or␣
↪indirectly) in between the first store and the load call; (3) called function␣
↪contains a for loop where the same memory location is changed in the condition or␣
↪the post or body block. When used in Solidity mode, this can only happen if the␣
↪experimental ABIEncoderV2 is activated and the experimental Yul optimizer has been␣
↪activated manually in addition to the regular optimizer in the compiler settings.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "introduced": "0.5.14",
        "fixed": "0.5.15",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true,
            "optimizer": true,
            "yulOptimizer": true
        }
    },
    {
        "name":
↪"ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
        "summary": "Reading from calldata structs that contain dynamically encoded,␣
↪but statically-sized members can result in incorrect values.",
        "description": "When a calldata struct contains a dynamically encoded, but␣
↪statically-sized member, the offsets for all subsequent struct members are␣
↪calculated incorrectly. All reads from such members will result in invalid values.␣
↪Only calldata structs are affected, i.e. this occurs in external functions with␣
↪such structs as argument. Using affected structs in storage or memory or as␣
↪arguments to public functions on the other hand works correctly.",
        "introduced": "0.5.6",
        "fixed": "0.5.11",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of␣
↪different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher␣
↪order bits set. In order to fit into a shared storage slot, these have to be set to␣
↪zero. When a conversion is done at the same time, the bits to set to zero were␣
↪incorrectly determined from the source and not the target type. This means that␣
↪such copy operations can lead to incorrect values being stored.",
        "link": "https://blog.ethereum.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },
    {
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized␣
↪arrays are not read properly when directly encoded in external function calls or in␣
↪abi.encode*.",
        "description": "When storage arrays whose elements occupy more than a single␣
↪storage slot are directly encoded in external function calls or using abi.encode*,␣
↪their elements are read in an overlapping manner, i.e. the element pointer is not␣
↪properly advanced between reads. This is not a problem when the storage data is␣
↪first copied to a memory variable or if the storage array only contains value types␣
↪or dynamically-sized arrays.",
        "link": "https://blog.ethereum.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
        "conditions": {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "ABIEncoderV2": true
        }
    },
    {
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that␣
↪contain dynamically-sized arrays reverts or decodes to invalid data.",
        "description": "During construction of a contract, constructor parameters are␣
↪copied from the code section to memory for decoding. The amount of bytes to copy␣
↪was calculated incorrectly in case all parameters are statically-sized but contain␣
↪dynamically-sized arrays as struct members or inner arrays. Such types are only␣
↪available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special␣
↪jump target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "name": "UninitializedFunctionPointerInConstructor_0.4.x",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special␣
↪jump target only considered the situation after deployment.",
        "introduced": "0.4.5",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect␣
↪event signature hash",
        "description": "Instead of using the type `address` in the hashed signature,␣
↪the actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "name": "IncorrectEventSignatureInLibraries_0.4.x",
        "summary": "Contract types used in events in libraries cause an incorrect␣
↪event signature hash",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "description": "Instead of using the type `address` in the hashed signature,␣
↪the actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.3.0",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but␣
↪can also overwrite other parts of the encoded data.",
        "link": "https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "ABIEncoderV2PackedStorage_0.4.x",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but␣
↪can also overwrite other parts of the encoded data.",
        "link": "https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.4.19",
        "fixed": "0.4.26",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "name": "IncorrectByteInstructionOptimization",
        "summary": "The optimizer incorrectly handles byte opcodes whose second␣
↪argument is 31 or a constant expression that evaluates to 31. This can result in␣
↪unexpected values.",
        "description": "The optimizer incorrectly handles byte opcodes that use the␣
↪constant 31 as second argument. This can happen when performing index access on␣
↪bytesNN types with a compile-time constant value (not index) of 31 or when using␣
↪the byte opcode in inline assembly.",
        "link": "https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.7",
        "severity": "very low",
        "conditions": {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "optimizer": true
        }
    },
    {
        "name": "DoubleShiftSizeOverflow",
        "summary": "Double bitwise shifts by large constants whose sum overflows 256␣
↪bits can result in unexpected values.",
        "description": "Nested logical shift operations whose total shift size is␣
↪2**256 or more are incorrectly optimized. This only applies to shifts by numbers of␣
↪bits that are compile-time constant expressions.",
        "link": "https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.6",
        "severity": "low",
        "conditions": {
            "optimizer": true,
            "evmVersion": ">=constantinople"
        }
    },
    {
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256␣
↪bits can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value␣
↪range of the type. Literal numbers as exponents are unaffected as are exponents or␣
↪bases of type uint256.",
        "link": "https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/",
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "link": "https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..
↪[?(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks␣
↪if such functions are present, not if they are called, which is required for the␣
↪contract to be affected.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "link": "https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^
↪\\]]*\\]\\s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a␣
↪compiler warning about the old-style constructor, so contracts only using new-style␣
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
    {
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most␣
↪five external functions in total, this function is called instead of the fallback␣
↪function if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
    },
    {
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution␣
↪outcome, but converts the value returned by the functioned called to a boolean␣
↪instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides.␣
↪This value is interpreted as a boolean and put onto the stack. This means if the␣
↪called function returns at least 32 zero bytes, .delegatecall() returns false even␣
↪if the call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "name": "SkipEmptyStringLiteral",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "summary": "If \"\" is used in a function call, the following function␣
↪arguments will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding␣
↪of all arguments following this is shifted left by 32 bytes and thus the function␣
↪call data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode␣
↪by routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying␣
↪memory, ignored its return value. On the public chain, calls to the identity␣
↪precompile can be made in a way that they never fail, but this might be different␣
↪on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was simplified to just␣
↪use the empty state, but this implementation was not done properly. This bug can␣
↪cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly␣
↪and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same␣
↪32 byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "link": "https://blog.ethereum.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not␣
↪properly reset across control flow paths and thus the optimizer sometimes thought␣
↪that the result of a SHA3 operation is already present on the stack. This could␣
↪result in data corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how␣
↪much Ether was sent with a call, the library function incorrectly assumed that␣
↪Ether was sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if␣
↪no Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.
↪",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite␣
↪loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this␣
↪length is zero, code was generated that did not terminate and thus consumed all gas.
↪",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "name": "OptimizerClearStateOnCodePathJoin",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was not done correctly.␣
↪This bug can cause data corruption, but it is probably quite hard to use for␣
↪targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned␣
↪before comparison.",
        "description": "Two variables of type bytesNN were considered different if␣
↪their higher order bits, which are not part of the actual value, were different. An␣
↪attacker might use this to reach seemingly unreachable code paths by providing␣
↪incorrectly formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32␣
↪bytes did not correctly clean the higher order bits, causing corruption in other␣
↪array elements.",
        "description": "Multiple elements of an array of values that are shorter than␣
↪17 bytes are packed into the same storage slot. Writing to a single element of such␣
↪an array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

3.31 Contributing

Help is always appreciated !

To get started, you can try *Compilation à partir des sources* 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 good first issue which are meant as introductory issues for external contributors.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project - in the issues, pull requests, or Gitter channels - you agree to abide by its terms.

3.31.1 Team Calls

If you have issues or pull requests to discuss, or are interested in hearing what the team and contributors are working on, you can join our public team calls :

— Monday at 12pm CET
— Wednesday at 3pm CET

Both calls take place on Google Hangouts.

3.31.2 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.

3.31.3 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. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the Solidity Development Gitter channel (different from the one mentioned above, this one is focused on compiler and language development instead of language use) first.

New features and bugfixes should be added to the `Changelog.md` file : please follow the style of previous entries, when applicable.

Finally, please make sure you respect the coding style 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 !

3.31.4 Running the compiler tests

Prerequisites

Some tests require the evmone library, others require libz3. The test script tries to discover the location of the `evmone` library, which can be located in the current directory, installed on the system level, or the `deps` folder in the project top level. The required file is called `libevmone.so` on Linux systems, `evmone.dll` on Windows systems and `libevmone.dylib` on macOS.

Running the tests

Solidity includes different types of tests, most of them bundled into the Boost C++ Test Framework application `soltest`. Running `build/test/soltest` or its wrapper `scripts/soltest.sh` is sufficient for most changes.

The `./scripts/tests.sh` script executes most Solidity tests automatically, including those bundled into the Boost C++ Test Framework application `soltest` (or its wrapper `scripts/soltest.sh`), as well as command line tests and compilation tests.

The test system automatically tries try to discover the location of the `evmone` library starting from the current directory. The required file is called `libevmone.so` on Linux systems, `evmone.dll` on Windows systems and `libevmone.dylib` on macOS. If it is not found, tests that use it are skipped. These tests are `libsolididty/semanticTests`, `libsolidity/GasCosts`, `libsolidity/SolidityEndToEndTest`, part of the soltest suite. To run all tests, download the library from GitHub and place it in the project root path or inside the `deps` folder.

If the `libz3` library is not installed on your system, you should disable the SMT tests by exporting `SMT_FLAGS=--no-smt` before running `./scripts/tests.sh` or running `./scripts/soltest.sh --no-smt`. These tests are `libsolidity/smtCheckerTests` and `libsolidity/smtCheckerTestsJSON`.

Note : To get a list of all unit tests run by Soltest, run `./build/test/soltest --list_content=HRF`.

For quicker results you can run a subset of, or specific tests.

To run a subset of tests, you can use filters : `./scripts/soltest.sh -t TestSuite/TestName`, where `TestName` can be a wildcard `*`.

Or, for example, to run all the tests for the yul disambiguator : `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` has extensive help on all of the options available.

See especially :
- show_progress (-p) to show test completion,
- run_test (-t) to run specific tests cases, and
- report-level (-r) give a more detailed report.

Note : Those working in a Windows environment wanting to run the above basic sets without libz3. Using Git Bash, you use : `./build/test/Release/soltest.exe -- --no-smt`. If you are running this in plain Command Prompt, use `.\build\test\Release\soltest.exe -- --no-smt`.

If you want to debug using GDB, make sure you build differently than the « usual ». For example, you could run the following command in your `build` folder :

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

This creates symbols so that when you debug a test using the `--debug` flag, you have access to functions and variables in which you can break or print with.

The CI runs additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

Writing and running syntax tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside the `tests/libsolidity/syntaxTests` folder. These files must contain annotations, stating the expected result(s) of the respective test. The test suite compiles and checks them against the given expectations.

For example : `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator `// ----`. The comments that follow the separator are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. If you want the contract to compile without any errors or warning you can leave out the separator and the comments that follow it.

In the above example, the state variable `variable` was declared twice, which is not allowed. This results in a `DeclarationError` stating that the identifier was already declared.

The `isoltest` tool is used for these tests and you can find it under `./build/test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable` :

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./build/test/isoltest` again results in a test failure :

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

`isoltest` prints the expected result next to the obtained result, and also provides a way to edit, update or skip the current contract file, or quit the application.

It offers several options for failing tests :

- `edit` : `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in that order).
- `update` : Updates the expectations for contract under test. This updates the annotations by removing unmet expectations and adding missing expectations. The test is then run again.
- `skip` : Skips the execution of this particular test.
- `quit` : Quits `isoltest`.

All of these options apply to the current contract, expect `quit` which stops the entire testing process.

Automatically updating the test above changes it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It now passes again :

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note : Choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

3.31.5 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and run a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g., the code contains an error. This way, fuzzing tools can find internal problems in the compiler.

We mainly use AFL for fuzzing. You need to download and install the AFL packages from your repositories (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler :

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage you should be able to see a message similar to the following :

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the cmake flags pointing to AFL's clang binaries :

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Otherwise, upon execution the fuzzer halts with an error saying binary is not instrumented :

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This makes it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests :

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of. There is also a tool called `afl-cmin` that can trim input files that result in similar behaviour of the binary.

Now run the fuzzer (the `-m` extends the size of memory to 60 MB) :

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer creates source files that lead to failures in `/tmp/fuzzer_reports`. Often it finds many similar source files that produce the same error. You can use the tool `scripts/uniqueErrors.sh` to filter out the unique errors.

3.31.6 Whiskers

Whiskers is a string templating system similar to Mustache. It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache. The template markers `{{` and `}}` are replaced by < and > in order to aid parsing and avoid conflicts with *Yul* (The symbols < and > are invalid in inline assembly, while `{` and `}` are used to delimit blocks). Another limitation is that lists are only resolved one depth and they do not recurse. This may change in the future.

A rough specification is the following :

Any occurrence of `<name>` is replaced by the string-value of the supplied variable `name` without any escaping and without iterated replacements. An area can be delimited by `<#name>...</name>`. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any `<inner>` items by their respective value. Top-level variables can also be used inside such areas.

There are also conditionals of the form `<?name>...<!name>...</name>`, where template replacements continue recursively either in the first or the second segment depending on the value of the boolean parameter `name`. If `<?+name>...<!+name>...</+name>` is used, then the check is whether the string parameter `name` is non-empty.

3.31.7 Documentation Style Guide

The following are style recommendations specifically for documentation contributions to Solidity.

English Language

Use English, with British English spelling preferred, unless using project or brand names. Try to reduce the usage of local slang and references, making your language as clear to all readers as possible. Below are some references to help :

— Simplified technical English
— International English
— British English spelling

Note : While the official Solidity documentation is written in English, there are community contributed *Traductions* in other languages available.

Title Case for Headings

Use title case for headings. This means capitalise all principal words in titles, but not articles, conjunctions, and prepositions unless they start the title.

For example, the following are all correct :

— Title Case for Headings
— For Headings Use Title Case
— Local and State Variable Names
— Order of Layout

Expand Contractions

Use expanded contractions for words, for example :

— « Do not » instead of « Don't ».
— « Can not » instead of « Can't ».

Active and Passive Voice

Active voice is typically recommended for tutorial style documentation as it helps the reader understand who or what is performing a task. However, as the Solidity documentation is a mixture of tutorials and reference content, passive voice is sometimes more applicable.

As a summary :

— Use passive voice for technical reference, for example language definition and internals of the Ethereum VM.
— Use active voice when describing recommendations on how to apply an aspect of Solidity.

For example, the below is in passive voice as it specifies an aspect of Solidity :

> Functions can be declared `pure` in which case they promise not to read from or modify the state.

For example, the below is in active voice as it discusses an application of Solidity :

> When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings.

Common Terms

— « Function parameters » and « return variables », not input and output parameters.

Code Examples

A CI process tests all code block formatted code examples that begin with `pragma solidity`, `contract`, `library` or `interface` using the `./test/cmdlineTests.sh` script when you create a PR. If you are adding new code examples, ensure they work and pass tests before creating the PR.

Ensure that all code examples begin with a `pragma` version that spans the largest where the contract code is valid. For example `pragma solidity >=0.4.0 <0.7.0;`.

Running Documentation Tests

Make sure your contributions pass our documentation tests by running `./scripts/docs.sh` that installs dependencies needed for documentation and checks for any problems such as broken links or syntax issues.

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y