**Layout of a Solidity Source File**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#layout-of-a-solidity-source-file)

Source files can contain an arbitrary number of [contract definitions](https://docs.soliditylang.org/en/v0.8.19/structure-of-a-contract.html#contract-structure), [import](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#import) , [pragma](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#pragma) and [using for](https://docs.soliditylang.org/en/v0.8.19/contracts.html#using-for) directives and [struct](https://docs.soliditylang.org/en/v0.8.19/types.html#structs), [enum](https://docs.soliditylang.org/en/v0.8.19/types.html" \l "enums), [function](https://docs.soliditylang.org/en/v0.8.19/contracts.html#functions), [error](https://docs.soliditylang.org/en/v0.8.19/contracts.html#errors) and [constant variable](https://docs.soliditylang.org/en/v0.8.19/contracts.html#constants) definitions.

**SPDX License Identifier**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#spdx-license-identifier)

Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable [SPDX license identifiers](https://spdx.org/). Every source file should start with a comment indicating its license:

// SPDX-License-Identifier: MIT

The compiler does not validate that the license is part of the [list allowed by SPDX](https://spdx.org/licenses/), but it does include the supplied string in the [bytecode metadata](https://docs.soliditylang.org/en/v0.8.19/metadata.html#metadata).

If you do not want to specify a license or if the source code is not open-source, please use the special value UNLICENSED. Note that UNLICENSED (no usage allowed, not present in SPDX license list) is different from UNLICENSE (grants all rights to everyone). Solidity follows [the npm recommendation](https://docs.npmjs.com/cli/v7/configuring-npm/package-json#license).

Supplying this comment of course does not free you from other obligations related to licensing like having to mention a specific license header in each source file or the original copyright holder.

The comment is recognized by the compiler anywhere in the file at the file level, but it is recommended to put it at the top of the file.

More information about how to use SPDX license identifiers can be found at the [SPDX website](https://spdx.org/ids-how).

**Pragmas**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#pragmas)

The pragma keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project. If you [import](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#import) another file, the pragma from that file does *not* automatically apply to the importing file.

**Version Pragma**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#version-pragma)

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

The version pragma is used as follows: pragma solidity ^0.5.2;

A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0 (this second condition is added by using ^). Because there will be no breaking changes until version 0.6.0, you can be sure that your code compiles the way you intended. The exact version of the compiler is not fixed, so that bugfix releases are still possible.

It is possible to specify more complex rules for the compiler version, these follow the same syntax used by [npm](https://docs.npmjs.com/cli/v6/using-npm/semver).

**Note**

Using the version pragma *does not* change the version of the compiler. It also *does not* enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.

**ABI Coder Pragma**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#abi-coder-pragma)

By using pragma abicoder v1 or pragma abicoder v2 you can select between the two implementations of the ABI encoder and decoder.

The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. Apart from supporting more types, it involves more extensive validation and safety checks, which may result in higher gas costs, but also heightened security. It is considered non-experimental as of Solidity 0.6.0 and it is enabled by default starting with Solidity 0.8.0. The old ABI coder can still be selected using pragma abicoder v1;.

The set of types supported by the new encoder is a strict superset of the ones supported by the old one. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-abicoder v2 contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling abicoder v2 for your contract is enough to make the error go away.

**Note**

This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.

**Note**

Up to Solidity 0.7.4, it was possible to select the ABI coder v2 by using pragma experimental ABIEncoderV2, but it was not possible to explicitly select coder v1 because it was the default.

**Experimental Pragma**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#experimental-pragma)

The second pragma is the experimental pragma. It can be used to enable features of the compiler or language that are not yet enabled by default. The following experimental pragmas are currently supported:

**ABIEncoderV2**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#abiencoderv2)

Because the ABI coder v2 is not considered experimental anymore, it can be selected via pragma abicoder v2 (please see above) since Solidity 0.7.4.

**SMTChecker[ℑ](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html" \l "smtchecker" \o "Permalink to this heading)**

This component has to be enabled when the Solidity compiler is built and therefore it is not available in all Solidity binaries. The [build instructions](https://docs.soliditylang.org/en/v0.8.19/installing-solidity.html#smt-solvers-build) explain how to activate this option. It is activated for the Ubuntu PPA releases in most versions, but not for the Docker images, Windows binaries or the statically-built Linux binaries. It can be activated for solc-js via the [smtCallback](https://github.com/ethereum/solc-js" \l "example-usage-with-smtsolver-callback) if you have an SMT solver installed locally and run solc-js via node (not via the browser).

If you use pragma experimental SMTChecker;, then you get additional [safety warnings](https://docs.soliditylang.org/en/v0.8.19/smtchecker.html#formal-verification) which are obtained by querying an SMT solver. The component does not yet support all features of the Solidity language and likely outputs many warnings. In case it reports unsupported features, the analysis may not be fully sound.

**Importing other Source Files**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#importing-other-source-files)

**Syntax and Semantics**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#syntax-and-semantics)

Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on). However, Solidity does not support the concept of a [default export](https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export#description).

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

import "filename";

The filename part is called an *import path*. This statement imports all global symbols from “filename” (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity). This form is not recommended for use, because it unpredictably pollutes the namespace. If you add new top-level items inside “filename”, they automatically appear in all files that import like this from “filename”. It is better to import specific symbols explicitly.

The following example creates a new global symbol symbolName whose members are all the global symbols from "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";.

If there is a naming collision, you can rename symbols while importing. For example, the code below creates new global symbols alias and symbol2 which reference symbol1 and symbol2 from inside "filename", respectively.

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

**Import Paths**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#import-paths)

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. For this reason import paths do not refer directly to files in the host filesystem. Instead the compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. The import path specified in an import statement is translated into a source unit name and used to find the corresponding source unit in this database.

Using the [Standard JSON](https://docs.soliditylang.org/en/v0.8.19/using-the-compiler.html#compiler-api) API it is possible to directly provide the names and content of all the source files as a part of the compiler input. In this case source unit names are truly arbitrary. If, however, you want the compiler to automatically find and load source code into the VFS, your source unit names need to be structured in a way that makes it possible for an [import callback](https://docs.soliditylang.org/en/v0.8.19/path-resolution.html#import-callback) to locate them. When using the command-line compiler the default import callback supports only loading source code from the host filesystem, which means that your source unit names must be paths. Some environments provide custom callbacks that are more versatile. For example the [Remix IDE](https://remix.ethereum.org/) provides one that lets you [import files from HTTP, IPFS and Swarm URLs or refer directly to packages in NPM registry](https://remix-ide.readthedocs.io/en/latest/import.html).

For a complete description of the virtual filesystem and the path resolution logic used by the compiler see [Path Resolution](https://docs.soliditylang.org/en/v0.8.19/path-resolution.html#path-resolution).

**Comments**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#comments)

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

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

*/\**

*This is a*

*multi-line comment.*

*\*/*

**Note**

A single-line comment is terminated by any unicode line terminator (LF, VF, FF, CR, NEL, LS or PS) in UTF-8 encoding. The terminator is still part of the source code after the comment, so if it is not an ASCII symbol (these are NEL, LS and PS), it will lead to a parser error.

Additionally, there is another type of comment called a NatSpec comment, which is detailed in the [style guide](https://docs.soliditylang.org/en/v0.8.19/style-guide.html#style-guide-natspec). They are written with a triple slash (///) or a double asterisk block (/\*\* ... \*/) and they should be used directly above function declarations or statements.