Specification for PHP

(Initially written in 2014 by Facebook, Inc., July 2014)

Table of Contents *generated with DocToc*

Introduction

This specification is intended to provide a complete and concise definition of the syntax and semantics of the PHP language, suitable for use by the following:

- Implementers of a PHP compiler.
- Implementers of a test suite for the PHP language.
- Programmers writing PHP code.

For now, the runtime library has been excluded, as that is documented at www.php.net. As such, all forward references to library facilities have placeholders of the form (§xx).

Conformance

In this specification, "must" is to be interpreted as a requirement on an implementation or on a program; conversely, "must not" is to be interpreted as a prohibition.

If a "must" or "must not" requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this specification by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined".

The word "may" indicates "permission", and is never used to mean "might".

A *strictly conforming program* must use only those features of the language described in this specification. In particular, it must not produce output or exhibit behavior dependent on any unspecified, undefined, or implementation-defined behavior.

A *conforming implementation* must accept any strictly conforming program. A conforming implementation may have extensions, provided they do not alter the behavior of any strictly conforming program.

A *conforming program* is one that is acceptable to a conforming implementation.

A conforming implementation must be accompanied by a document that defines all implementation-defined characteristics and all extensions.

Some Syntax sections are followed by a Constraints section, which further restricts the grammar. After issuing a diagnostic for a constraint violation, a conforming implementation may continue program execution. In some cases, such continuation behavior is documented (for example, what happens when passing too few arguments to a function). Making such things constraint violations simply forces the issuance of a diagnostic; it does not require that program execution terminate.

This specification contains explanatory material—called *informative* or *non-normative* text—that, strictly speaking, is not necessary in a formal language specification. Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related clauses. Notes and Implementer Notes are provided to give advice or guidance to implementers or programmers. Informative annexes provide additional information and summarize the information contained in this specification. All text not marked as informative is *normative*.

Certain features are marked as *deprecated*. While these are normative for the current edition of this specification, they are not guaranteed to exist in future revisions. Usually, they are old approaches that have been superseded by new ones, and use of the old approach is discouraged. (Examples of this include the use of braces ({ }) for subscripting, and the use of old-style constructor names).

Terms and Definitions

For the purposes of this document, the following terms and definitions apply:

argument – an expression passed to a function, that is intended to map to a corresponding parameter.

behavior – external appearance or action.

behavior, implementation-defined – behavior specific to an implementation, where that implementation must document that behavior.

behavior, undefined – behavior on handling an erroneous program construct or data.

behavior, unspecified – behavior for which this specification provides no requirements.

constraint – restriction, either syntactic or semantic, on how language elements can be used.

error, fatal – a translation or runtime condition from which the translator or engine cannot recover.

error, fatal, catchable – a fatal error that can be caught by a user-defined handler.

error, non-fatal – an error that is not fatal.

lvalue – an expression that designates a memory location having a type.

lvalue, modifiable – an lvalue whose value can be changed.

lvalue, non-modifiable – an lvalue whose value cannot be changed.

parameter – a variable declared in the parameter list of a function that is intended to map to a corresponding argument in a call to that function.

PHP Run-Time Engine – the machinery that executes a PHP program. Referred to as *the Engine* throughout this specification.

value – precise meaning of the contents of a memory location when interpreted as having a specific type.

Other terms are defined throughout this specification, as needed, with the first usage being typeset *like this*.

Basic Concepts

Program Structure

A PHP *program* consists of one or more source files, known formally as *scripts*.

```
script:
 script-section
 script   script-section

script-section:
   text_opt <?php statement-list_opt ?>_opt text_opt

text:
```

```
arbitrary text not containing the sequence <?php
```

All of the sections in a script are treated as though they belonged to one continuous section, except that any intervening text is treated as though it were a string literal given to the intrinsic `echo` (§§).

A script can import another script via a script inclusion operator (§§).

statement-list is defined in §§.

The top level of a script is simply referred to as the *top level*.

Program Start-Up

A program begins execution at the start of a script (§§) designated in some unspecified manner. This script is called the *start-up script*.

Once a program is executing, it has access to certain environmental information (§§), as follows:

- The number of *command-line arguments*, via the predefined variable `$argc`.
- A series of one or more command-line arguments as strings, via the predefined variable `$argv`.
- A series of *environment variable* names and their definitions.

When a top level (§§) is the main entry point for a script, it gets the global variable environment. When a top level is invoked via `include/require` (§§), it inherits the variable environment of its caller. Thus, when looking at one top level in isolation, it's not possible to tell statically whether it will have the global variable environment or some local variable environment. It depends on how the pseudo-main is invoked and it depends on the runtime state of the program when it's invoked.

Program Termination

A program may terminate normally in the following ways:

- Execution reaches the end of the start-up script (§§).
- A `return` statement (§§) in the start-up script is executed.
- The intrinsic `exit` (§§) is called explicitly.

The behavior of the first two cases is equivalent to corresponding calls to exit.

A program may terminate abnormally under various circumstances, such as the detection of an uncaught exception, or the lack of memory or other critical resource. If execution reaches the end of the start-up script via a fatal error, or via an uncaught exception and there is no uncaught exception handler registered by `set_exception_handler`, that is equivalent to `exit(255)`. If execution reaches the end of the start-up script via an uncaught exception and an uncaught exception handler was registered by `set_exception_handler`, that is equivalent to exit(0). It is unspecified whether object destructors (§§) are run. In all other cases, the behavior is unspecified.

The Memory Model

General

This subclause and those immediately following it describe the abstract memory model used by PHP for storing variables. A conforming implementation may use whatever approach is desired as long as from any testable viewpoint it appears to behave as if it follows the abstract model. The abstract model makes no explicit or implied restrictions or claims about performance, memory consumption, and machine resource usage.

The abstract model presented here defines three kinds of abstract memory locations:

- A *value storage location* (VStore) is used to represent a program value, and is created by the Engine as needed. A VStore can contain a scalar value such as an integer or a Boolean, or it can contain a handle pointing to an HStore (see below).
- A *variable slot* (VSlot) is used to represent a variable named by the programmer in the source code, such as a local variable, an array element, an instance property of an object, or a static property of a class. A VSlot comes into being based on explicit usage of a variable in the source code. A VSlot contains a pointer to a VStore.
- A *heap storage location* (HStore) is used to represent the contents of any non-scalar value, and is created by the Engine as needed.

Each existing variable has its own VSlot, which at any time contains a pointer to a VStore. A VSlot cannot contain a null pointer. A VSlot can be changed to point to different VStores over time. Multiple VSlots may simultaneously point to the same VStore. When a new VSlot is created, a new VStore is also created and the VSlot is initially set to point to the new VStore.

A VStore can be changed to contain different scalar values and handles over time. Multiple VStores may simultaneously contain handles that point to the same HStore. When a VStore is created it initially contains the scalar value NULL unless specified otherwise. In addition to containing a value, VStores also carry a *type tag* that indicates the type (§§) of the VStore's value. A VStore's type tag can be changed over time. At any given time a VStore's type tag may be one of the following: `Null`, `Bool`, `Int`, `Float`, `Str`, `Arr`, `Arr-D` (see §§), `Obj`, or `Res`.

An HStore represents the contents of a non-scalar value, and it may contain zero or more VSlots. At run time, the Engine may add new VSlots and it may remove and destroy existing VSlots as needed to support adding/removing array elements (for arrays) and to support adding/removing instance properties (for objects). HStores that represent the contents of arrays and objects have some unspecified way to identify and retrieve a contained VSlot using a dictionary scheme (such as having values with integer keys or case-sensitive string keys). Whether an HStore is a fixed-size during its whole lifetime or whether it can change size, is unspecified. Whether it allocates auxiliary chunks of memory or not, is unspecified. Whether it organizes it's contained VSlots in a linked list or some other manner is unspecified.

An HStore's VSlots (i.e., the VSlots contained within the HStore) point to VStores, and each VStore contains a scalar value or a handle to an HStore, and so on through arbitrary levels, allowing arbitrarily complex data structures to be represented. For example, a singly linked list might consist of a variable called `$root`, which is represented by a VSlot pointing to a VStore containing a handle to the first node. Each node is represented by an HStore that contains the data for that node in one or more VSlots, as well as a VSlot pointing to VStore containing a handle to the next node. Similarly, a binary tree might consist of a variable called `$root`, which is represented by a VSlot pointing to a VStore containing a handle to the root node. Each node is represented by an HStore that contains the data for that node in one or more VSlots, as well as a pair of VSlots pointing to VStores containing the handles to the left and right branch nodes. The leaves of the tree would be VStores or HStores, as needed.

VSlots cannot contain pointers to VSlots or handles to HStores. VStores cannot contain pointers to VSlots or VStores. HStores cannot directly contain any pointers or handles to any abstract memory location; HStores can only directly contain VSlots.

Here is an example demonstrating one possible arrangement of VSlots, VStores, and HStores:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                                        |            |
                                                        V            V
                                                [VStore Int 1]  [VStore Int 3]
```

In this picture the VSlot in the upper left corner represents the variable `$a`, and it points to a VStore that represents `$a`'s current value. This VStore contains a handle to an HStore which represents the contents of an object of type Point with two instance properties `$x` and `$y`. This HStore contains two VSlots representing instance properties `$x` and `$y`, and each of these VSlots points to a distinct VStore which contains an integer value.

Implementation Notes: php.net's implementation can be mapped roughly onto the abstract memory model as follows: `zval pointer => VSlot, zval => VStore, HashTable => HStore`, and `zend_object/zend_object_handlers => HStore`. Note, however, that the abstract memory model is not intended to exactly match the php.net implementation's model, and for generality and simplicity there are some superficial differences between the two models.

For most operations, the mapping between VSlots and VStores remains the same. Only the following program constructs can change a VSlot to point to different VStore, all of which are *byRef-aware* operations and all of which (except unset) use the & punctuator:

- byRef assignment (§§).
- byRef parameter declaration (§§).
- byRef function return (§§, §§).
- byRef value in a foreach statement (§§).
- byRef initializer for an array element (§§).
- byRef variable-use list in an anonymous function (§§).
- unset (§§).

Reclamation and Automatic Memory Management

The Engine is required to manage the lifetimes of VStores and HStores using some form of automatic memory management.

When dealing with VStores and HStores, the Engine is required to implement some form of automatic memory management. When a VStore or HStore is created, memory is allocated for it, and for an HStore that represents an object (§§), its constructor (§§) is invoked.

Later, if a VStore or HStore becomes unreachable through any existing variable, they become eligible for reclamation to release the memory they occupy. The engine may reclaim a VStore or HStore at any time between when it becomes eligible for reclamation and when the script exits. Before reclaiming an HStore that represents an object (§§), the Engine will invoke the object's destructor (§§) if one is defined.

The Engine must reclaim each VSlot when the storage duration (§§) of its corresponding variable ends, when the variable is explicitly unset by the programmer, or when the script exits, whichever comes first. In the case where a VSlot is contained within an HStore (i.e. an array element or an object instance property), the engine must immediate reclaim the VSlot when it is explicitly unset by the programmer, when the containing HStore is reclaimed, or when the script exits, whichever comes first.

The precise form of automatic memory management used by the Engine is unspecified, which means that the time and order of the reclamation of VStores and HStores is unspecified.

A VStore's refcount is defined as the number of unreclaimed VSlots that point to the VStore. Because the precise form of automatic memory management is not specified, a VStore's refcount at a given time may differ between conforming implementations due to VSlots, VStores, and HStores being reclaimed at different times. Despite the use of the term refcount, conforming implementations are not required to use a reference counting-based implementation for automatic memory management.

(dead): In some pictures, storage-location boxes are shown as (dead). For a VStore or an HStore this indicates that the VStore or HStore is no longer reachable through any variable and is eligible for reclamation. For a VSlot, this indicates that the VSlot has been reclaimed or, in the case of a VSlot contained with an HStore, that the containing HStore has been reclaimed or is eligible for reclamation.

Assignment

General

This subclause and those immediately following it describe the abstract model's implementation of *value assignment* and *byRef assignment*. Value assignment of non-array types to local variables is described first, followed by byRef assignment with local variables, followed by value assignment of array types to local variables, and ending with value assignment with complex left-hand side expressions, and byRef assignment with complex expressions on the left- or right-hand side.

Value assignment and byRef assignment are core to the PHP language, and many other operations in this specification are described in terms of value assignment and byRef assignment.

Value Assignment of Scalar Types to a Local Variable

Value assignment is the primary means by which the programmer can create local variables. If a local variable appears on the left-hand side of value assignment does not exist, the engine will bring a new local variable into existence and create a VSlot and initial VStore for storing the local variable's value.

Consider the following example of value assignment (§§) of scalar values to local variables:

```
$a = 123;

$b = false;
```

```
[VSlot $a *]-->[VStore Int 123]

[VSlot $b *]-->[VStore Bool false]
```

Variable `$a` comes into existence and is represented by a newly created VSlot pointing to a newly created VStore. Then the integer value 123 is written to the VStore. Next, `$b` comes into existence represented by a VSlot and corresponding VStore, and the Boolean value false is written to the VStore.

Next consider the value assignment `$b = $a`:

```
[VSlot $a *]-->[VStore Int 123]

[VSlot $b *]-->[VStore Int 123 (Bool false was overwritten)]
```

The integer value 123 is read from `$a`'s VStore and is written into `$b`'s VStore, overwriting its previous contents. As we can see, the two variables are completely self-contained; each has its own VStore containing a separate copy of the integer value 123. Value assignment reads the contents of one VStore and overwrites the contents of the other VStore, but the relationship of VSlots to VStores remains unchanged. Changing the value of `$b` has no effect on `$a`, and vice versa.

Using literals or arbitrarily complex expressions on the right hand side of value assignment value works the same as it does for variables, except that the literals or expressions don't have their own VSlots or VStores. The scalar value or handle produced by the literal or expression is written into the VStore of the left hand side, overwriting its previous contents.

Implementation Notes: For simplicity, the abstract model's definition of value assignment never changes the mapping from VSlots to VStores. This aspect of the abstract model is superficially different from the php.net implementation's model, which in some cases will set two variable slots to point to the same zval when performing value assignment. Despite this superficial difference, php.net's implementation produces the same observable behavior as the abstract model presented here.

To illustrate the semantics of value assignment further, consider `++$b`:

```
[VSlot $a *]-->[VStore Int 123]

[VSlot $b *]-->[VStore Int 124 (123 was overwritten)]
```

Now consider `$a = 99`:

```
[VSlot $a *]-->[VStore Int 99 (123 was overwritten)]

[VSlot $b *]-->[VStore Int 124]
```

In both of these examples, one variable's value is changed without affecting the other variable's value. While the above examples only demonstrate value assignment for integer and Boolean values, the same mechanics apply for all scalar types.

Note that strings are also considered scalar values for the purposes of the abstract memory model. Unlike non-scalar types which are represented using a VStore pointing to an HStore containing the non-scalar value's contents, the abstract model assumes that a string's entire contents (i.e., the string's characters and its length) can be stored in a VStore and that value assignment for a string eagerly copies a string's entire contents to the VStore being written to. Consider the following example:

```
$a = 'gg';

$b = $a;
```

```
[VSlot $a *]-->[VStore Str 'gg']

[VSlot $b *]-->[VStore Str 'gg']
```

`$a`'s string value and `$b`'s string values are distinct from each other, and mutating `$a`'s string will not affect `$b`. Consider `++$b`, for example:

```
[VSlot $a *]-->[VStore Str 'gg']

[VSlot $b *]-->[VStore Str 'gh']
```

Implementation Notes: For simplicity, the abstract model represents a string as a scalar value that can be entirely contained within VStore. This aspect of the abstract model is superficially different from the php.net implementation's model, where a zval points to a separate buffer in memory containing a string's characters and in the common case multiple slots point to the same zval that holds the string. Despite this superficial difference, php.net's implementation produces the same observable behavior (excluding performance and resource consumption) as the abstract model presented here.

Because a string's content can be arbitrarily large, copying a string's entire contents for value assignment can be expensive. In practice an application written in PHP may rely on value assignment of strings being relatively inexpensive (in order to deliver acceptable performance), and as such it is common for an implementation to use a deferred copy mechanism to reduce the cost of value assignment for strings. Deferred copy mechanisms work by not copying a string during value assignment and instead allowing multiple variables to share the string's contents indefinitely until a mutating operation (such as the increment operator) is about to be executed on the string, at which time some or all of the string's contents are copied. A conforming implementation may choose to defer copying a string's contents for value assignment so long as it has no observable effect on behavior from any testable viewpoint (excluding performance and resource consumption).

Value Assignment of Object and Resource Types to a Local Variable

To demonstrate value assignment of objects to local variables, consider the case in which we have a Point class that supports a two-dimensional Cartesian system. An instance of Point contains two instance properties, `$x` and `$y`, that store the x- and y-coordinates, respectively. A constructor call (§§) of the form `Point(x, y)` used with operator `new` (§§) creates a new point at the given location, and a method call of the form `move(newX, newY)` moves a `Point` to the new location.

With the `Point` class, let us consider the value assignment `$a = new Point(1, 3)`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                                          |            |
                                                          V            V
                                                  [VStore Int 1]  [VStore Int 3]
```

Variable `$a` is given its own VSlot, which points to a VStore that contains a handle pointing to an HStore allocated by `new`

(§§) and that is initialized by `Point`'s constructor.

Now consider the value assignment `$b = $a`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                 ^                     |           |
                                 |                     V           V
[VSlot $b *]-->[VStore Obj *]-----+             [VStore Int 1] [VStore Int 3]
```

`$b`'s VStore contains a handle that points to the same object as does `$a`'s VStore's handle. Note that the Point object itself was not copied, and note that `$a`'s and `$b`'s VSlots point to distinct VStores.

Let's modify the value of the Point whose handle is stored in `$b` using `$b->move(4, 6)`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                 ^                     |           |
                                 |                     V           V
[VSlot $b *]-->[VStore Obj *]-----+         [VStore Int 4] [VStore Int 6]
                                        (1 was overwritten) (3 was overwritten)
```

As we can see, changing `$b`'s Point changes `$a`'s as well.

Now, let's make `$a` point to a different object using `$a = new Point(2, 1)`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                                       |           |
[VSlot $b *]-->[VStore Obj *]-----+                    V           V
                                 |             [VStore Int 2] [VStore Int 1]
                                 V
                               [HStore Point [VSlot $x *] [VSlot $y *]]
                                                       |           |
                                                       V           V
                                               [VStore Int 4] [VStore Int 6]
```

Before `$a` can take on the handle of the new `Point`, its handle to the old `Point` must be removed, which leaves the handles of `$a` and `$b` pointing to different Points.

We can remove all these handles using `$a = NULL` and `$b = NULL`:

```
[VSlot $a *]-->[VStore Null]    [HStore Point [VSlot $x *] [VSlot $y *] (dead)]
                                                       |           |
[VSlot $b *]-->[VStore Null]    [VStore Int 2 (dead)]<--+           V
                                                         [VStore Int 1 (dead)]

                               [HStore Point [VSlot $x *] [VSlot $y *] (dead)]
                                                       |           |
                               [VStore Int 4 (dead)]<--+           V
                                                       [VStore Int 6 (dead)]
```

By assigning null to `$a`, we remove the only handle to `Point(2,1)`, which allows that object's destructor (§§) to run. A similar thing happens with `$b`, as it too is the only handle to its Point.

Although the examples above only show with only two instance properties, the same mechanics apply for value assignment of all object types, even though they can have an arbitrarily large number of instance properties of arbitrary type. Likewise, the same mechanics apply to value assignment of all resource types.

ByRef Assignment for Scalar Types with Local Variables

Let's begin with the same value assignment (§§) as in the previous subclause, `$a = 123` and `$b = false`:

```
[VSlot $a *]-->[VStore Int 123]

[VSlot $b *]-->[VStore Bool false]
```

Now consider the byRef assignment (§§) `$b =& $a`, which has byRef semantics:

```
[VSlot $a *]-->[VStore Int 123]
                 ^
                 |
[VSlot $b *]-----+     [VStore Bool false (dead)]
```

In this example, byRef assignment changes `$b`'s VSlot point to the same VStore that `$a`'s VSlot points to. The old VStore that `$b`'s VSlot used to point to is now unreachable. As stated in §§, it is not possible for a VSlot to point to another VSlot, so `$b`'s VSlot cannot point to `$a`'s VSlot. When multiple variables' VSlots point to the same VStore, the variables are said to be *aliases* of each other or they are said to have an *alias relationship*. In the example above, after the byRef assignment executes the variables `$a` and `$b` will be aliases of each other.

Now, let's observe what happens when we change the value of `$b` using `++$b`:

```
[VSlot $a *]-->[VStore Int 124 (123 was overwritten)]
                 ^
                 |
[VSlot $b *]-----+
```

`$b`'s value, which is stored in the VStore that `$b`'s VSlot points, is changed to 124. And as that VStore is also aliased by `$a`'s VSlot, the value of `$a` is also 124. Indeed, any variable's VSlot that is aliased to that VStore will have the value 124.

Now consider the value assignment `$a = 99`:

```
[VSlot $a *]-->[VStore Int 99 (124 was overwritten)]
                 ^
                 |
[VSlot $b *]-----+
```

The alias relationship between `$a` and `$b` can be broken explicitly by using `unset` on variable `$a` or variable `$b`. For example, consider `unset($a)`:

```
[VSlot $a (dead)]      [VStore Int 99]
                         ^
                         |
[VSlot $b *]-------------+
```

Unsetting `$a` causes variable `$a` to be destroyed and its corresponding alias to the VStore to be removed, leaving `$c`'s VSlot as the only pointer remaining to the VStore.

Other operations can also break an alias relationship between two or more variables. For example, `$a = 123` and `$b =& $a`, and `$c = 'hi'`:

```
[VSlot $a *]-->[VStore Int 123]
                 ^
                 |
[VSlot $b *]-----+
```

```
[VSlot $c *]-->[VStore Str 'hi']
```

After the byRef assignment, `$a` and `$b` now have an alias relationship. Next, let's observe what happens for `$b = &$c`:

```
[VSlot $a *]-->[VStore Int 123]

[VSlot $b *]-----+
                 |
                 V
[VSlot $c *]-->[VStore Str 'hi']
```

As we can see, the byRef assignment above breaks the alias relationship between `$a` and `$b`, and now `$b` and `$c` are aliases of each other. When byRef assignment changes a VSlot to point to a different VStore, it breaks any existing alias relationship the left hand side variable had before the assignment operation.

It is also possible to use byRef assignment to make three or more VSlots point to the same VStore. Consider the following example:

```
$b =& $a;
$c =& $b;
$a = 123;
```

```
[VSlot $a *]-->[VStore Int 123]
                 ^   ^
                 |   |
[VSlot $b *]-----+   |
                     |
[VSlot $c *]---------+
```

Like value assignment, byRef assignment provides a means for the programmer to created variables. If the local variables that appear on the left- or right-hand side of byRef assignment do not exist, the engine will bring new local variables into existence and create a VSlot and initial VStore for storing the local variable's value.

Note that literals, constants, and other expressions that don't designate a modifiable lvalue cannot be used on the left- or right-hand side of byRef assignment.

Byref Assignment of Non-Scalar Types with Local Variables

byRef assignment of non-scalar types works using the same mechanism as byRef assignment for scalar types. Nevertheless, it is worthwhile to describe a few examples to clarify the semantics of byRef assignment. Recall the example from §§) using the `Point` class:

`$a = new Point(1, 3);`

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]
                                                    |            |
                                                    V            V
                                            [VStore Int 1]  [VStore Int 3]
```

Now consider the byRef assignment (§§) `$b =& $a`, which has byRef semantics:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *][VSlot $y *]]
                 ^                                  |           |
                 |                                  V           V
[VSlot $b *]-----+                          [VStore Int 1] [VStore Int 3]
```

`$a` and `$b` now aliases of each other. Note that byRef assignment produces a different result than `$b =` $a where `$a` and `$b` would point to distinct VStores pointing to the same HStore.

Let's modify the value of the `Point` aliased by `$a` using `$a->move(4, 6)`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *] VSlot $y *]]
                 ^                                       |           |
                 |                                       V           V
[VSlot $b *]-----+                           [VStore Int 4] [VStore Int 6]
                                        (1 was overwritten) (3 was overwritten)
```

Now, let's change `$a` itself using the value assignment `$a = new Point(2, 1)`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Point [VSlot $x *][VSlot $y *]]
                 ^                                       |           |
                 |                                       V           V
[VSlot $b *]-----+                             [VStore Int 2] [VStore Int 1]

                               [HStore Point [VSlot $x *]   [VSlot $y *] (dead)]
                                                        |             |
                                                        V             V
                                     [VStore Int 4 (dead)] [VStore Int 6 (dead)]
```

As we can see, `$b` continues to have an alias relationship with `$a`. Here's what's involved in that assignment: `$a` and `$b`'s VStore's handle pointing to `Point(4,6)` is removed, `Point(2,1)` is created, and `$a` and `$b`'s VStore is overwritten to contain a handle pointing to that new `Point`. As there are now no VStores pointing to `Point(4,6)`, its destructor (§§) can run.

We can remove these aliases using `unset($a, $b)`:

```
[VSlot $a (dead)]       [HStore Point [VSlot $x *] [VSlot $y *] (dead)]
                                               |            |
                                               V            V
[VSlot $b (dead)]             [VStore Int 2 (dead)]  [VStore Int 1 (dead)]
```

Once all the aliases to the VStores are gone, the VStores can be destroyed, in which case, there are no more pointers to the HStore, and its destructor ([§§]estructors](#Destructors)) can be run.

Value Assignment of Array Types to Local Variables

The semantics of value assignment of array types is different from value assignment of other types. Recall the `Point` class from the examples in §§, and consider the following value assignments (§§) and their abstract implementation:

`$a = array(10, 'B' => new Point(1, 3));`

```
[VSlot $a *]-->[VStore Arr *]-->[HStore Array [VSlot 0 *] [VSlot 'B' *]]
                                                        |             |
                                                        V             V
                                              [VStore Int 10]   [VStore Obj *]
                                                                            |
                                [HStore Point [VSlot $x *] [VSlot $y *]]<--+
                                                         |            |
                                                         V            V
                                             [VStore Int 1]  [VStore Int 3]
```

In the example above, `$a`'s VStore is initialized to contain a handle to an HStore for an array containing two elements, where one element is an integer and the other is a handle to an HStore for an object.

Now consider the following value assignment `$b = $a`. A conforming implementation must implement value assignment of arrays in one of the following ways: (1) eager copying, where the implementation makes a copy of `$a`'s array during value assignment and changes `$b`'s VSlot to point to the copy; or (2) deferred copying, where the implementation uses a deferred copy mechanism that meets certain requirements. This subclause describes eager copying, and the subclause that immediately follows (§§) describes deferred copying.

To describe the semantics of eager copying, let's begin by considering the value assignment `$b = $a`:

```
[VSlot $a *]-->[VStore Arr *]-->[HStore Array [VSlot 0 *] [VSlot 'B' *]]
                                                     |             |
[VSlot $b *]-->[VStore Arr *]                        V             V
                          |                   [VStore Int 10]  [VStore Obj *]
                          V                                                |
[HStore Array [VSlot 0 *] [VSlot 'B' *]]                                   |
                      |             |                                      |
            +---------+   +---------+                                      |
            V             V                                                |
[VStore Int 10] [VStore Obj *]-->[HStore Point [VSlot $x *] [VSlot $y *]]<---+
                                                       |            |
                                                       V            V
                                               [VStore Int 1]  [VStore Int 3]
```

The value assignment `$b = $a` made a copy of `$a`'s array. Note how `$b`'s VSlot points to a different VStore than `$a`'s VSlot, and `$b`'s VStore points to a different HStore than `$a`'s VStore. Each source array element is copied using *member-copy assignment* `=*`, which is defined as follows:

```
   $destination =* $source
```

- If `$source`'s VStore has a refcount equal to 1, the Engine copies the array element using value assignment (`destination = $source`).
- If `$source`'s VStore has a refcount that is greater than 1, the Engine uses an implementation-defined algorithm to decide whether to copy the element using value assignment (`$destination = $source`) or byRef assignment (`$destination =& $source`).

Note the member-copy assignment `=*` is **not** an operator or language construct in the PHP language, but instead it is used internally to describe behavior for the engine for array copying and other operations

For the particular example above, member-copy assignment exhibits the same semantics as value assignment for all conforming implementations because all of the array elements' VStores have a refcount equal to 1. The first element VSlots in `$a`'s array and `$b`'s array point to distinct VStores, each of which contain a distinct copy of the integer value 10. The second element VSlots in `$a`'s array and `$b`'s array point to distinct VStores, each of which contain a handle to the same object HStore.

Let's consider another example:

```
$x = 123;
$a = array(array(&$x, 'hi'));
$b = $a;
```

Eager copying can produce two possible outcomes depending on the implementation. Here is the first possible outcome:

```
[VSlot $a *]---->[VStore Arr *]---->[HStore Array [VSlot 0 *]]
                                                           |
[VSlot $x *]-------------------------+   [VStore Arr *]<---+
                                     |               |
[VSlot $b *]-->[VStore Arr *]        |               V
                          |          |  [HStore Array [VSlot 0 *][VSlot 1 *]]
                          V          |                          |          |
```

```
         [HStore Array [VSlot 0 *]]  |                          V          |
                                |    +---------------->[VStore Int 123]    |
                                V                          ^               V
                     [VStore Arr *]                        |   [VStore Str 'hi']
                                 |          +--------------+
                                 V          |
                     [HStore Array [VSlot 0 *] [VSlot 1 *]]
                                                        |
                                                        V
                                                     [VStore Str 'hi']
```

Here is the second possible outcome:

```
[VSlot $a *]---->[VStore Arr *]---->[HStore Array [VSlot 0 *]]
                                                           |
[VSlot $x *]-------------------------+  [VStore Arr *]<----+
                                     |               |
[VSlot $b *]-->[VStore Arr *]        |               V
                           |         |  [HStore Array [VSlot 0 *] [VSlot 1 *]]
                           V         |                         |           |
         [HStore Array [VSlot 0 *]]  |                         V           |
                                |    +---------------->[VStore Int 123]    |
                                V                                          V
                     [VStore Arr *]                            [VStore Str 'hi']
                                 |
                                 V
                    [HStore Array [VSlot 0 *] [VSlot 1 *]]
                                           |           |
                                           V           V
                                  [VStore Int 123]  [VStore Str 'hi']
```

In both possible outcomes, value assignment with eager copying makes a copy of `$a`'s array, copying the array's single element using member-copy assignment (which in this case will exhibit the same semantics of value assignment for all implementations), which in turn makes a copy of the inner array inside `$a`'s array, copying the inner array's elements using member-copy assignment. The inner array's first element VSlot points to a VStore that has a refcount that is greater than 1, so an implementation-defined algorithm is used to decide whether to use value assignment or byRef assignment. The first possible outcome shown above demonstrates what happens if the implementation chooses to do byRef assignment, and the second possible outcome shown above demonstrates what happens if the implementation chooses to do value assignment. The inner array's second element VSlot points to a VStore that has a refcount equal to 1, so value assignment is used to copy the inner array's second element for all conforming implementations that use eager copying.

Although the examples in this subclause only use arrays with one element or two elements, the model works equally well for all arrays even though they can have an arbitrarily large number of elements. As to how an HStore accommodates all of them, is unspecified and unimportant to the abstract model.

Deferred Array Copying

As mentioned in the previous subclause (§§), an implementation may choose to use a deferred copy mechanism instead of eagerly making a copy for value assignment of arrays. An implementation may use any deferred copy mechanism desired so long as it conforms to the abstract model's description of deferred array copy mechanisms presented in this subclause.

Because an array's contents can be arbitrarily large, eagerly copying an array's entire contents for value assignment can be expensive. In practice an application written in PHP may rely on value assignment of arrays being relatively inexpensive for the common case (in order to deliver acceptable performance), and as such it is common for an implementation to use a deferred array copy mechanism in order to reduce the cost of value assignment for arrays.

Unlike conforming deferred string copy mechanisms discussed in §§ that must produce the same observable behavior as eager string copying, deferred array copy mechanisms are allowed in some cases to exhibit observably different behavior than eager array copying. Thus, for completeness this subclause describes how deferred array copies can be modeled in the abstract memory model and how conforming deferred array copy mechanisms must behave.

Conforming deferred array copy mechanisms work by not making an array copy during value assignment, by allowing the destination VStore to share an array HStore with the source VStore, and by making a copy of the array HStore at a later time if or when it is necessary. The abstract model represents a deferred array copy relationship by marking the destination VStore with a special “Arr-D” type tag and by sharing the same array HStore between the source and destination VStores. Note that the source VStore's type tag remains unchanged. For the purposes of this abstract model, the “Arr-D” type tag is considered identical to the “Arr” type in all respects except when specified otherwise.

To illustrate this, let's see how the previous example would be represented under the abstract model assuming the implementation defers the copying the array:

```
$x = 123;
$a = array(array(&$x, 'hi'));
$b = $a;
```

```
[VSlot $a *]--->[VStore Arr *]--->[HStore Array [VSlot 0 *]]
                                    ^                    |
                                    |   [VStore Arr *]<--+
[VSlot $b *]--->[VStore Arr-D *]----+               |
                                                    V
                                        [HStore Array [VSlot 0 *] [VSlot 1 *]]
                                                               |           |
                                                               V           |
[VSlot $x *]------------------------------------------->[VStore Int 123]    |
                                                                           V
                                                               [VStore Str 'hi']
```

As we can see, both `$a`'s VStore (the source VStore) and `$b`'s VStore (the destination VStore) point to the same array HStore. Note the asymmetric nature of how deferred array copies are represented in the abstract model. In the above example the source VStore's type tag remains unchanged after value assignment, whereas the destination VStore's type tag was changed to “Arr-D”.

When the engine is about to perform an array-mutating operation on a VStore tagged “Arr” that participates in a deferred array copy relationship or on a VStore tagged “Arr-D”, the engine must first take certain actions that involve making a copy of the array (described in the next paragraph) before performing the array-mutating operation. An array-mutating operation is any operation can add or remove array elements, overwrite existing array elements, change the state of the array's internal cursor, or cause the refcount of one or more of the array's element VStores or subelement VStores to increase from 1 to a value greater than 1. This requirement to take certain actions before performing an array-mutation operation on a VStore participating in a deferred array copy relationship is commonly referred to as the copy-on-write requirement.

When an array-mutating operation is about to be performed on a given VStore X with an “Arr” type tag that participates in a deferred array copy relationship, the engine must find all of the VStores tagged “Arr-D” that point to the same array HStore that VStore X points to, make a copy of the array (using member-copy assignment to copy the array's elements as described in §§), and update all of these VStores tagged “Arr-D” to point to the newly created copy (note that VStore X remains unchanged). When an array-mutation operation is about to be performed on a given VStore X with an “Arr-D” type tag, the engine must make a copy of the array (as described in §§), update VStore X to point to the newly created copy, and change VStore X's type tag to “Arr”. These specific actions that the engine must perform on VStore at certain times to satisfy the copy-on-write requirement are collectively referred to as “array-separation” or “array-separating the VStore”. An array-mutation operation is said to “trigger” an array-separation.

Note that for any VStore with an “Arr” type tag that participates in a deferred array copy relationship, or for any VStore with an “Arr-D” type tag, a conforming implementation may choose to array-separate the VStore at any time for any reason as long as the copy-on-write requirement is upheld.

Continuing with the previous example, consider the array-mutating operation `$b[1]++`. Depending on the implementation, this can produce one of three possible outcomes. Here is the one of the possible outcomes:

```
[VSlot $a *]---->[VStore Arr *]---->[HStore Array [VSlot 0 *]]
                                                           |
[VSlot $b *]-->[VStore Arr *]            [VStore Arr *]<---+
                             |                        |
      +----------------------+              +---------+
      V                                     V
  [HStore Array [VSlot 0 *] [VSlot 1 *]]  [HStore Array [VSlot 0 *] [VSlot 1 *]]
                          |           |      ^                   |          |
                          |           V      |                   V          |
                          |  [VStore Int 1]  |             [VStore Int 123] |
                          V                  |             ^                V
                        [VStore Arr-D *]-----+             |   [VStore Str 'hi']
                                                           |
 [VSlot $x *]----------------------------------------------+
```

As we can see in the outcome shown above, `$b`'s VStore was array-separated and now `$a`'s VStore and `$b`'s VStore point to distinct array HStores. Performing array-separation on `$b`'s VStore was necessary to satisfy the copy-on-write requirement. `$a`'s array remains unchanged and that `$x` and `$a[0][0]` still have an alias relationship with each other. For this particular example, conforming implementations are required to preserve `$a`'s array's contents and to preserve the alias relationship between `$x` and `$a[0][0]`. Finally, note that `$a[0]` and `$b[0]` have a deferred copy relationship with each other in the outcome shown above. For this particular example, a conforming implementation is not required to array-separate `$b[0]`'s VStore, and the outcome shown above demonstrates what happens when `$b[0]`'s VStore is not array-separated. However, an implementation can choose to array-separate `$b[0]`'s VStore at any time if desired. The other two possible outcomes shown below demonstrate what can possibly happen if the implementation choose to array-separate `$b[0]`'s VStore as well. Here is the second possible outcome:

```
[VSlot $a *]---->[VStore Arr *]---->[HStore Array [VSlot 0 *]]
                                                           |
[VSlot $b *]-->[VStore Arr *]            [VStore Arr *]<---+
                          |                           |
                          V                           V
  [HStore Array [VSlot 0 *] [VSlot 1 *]]  [HStore Array [VSlot 0 *] [VSlot 1 *]]
                          |           |                           |          |
       +------------------+           V                           |          |
       |                     [VStore Int 1]                 +----+           |
       V                                                    |                V
  [VStore Arr-D *]-->[HStore Array [VSlot 0 *] [VSlot 1 *]] | [VStore Str 'hi']
                                            |           |   |
                                    +-------+           |   |
                                    |                   V   |
                                    |    [VStore Str 'hi']  |
                                    V                       |
 [VSlot $x *]--------------------->[VStore Int 123]<--------+
```

Here is the third possible outcome:

```
[VSlot $a *]---->[VStore Arr *-]---->[HStore Array [VSlot 0 *]]
                                                            |
[VSlot $b *]-->[VStore Arr *]             [VStore Arr *]<---+
                            |                          |
                            V                          V
 [HStore Array [VSlot 0 *] [VSlot 1 *]]  [HStore Array [VSlot 0 *] [VSlot 1 *]]
                         |           |                           |           |
       +------------------+          V                           |           |
       |                     [VStore Int 1]                  +----+          |
       V                                                     |               V
   [VStore Arr-D *]-->[HStore Array [VSlot 0 *] [VSlot 1 *]] | [VStore Str 'hi']
                                             |           |   |
```

```
                      [VStore Int 123]<-------+           |   |
                                                          V   |
                                           [VStore Str 'hi']  |
                                                              |
  [VSlot $x *]--------------------->[VStore Int 123]<--------+
```

The second and third possible outcomes show what can possibly happen if the implementation chooses to array-separate `$b[0]`'s VStore. In the second outcome, `$b[0][0]` has an alias relationship with `$x` and `$a[0][0]`. In the third outcome, `$b[0][0]` does not have an alias relationship, though `$x` and `$a[0][0]` still have an alias relationship with each other. The differences between the second and third outcome are reflect that different possibilities when the engine uses member-copy assignment to copy `$a[0]`'s arrays's elements into `$b[0]`'s array.

Finally, let's briefly consider one more example:

```
$x = 0;
$a = array(&$x);
$b = $a;
$x = 2;
unset($x);
$b[1]++;
$b[0]++;
echo $a[0], ' ', $b[0];
```

For the example above, a conforming implementation could output “2 1”, “2 3”, or “3 3” depending on how it implements value assignment for arrays.

For portability, it is generally recommended that programs written in PHP should avoid performing value assignment with a right-hand side that is an array with one or more elements or sub-elements that have an alias relationship.

Implementation Notes: For generality and for simplicity, the abstract model represents deferred array copy mechanisms in a manner that is more open-ended and superficially different than the php.net implementation's model, which uses a symmetric deferred copy mechanism where a single zval contains the sole pointer to a given Hashtable and deferred array copies are represented as multiple slots pointing to the same single zval that holds the array. Despite this superficial difference, php.net's implementation produces behavior that is compatible with the abstract model's definition of deferred array copy mechanisms.

General Value Assignment

The subclauses above thus far have described the mechanics of value assignment to a local variable. This subclause describes how value assignment works when general modifiable lvalue expressions are used on the left hand side.

[TODO: Add description and examples here involving array elements and object instance properties. Describe how new array elements and object instance properties can be created via value assignment.]

General ByRef Assignment

The subclauses above thus far have described the mechanics of byref assignment with local variables. This subclause describes how byref assignment works when general modifiable lvalue expressions are used on the left hand side and/or the right hand side.

[TODO: Add description and examples here involving array elements and object instance properties. Describe how new array elements and object instance properties can be created via byref assignment.]

Argument Passing

Argument passing is defined in terms of simple assignment (§§, §§, §§, and §§) or byRef assignment ([§§]), §§, and §§), depending on how the parameter is declared. That is, passing an argument to a function having a corresponding parameter is like assigning that argument to that parameter. The function-call situations involving missing arguments or undefined-variable arguments are discussed in (§§).

Value Returning

Returning a value from a function is defined in terms of simple assignment (§§, §§, §§, and §§) or byRef assignment (§§, §§, and §§) depending on how the function is declared. That is, returning a value from a function to its caller is like assigning that value to the user of the caller's return value. The function-return situations involving a missing return value are discussed in (§§).

Cloning objects

When an instance is allocated, operator `new` (§§) returns a handle that points to that object. As described in §§), value assignment of a handle to an object does not copy the object HStore itself. Instead, it creates a copy of the handle. How then to make a copy of the object itself? Our only access to it is via the handle. The PHP language allows us to do this via operator `clone` (§§).

To demonstrate how the `clone` operator works, consider the case in which an instance of class `Widget` contains two instance properties: `$p1` has the integer value 10, and `$p2` is a handle to an array of elements of some type(s) or to an instance of some other type.

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Widget [VSlot $p1 *][VSlot $p2 *]]
                                                      |          |
                                                      V          V
                                          [VStore Int 10] [VStore Obj *]
                                                                     |
                                                                     V
                                                             [HStore ...]
```

Let us consider the result of `$b = clone $a`:

```
[VSlot $a *]-->[VStore Obj *]-->[HStore Widget [VSlot $p1 *][VSlot $p2 *]]
                                                      |            |
[VSlot $b *]-->[VStore Obj *]                         V            V
                             |                 [VStore Int 10] [VStore Obj *]
     +-----------------------+                                          |
     V                                                                  V
   [HStore Widget [VSlot $p1 *] [VSlot $p2 *]]             +--->[HStore ...]
                             |             |               |
                             V             V               |
                 [VStore Int 10] [VStore Obj *]-------------+
```

The clone operator will create another object HStore of the same class as the original, copy `$a`'s object's instance properties using member-copy assignment `=*` (§§). For the example shown above, the handle to the newly created HStore stored into `$b` using value assignment. Note that the clone operator will not recursively clone objects held in `$a`'s instance properties; hence the object copying performed by the clone operator is often referred to as a *shallow copy*. If a *deep copy* of an object is desired, the programmer must achieve this manually by using the method `__clone` (§§) or by other means.

Scope

The same name can designate different things at different places in a program. For each different thing that a name designates, that name is visible only within a part of the program called that name's *scope*. The following distinct scopes exist:

- Script, which means from the point of declaration/first initialization through to the end of that script, including any included

and required files (§§).
- Function, which means from the point of declaration/first initialization through to the end of that function (§§).
- Class, which means the body of that class and any classes derived from it (§§).
- Interface, which means the body of that interface, any interfaces derived from it, and any classes that implement it (§§).
- Namespace, which means from the point of declaration/first initialization through to the end of that namespace (§§).

A variable declared or first initialized inside a function, has function scope; otherwise, the variable has script scope.

Superglobals (§§) are always in scope; they never need explicit declaration.

Each function has its own function scope. An anonymous function (§§) has its own scope separate from that of any function inside which that anonymous function is defined.

The scope of a parameter is the body of the function in which the parameter is declared. For the purposes of scope, a catch-block (§§) is treated like a function body, in which case, the *variable-name* in *parameter-declaration-list* is treated like a parameter.

The scope of a *named-label* (§§) is the body of the function in which the label is defined.

The scope of a class member m (§§) declared in, or inherited by, a class type C is the body of C.

The scope of an interface member m (§§) declared in, or inherited by, an interface type I is the body of I.

When a trait (§§) is used by a class or an interface, the trait's members (§§) take on the scope of a member of that class or interface.

Storage Duration

The lifetime of a variable is the time during program execution that storage for that variable is guaranteed to exist. This lifetime is referred to as the variable's *storage duration*, of which there are three kinds: automatic, static, and allocated.

A variable having *automatic storage duration* comes into being and is initialized at its declaration or on its first use, if it has no declaration. Its lifetime is delimited by an enclosing scope (§§). The automatic variable's lifetime ends at the end of that scope. Automatic variables lend themselves to being stored on a stack where they can help support argument passing and recursion. Local variables (§§), which include function parameters (§§), have automatic storage duration.

A variable having *static storage duration* comes into being and is initialized before its first use, and lives until program shutdown. The following kinds of variables have static storage duration: constants (§§), function statics (§§), global variables (§§), static properties (§§), and class and interface constants (§§).

A variable having *allocated storage duration* comes into being based on program logic by use of the new operator (§§). Ordinarily, once such storage is no longer needed, it is reclaimed automatically by the Engine via its garbage-collection process (§§) and the use of destructors (§§). The following kinds of variables have allocated storage duration: array elements (§§) and instance properties (§§).

Although all three storage durations have default ends-of-life, their lives can be shortened by calling the intrinsic unset (§§), which destroys any given set of variables.

The following example demonstrates the three storage durations:

```
class Point { ... }

$av1 = new Point(0, 1);       // auto variable $av1 created and initialized
static $sv1 = ...;          // static variable $sv1 created and initialized

function doit($p1)
{
  $av2 = ...;           // auto variable $av2 created and initialized
```

```
  static $sv2 = ...;        // static variable $sv2 created and initialized
  if ($p1)
  {
    $av3 = ...;         // auto variable $av3 created and initialized
    static $sv3 = ...;    // static variable $sv3 created and initialized
    ...
  }
  global $av1;
  $av1 = new Point(2, 3);   // Point(0,1) is eligible for destruction
  ...
}                   // $av2 and $av3 are eligible for destruction

doit(TRUE);

// At end of script, $av1, $sv1, $sv2, and $sv3 are eligible for destruction
```

The comments indicate the beginning and end of lifetimes for each variable. In the case of the initial allocated Point variable whose handle is stored in `$av1`, its life ends when `$av1` is made to point to a different Point.

If function `doit` is called multiple times, each time it is called, its automatic variables are created and initialized, whereas its static variables retain their values from previous calls.

Consider the following recursive function:

```
function factorial($i)
{
  if ($i > 1) return $i * factorial($i - 1);
  else if ($i == 1) return $i;
  else return 0;
}
```

When `factorial` is first called, the local variable parameter `$i` is created and initialized with the value of the argument in the call. Then, if this function calls itself, the same process is repeated each call. Specifically, each time `factorial` calls itself, a new local variable parameter `$i` is created and initialized with the value of the argument in the call.

The lifetime of any VStore (§4.4.1) or HStore (§4.4.1) can be extended by the Engine as long as needed. Conceptually, the lifetime of a VStore ends when it is no longer pointed to by any VSlots (§§). Conceptually, the lifetime of an HStore ends when no VStores have a handle to it.

Types

General

The meaning of a value is determined by its *type*. PHP's types are categorized as *scalar types* and *composite types*. The scalar types are Boolean (§§), integer (§§), floating-point (§§), string (§§), and null (§§). The composite types are array (§§), object (§§), and resource (§§).

The scalar types are *value types*. That is, a variable of scalar type behaves as though it contains its own value. On the other hand, the composite types are *handle types*. A variable of composite type contains information—in a *handle*—that leads to the value. The differences between value and handle types become apparent when it comes to understanding the semantics of assignment, and passing arguments to, and returning values from, functions (§§). That said, array types really are a hybrid; on the one hand, an array may contain an arbitrary number of elements separate from the array variable itself, yet on the other hand, certain array operations do have value semantics.

Variables are not declared to have a particular type. Instead, a variable's type is determined at runtime by the context in which it is used.

Useful library functions for interrogating and using type information include `gettype` (§xx), `is_type` (§xx), `settype` (§xx), and `var_dump` (§xx).

Scalar Types

General

The integer and floating-point types are collectively known as *arithmetic types*. The library function `is_numeric` (§xx) indicates if a given value is a number or a numeric string (§§).

The library function `is_scalar` (§xx) indicates if a given value has a scalar type. However, that function does not consider `NULL` to be scalar. To test for `NULL`, use `is_null` (§xx).

The Boolean Type

The Boolean type is `bool`, for which the name boolean is a synonym. This type is capable of storing two distinct values, which correspond to the Boolean values `TRUE` and `FALSE` (§§), respectively. The representation of this type and its values is unspecified.

The library function `is_bool` (§xx) indicates if a given value has type `bool`.

The Integer Type

There is one integer type, `int`, for which the name integer is a synonym. This type is binary, signed, and uses twos-complement representation for negative values. The range of values that can be stored is implementation-defined; however, the range [-2147483648, 2147483647], must be supported.

Certain operations on integer values produce a mathematical result that cannot be represented as an integer. Examples include the following:

- Incrementing the largest value or decrementing the smallest value.
- Applying the unary minus to the smallest value.
- Multiplying, adding, or subtracting two values.

In such cases, the resulting type and value is implementation-defined, but must be one of the following:

- The computation is done as though the types of the values were `float` with the result having that type.
- The result type is int and the value reflects wrap-around (for example adding 1 to the largest value results in the smallest value).
- The computation is done as though the type had some unspecified, arithmetic-like object type with the result being mathematically correct.

The constants `PHP_INT_SIZE` (§6.3) and `PHP_INT_MAX` (§6.3) define certain characteristics about type `int`.

The library function `is_int` (§xx) indicates if a given value has type `int`.

The Floating-Point Type

There is one floating-point type, `float`, for which the names `double` and `real` are synonyms. The `float` type must support at least the range and precision of IEEE 754 64-bit double-precision representation.

The library function `is_float` (§xx) indicates if a given value has type `float`. The library function `is_finite` (§xx) indicates if a given floating-point value is finite. The library function `is_infinite` (§xx) indicates if a given floating-point value is infinite.

The library function `is_nan` (§xx) indicates if a given floating-point value is a `NaN`.

The String Type

A string is a set of contiguous bytes that represents a sequence of zero or more characters.

Conceptually, a string can be considered as an array (§§) of bytes—the *elements*—whose keys are the `int` values starting at zero. The type of each element is `string`. However, a string is *not* considered a collection, so it cannot be iterated over.

A string whose length is zero is an *empty string*.

As to how the bytes in a string translate into characters is unspecified.

Although a user of a string might choose to ascribe special semantics to bytes having the value `U+0000`, from PHP's perspective, such *null bytes* are simply just bytes! PHP does not assume strings contain any specific data or assign special values to any bytes or sequences. However, many library functions assume the strings they receive as arguments are UTF-8 encoded, often without explicitly mentioning that fact.

A *numeric string* is a string whose content exactly matches the pattern defined using integer format by the production *integer-literal* (§§) or using floating-point format by the production *floating-literal* (§§), where leading whitespace is permitted. A *leading-numeric string* is a string whose initial characters follow the requirements of a numeric string, and whose trailing characters are non-numeric. A *non-numeric string* is a string that is not a numeric string.

Only one mutation operation may be performed on a string, offset assignment, which involves the simple assignment operator = (§§).

The library function `is_string` (§xx) indicates if a given value has type string.

The Null Type

The null type has only one possible value, `NULL` (§§). The representation of this type and its value is unspecified.

The library function `is_null` (§xx) indicates if a given value is `NULL`.

Composite Types

Array Types

An array is a data structure that contains a collection of zero or more elements whose values are accessed through keys that are of type `int` or `string`. Arrays are described in §§.

The library function `is_array` (§xx) indicates if a given value is an array.

Object Types

An *object* is an instance of a class (§§). Each distinct *class-declaration* (§§) defines a new class type, and each class type is an object type. The representation of object types is unspecified.

The library function `is_object` (§xx) indicates if a given value is an object, and the library function `get_class` (§xx) indicates the name of an object's class.

Resource Types

A *resource* is a descriptor to some sort of external entity. (Examples include files, databases, and sockets).

A resource is an abstract entity whose representation is unspecified. Resources are only created or consumed by the implementation; they are never created or consumed by PHP code.

Each distinct resource has a unique ID of some unspecified form.

When scripts execute in a mode having a command-line interface, the following predefined resource constants that correspond to file streams are automatically opened at program start-up:

- STDIN, which maps to standard input (php://stdin).
- STDOUT, which maps to standard output (php://stdout).
- STDERR, which maps to standard error (php://stderr).

The library function `is_resource` (§xx) indicates if a given value is a resource, and the library function `get_resource_type` (§xx) indicates the type of a resource.

Constants

General

A *constant* is a name (§§) for a value that once given its initial value, cannot be changed.

A constant can be defined in one of two ways: as a *c-constant* using a *const-declaration* (§§), or as a *d-constant* by calling the library function `define` (§xx). However, the two approaches differ slightly. Specifically:

- The name of a c-constant must comply with the lexical grammar for a name while that for a d-constant can contain any source character.
- The name of a c-constant is case-insensitive while that for a d-constant can be case-sensitive or case-insensitive based on the value of the third argument passed to `define`.
- If `define` is able to define the given name, it returns `TRUE`; otherwise, it returns `FALSE`.

The library function `defined` (§xx) reports if a given name (specified as a string) is defined as a constant. The library function `constant` (§xx) returns the value of a given constant whose name is specified as a string.

Examples

```
const MAX_HEIGHT = 10.5;              // define two (case-insensitive) c-constants
const UPPER_LIMIT = MAX_HEIGHT;
define('COEFFICIENT_1', 2.345, TRUE); // define a case-insensitive d-constant
define('FAILURE', TRUE, FALSE);       // define a case-sensitive d-constant
```

Context-Dependent Constants

The following constants—sometimes referred to as *magic constants*—are automatically available to all scripts; their values are not fixed:

Constant Name	Description
`__CLASS__`	`string`; The name of the current class. From within a trait method, the name of the class in which that trait is used. If the current namespace is other than the default, the namespace name and "\" are prepended, in that order. If used outside all classes, the result is the empty string.
`__DIR__`	`string`; The directory name of the script. A directory separator is only appended for the root directory.
`__FILE__`	`string`; The full name of the script.

`__FUNCTION__`	`string` ; Inside a function, the name of the current function exactly as it was declared, with the following prepended: If a named namespace exists, that namespace name followed by "\". If used outside all functions, the result is the empty string. For a method, no parent-class prefix is present. (See `__METHOD__` and §§).
`__LINE__`	`int` ; the number of the current source line
`__METHOD__`	`string` ; Inside a method, the name of the current method exactly as it was declared, with the following prepended, in order: If a named namespace exists, that namespace name followed by "\"; the parent class name or trait name followed by `::` . If used outside all methods, the result is the same as for `__FUNCTION__` .
`__NAMESPACE__`	`string` ; The name of the current namespace exactly as it was declared. For the default namespace, the result is the empty string.
`__TRAIT__`	`string` ; The name of the current trait. From within a trait method, the name of the current trait. If used outside all traits, the result is the empty string.

Constants beginning with __ are reserved for future use by the Engine.

Core Predefined Constants

The following constants are automatically available to all scripts:

Constant Name	Description
`__COMPILER_HALT_OFFSET__`	`int` ; When the library function `__HALT_COMPILER__` (§xx) is called, this constant contains the location in the source file immediately following the `__HALT_COMPILER__()` ; token.
`DEFAULT_INCLUDE_PATH`	`string` ; the `fopen` library function (§xx) include path is used if it is not overridden by the `php.ini` setting `include_path` .
`E_ALL`	`int` ; All errors and warnings, as supported.
`E_COMPILE_ERROR`	`int` ; Fatal compile-time errors. This is like an `E_ERROR` , except that `E_COMPILE_ERROR` is generated by the scripting engine.
`E_COMPILE_WARNING`	`int` ; Compile-time warnings (non-fatal errors). This is like an `E_WARNING` , except that `E_COMPILE_WARNING` is generated by the scripting engine.
`E_CORE_ERROR`	`int` ; Fatal errors that occur during PHP's initial start-up. This is like an `E_ERROR` , except that `E_CORE_ERROR` is generated by the core of PHP.
`E_CORE_WARNING`	`int` ; Warnings (non-fatal errors) that occur during PHP's initial start-up. This is like an `E_WARNING` , except that `E_CORE_WARNING` is generated by the core of PHP.
`E_DEPRECATED`	`int` ; Run-time notices. Enable this to receive warnings about code that will not work in future versions.
`E_ERROR`	`int` ; Fatal run-time errors. These indicate errors that cannot be recovered from, such as a memory allocation problem. Execution of the script is halted.
`E_NOTICE`	`int` ; Run-time notices. Indicate that the script encountered something that could indicate an error, but could also happen in the normal course of running a script.
`E_PARSE`	`int` ; Compile-time parse errors.
`E_RECOVERABLE_ERROR`	`int` ; Catchable fatal error. It indicates that a probably dangerous error occurred, but did not leave the Engine in an unstable state. If the error is not caught by a user defined handler (see the library function `set_error_handler` (§xx)), the application aborts as it was

	an `E_ERROR`.
`E_STRICT`	`int`; Have PHP suggest changes to the source code to ensure the best interoperability.
`E_USER_DEPRECATED`	`int`; User-generated error message. This is like an `E_DEPRECATED`, except that `E_USER_DEPRECATED` is generated in PHP code by using the library function `trigger_error` (§xx).
`E_USER_ERROR`	`int`; User-generated error message. This is like an `E_ERROR`, except that `E_USER_ERROR` is generated in PHP code by using the library function `trigger_error` (§xx).
`E_USER_NOTICE`	`int`; User-generated warning message. This is like an `E_NOTICE`, except that `E_USER_NOTICE` is generated in PHP code by using the library function `trigger_error` (§xx).
`E_USER_WARNING`	`int`; User-generated warning message. This is like an `E_WARNING`, except that `E_USER_WARNING` is generated in PHP code by using the library function `trigger_error` (§xx).
`E_WARNING`	`int`; Run-time warnings (non-fatal errors). Execution of the script is not halted.
`E_USER_DEPRECATED`	`int`; User-generated warning message. This is like an `E_DEPRECATED`, except that `E_USER_DEPRECATED` is generated in PHP code by using the library function `trigger_error` (§xx).
`FALSE`	`bool`; the case-insensitive Boolean value `FALSE`.
`INF`	`float`; Infinity
`M_1_PI`	`float`; 1/pi
`M_2_PI`	`float`; 2/pi
`M_2_SQRTPI`	`float`; 2/sqrt(pi)
`M_E`	`float`; e
`M_EULER`	`float`; Euler constant
`M_LN10`	`float`; log_e 10
`M_LN2`	`float`; log_e 2
`M_LNPI`	`float`; log_e(pi)
`M_LOG10E`	`float`; log_10 e
`M_LOG2E`	`float`; log_2 e
`M_PI`	`float`; Pi
`M_PI_2`	`floa`t; pi/2
`M_PI_4`	`float`; pi/4
`M_SQRT1_2`	`float`; 1/sqrt(2)
`M_SQRT2`	`float`; sqrt(2)
`M_SQRT3`	`float`; sqrt(3)
`M_SQRTPI`	`float`; sqrt(pi)
`NAN`	`float`; Not-a-Number
`NULL`	`null`; the case-insensitive value `NULL`.

`PHP_BINARY`	`string`; the PHP binary path during script execution.
`PHP_BINDIR`	`string`; the installation location of the binaries.
`PHP_CONFIG_FILE_PATH`	`string`; location from which php.ini values were parsed
`PHP_CONFIG_FILE_SCAN_DIR`	`string`; The directory containing multiple INI files, all of which were parsed on start-up.
`PHP_DEBUG`	`int`; Indicates whether the engine was built with debugging enabled.
`PHP_EOL`	`string`; the end-of-line terminator for this platform.
`PHP_EXTENSION_DIR`	`string`; The directory to be searched by the library function dl (§xx) when looking for runtime extensions.
`PHP_EXTRA_VERSION`	`string`; the current PHP extra version.
`PHP_INT_MAX`	`int`; the largest representable value for an integer.
`PHP_INT_SIZE`	`int`; the number of bytes used to represent an integer.
`PHP_MAJOR_VERSION`	`int`; the current PHP major version
`PHP_MANDIR`	`string`; the installation location of the manual pages.
`PHP_MAXPATHLEN`	`int`; the maximum length of a fully qualified filename supported by this build.
`PHP_MINOR_VERSION`	`int`; the current PHP minor version
`PHP_OS`	`string`; the current operating system.
`PHP_PREFIX`	`string`; the value to which "--prefix" was set when configured.
`PHP_RELEASE_VERSION`	`int`; the current PHP release version
`PHP_ROUND_HALF_DOWN`	`int`; Round halves down
`PHP_ROUND_HALF_EVEN`	`int`; Round halves to even numbers
`PHP_ROUND_HALF_ODD`	`int`; Round halves to odd numbers
`PHP_ROUND_HALF_UP`	`int`; Round halves up
`PHP_SAPI`	`string`; the Server API for this build.
`PHP_SHLIB_SUFFIX`	`string`; build-platform's shared library suffix.
`PHP_SYSCONFDIR`	`string`; the PHP system configuration directory.xx
`PHP_VERSION`	`string`; the current PHP version in the form "major.minor.release[extra]".
`PHP_VERSION_ID`	`int`; the current PHP version
`PHP_ZTS`	`int`; Indicates whether the compiler was built with thread safety enabled.
`TRUE`	`bool`; the case-insensitive Boolean value `TRUE`.

The members of the `E_\*` family have values that are powers of 2, so they can be combined meaningfully using bitwise operators.

User-Defined Constants

A constant may be defined inside or outside of functions (§§), inside a class (§§), or inside an interface (§§).

Variables

General

A *variable* is a named area of data storage that has a type and a value, both of which can change. A variable is represented by a VSlot (§§). A variable is created by assigning a value to it (§§, §§, §§, §§, §§). A variable is destroyed by *unsetting* it, either by an explicit call to the intrinsic unset (§§), or by the Engine. The intrinsic `isset` (§§) tests if a given variable exists and is not set to `NULL`. A variable that somehow becomes defined, but is not initialized starts out with the value `NULL`.

Variables have names as defined in §§. Distinct variables may have the same name provided they are in different scopes (§§).

A constant (§§) is a variable that, once initialized, its value cannot be changed.

Based on the context in which it is declared, a variable has a scope (§§) and a storage duration (§§).

A *superglobal* variable is one that is accessible in all scopes without the need for a *global-declaration* (§§).

The following kinds of variable may exist in a script:

- Constant (§§).
- Local variable (§§).
- Array element (§§).
- Function static (§§).
- Global variable (§§).
- Instance property (§§).
- Static property (§§).
- Class and interface constant (§§).

Kinds of Variables

Constants

Syntax:

See §§.

Constraints:

Outside of a class or interface, a c-constant can be defined only at the top level of a script.

Semantics:

See §§ and §§.

A constant defined outside of a class or interface is a superglobal (§§).

A constant defined inside a function has function scope (§§). A constant defined at the top level has script scope. A constant has static storage duration (§§) and is a non-modifiable lvalue.

Examples

```
const MAX_HEIGHT = 10.5;        // define two c-constants
const UPPER_LIMIT = MAX_HEIGHT;
define('COEFFICIENT_1', 2.345); // define two d-constants
define('FAILURE', TRUE);
```

Local Variables

Syntax:

See Semantics below.

Semantics:

Except for a parameter, a local variable is never defined explicitly; instead, it is created when it is first assigned a value. A local variable can be assigned to as a parameter in the parameter list of a function definition (§§) or inside any compound statement (§§). It has function scope (§§) and automatic storage duration (§§). A local variable is a modifiable lvalue.

Examples

```
function doit($p1)  // assigned the value TRUE when called
{
  $count = 10;
    ...
  if ($p1)
  {
    $message = "Can't open master file.";
    ...
  }
  ...
}
doit(TRUE);
// -----------------------------------------
function f()
{
  $lv = 1;
  echo "\$lv = $lv\n";
  ++$lv;
}
for ($i = 1; $i <= 3; ++$i)
  f();
```

Unlike the function static equivalent in §§, function `f` outputs "`$lv = 1`" each time.

See the recursive function example in §§.

Array Elements

Syntax:

Arrays (§§) are created via the array-creation operator (§§) or the intrinsic `array` (§§). At the same time, one or more elements may be created for that array. New elements are inserted into an existing array via the simple-assignment operator (§§) in conjunction with the subscript operator `[]` (§§). Elements can be removed by calling the `unset` intrinsic (§§).

Semantics:

The scope (§§) of an array element is the same as the scope of that array's name. An array element has allocated storage duration (§§).

Examples

```
$colors = ["red", "white", "blue"]; // create array with 3 elements
$colors[] = "green";                // insert a new element
```

Function Statics

Syntax:

```
function-static-declaration:
  static name   function-static-initializer_opt ;
function-static-initializer:
  = const-expression
```

name is defined in (§§), and *const-expression* is defined in (§§).

Constraints:

A function static must be defined inside a function.

Semantics:

A function static may be defined inside any compound statement (§§). It is a modifiable lvalue.

A function static has function scope (§§) and static storage duration (§§).

The value of a function static is retained across calls to its parent function. Each time the function containing a function static declaration is called, that execution is dealing with an alias (§§) to that static variable. If that alias is passed to the `unset` intrinsic (§§), only that alias is destroyed. The next time that function is called, a new alias is created.

Examples

```
function f()
{
  static $fs = 1;
  echo "\$fs = $fs\n";
  ++$fs;
}
for ($i = 1; $i <= 3; ++$i)
  f();
```

Unlike the local variable equivalent in §§, function `f` outputs "`$fs = 1`", "`$fs = 2`", and "`$fs = 3`", as `$fs` retains its value across calls.

Global Variables

Syntax:

```
global-declaration:
  global variable-name-list ;

variable-name-list:
  expression
  variable-name-list  , expression
```

expression is defined in §§.

Constraints:

Each *expression* must designate a variable name.

Semantics:

A global variable is never defined explicitly; instead, it is created when it is first assigned a value. That may be done at the top level of a script, or from within a block in which that variable has been declared (*imported*, that is) using the `global` keyword.

As described in §§, `$GLOBALS` is a superglobal (§§) array whose elements' key/value pairs contain the name and value, respectively, of each global variable currently defined. As such, a global variable `gv` can be initialized with the value `v`, and possibly be created, using the following form of assignment:

`$GLOBALS['gv'] = v`

As `$GLOBALS` is a superglobal, `gv` need not first be the subject of a *global-declaration*.

A global variable has script scope (§§) and static storage duration (§§). A global variable is a modifiable lvalue.

When a global value is imported into a function, each time the function is called, that execution is dealing with an alias (§§) to that global variable. If that alias is passed to the `unset` intrinsic (§§), only that alias is destroyed. The next time that function is called, a new alias is created with the current value of the global variable.

Examples

```
$colors = array("red", "white", "blue");
$GLOBALS['done'] = FALSE;
// -----------------------------------------
$min = 10; $max = 100; $average = NULL;
global $min, $max;         // allowed, but serves no purpose
function compute($p)
{
  global $min, $max;
  global $average;
  $average = ($max + $min)/2;

  if ($p)
  {
    global $result;
    $result = 3.456;  // initializes a global, creating it, if necessary
  }
}
compute(TRUE);
echo "\$average = $average\n";  // $average = 55
echo "\$result = $result\n";  // $result = 3.456
// -----------------------------------------
$g = 100;
function f()
{
  $v = 'g';
  global $$v;          // import global $g
  ...
}
```

Instance Properties

These are described in (§§). They have class scope (§§) and allocated storage duration (§§).

Static Properties

These are described in (§§). They have class scope (§§) and static storage duration (§§).

Class and Interface Constants

These are described in §§ and §§. They have class or interface scope (§§) and static storage duration (§§).

Predefined Variables

The following variables are automatically available to all scripts:

Variable Name	Description
`$argc`	`int`; The number of command-line arguments passed to the script. This is at least 1. (See `$argv` below).
`$argv`	`array`; An array of `$argc` elements containing the command-line arguments passed to the script as strings. Each element has an `int` key with the keys being numbered sequentially starting at zero through `$argc-1`. `$argv[0]` is the name of the script. It is implementation-defined as to how white space on command lines is handled, whether letter casing is preserved, which characters constitute quotes, or how `$argv[0]`'s string is formatted. As to how command-line arguments are defined, is unspecified.
`$_COOKIE`	`array`; The variables passed to the current script via HTTP Cookies.
`$_ENV`	`array`; A superglobal (§§) array in which the environment variable names are element keys, and the environment variable value strings are element values. As to how an environment variable is defined, is unspecified.
`$_FILES`	`array`; The items uploaded to the current script via the HTTP POST method.
`$_GET`	`array`; The variables passed to the current script via the URL parameters.
`$GLOBALS`	`array`; A superglobal (§§) array containing the names of all variables that are currently defined in the global scope of the script. The variable names are the element keys, and the variable values are the element values.
`$_POST`	`array`; The variables passed to the current script via the HTTP POST method.
`$_REQUEST`	`array`; By default contains the contents of `$_COOKIE`, `$_GET`, and `$_POST`.
`$_SERVER`	`array`; Server and execution environment information, such as headers, paths, and script locations. The entries in this array are created by the web server.
`$_SESSION`	`array`; The session variables available to the current script.

Conversions

General

Some operators implicitly convert automatically the values of operands from one type to another. Explicit conversion is performed using the cast operator (§§).

If an expression is converted to its own type, the type and value of the result are the same as the type and value of the expression.

Converting to Boolean Type

The result type is `bool`.

If the source type is `int` or `float`, then if the source value tests equal to 0, the result value is `FALSE`; otherwise, the result value is `TRUE`.

If the source value is `NULL`, the result value is `FALSE`.

If the source is an empty string or the string "0", the result value is `FALSE`; otherwise, the result value is `TRUE`.

If the source is an array with zero elements, the result value is `FALSE`; otherwise, the result value is `TRUE`.

If the source is an object, the result value is `TRUE`.

If the source is a resource, the result value is `TRUE`.

The library function `boolval` (§xx) allows values to be converted to `bool`.

Converting to Integer Type

The result type is `int`.

If the source type is `bool`, then if the source value is `FALSE`, the result value is 0; otherwise, the result value is 1.

If the source type is `float`, for the values `INF`, `-INF`, and `NAN`, the result value is implementation-defined. For all other values, if the precision can be preserved, the fractional part is rounded towards zero and the result is well defined; otherwise, the result is undefined.

If the source value is `NULL`, the result value is 0.

If the source is a numeric string or leading-numeric string (§§) having integer format, if the precision can be preserved the result value is that string's integer value; otherwise, the result is undefined. If the source is a numeric string or leading-numeric string having floating-point format, the string's floating-point value is treated as described above for a conversion from `float`. The trailing non-numeric characters in leading-numeric strings are ignored. For any other string, the result value is 0.

If the source is an array with zero elements, the result value is 0; otherwise, the result value is 1.

If the source is an object, the conversion is invalid.

If the source is a resource, the result is the resource's unique ID.

The library function `intval` (§xx) allows values to be converted to `int`.

Converting to Floating-Point Type

The result type is `float`.

If the source type is `int`, if the precision can be preserved the result value is the closest approximation to the source value; otherwise, the result is undefined.

If the source is a numeric string or leading-numeric string (§§) having integer format, the string's integer value is treated as described above for a conversion from `int`. If the source is a numeric string or leading-numeric string having floating-point format, the result value is the closest approximation to the string's floating-point value. The trailing non-numeric characters in leading-numeric strings are ignored. For any other string, the result value is 0.

If the source is an object, the conversion is invalid.

For sources of all other types, the conversion is performed by first converting the source value to `int` (§§) and then to `float`.

If the source is a resource, the result is the resource's unique ID.

The library function `floatval` (§xx) allows values to be converted to float.

Converting to String Type

The result type is string.

If the source type is `bool`, then if the source value is `FALSE`, the result value is the empty string; otherwise, the result value is "1".

If the source type is `int` or `float`, then the result value is a string containing the textual representation of the source value (as specified by the library function `sprintf` (§xx)).

If the source value is `NULL`, the result value is an empty string.

If the source is an array, the result value is the string "Array".

If the source is an object, then if that object's class has a `__toString` method (§§), the result value is the string returned by that method; otherwise, the conversion is invalid.

If the source is a resource, the result value is an implementation-defined string.

The library function `strval` (§xx) allows values to be converted to string.

Converting to Array Type

The result type is `array`.

If the source type is `bool`, `int`, `float`, or `string`, the result value is an array of one element whose type and value is that of the source.

If the source value is `NULL`, the result value is an array of zero elements.

If the source is an object, the result is an array of zero or more elements, where the elements are key/value pairs corresponding to the object's instance properties. The order of insertion of the elements into the array is the lexical order of the instance properties in the *class-member-declarations* (§§) list. The key for a private instance property has the form "\0*name*\0*name*", where the first *name* is the class name, and the second name is the property name. The key for a protected instance property has the form "\0*\0*name*", where *name* is that of the property. The key for a public instance property has the form "*name*", where *name* is that of the property. The value for each key is that from the corresponding property's initializer, if one exists, else `NULL`.

If the source is a resource, the result is an array of one element containing the implementation-defined value of the resource.

Converting to Object Type

The result type is `object`.

If the source has any type other than object, the result is an instance of the predefined class `stdClass` (§§). If the value of the source is `NULL`, the instance is empty. If the value of the source has a scalar type and is non-`NULL`, the instance contains a public property called scalar whose value is that of the source. If the value of the source is an array, the instance contains a set of public properties whose names and values are those of the corresponding key/value pairs in the source. The order of the properties is the order of insertion of the source's elements.

Lexical Structure

Scripts

A script (§§) is an ordered sequence of characters. Typically, a script has a one-to-one correspondence with a file in a file system, but this correspondence is not required.

Conceptually speaking, a script is translated using the following steps:

1. Transformation, which converts a script from a particular character repertoire and encoding scheme into a sequence of 8-bit characters.

2. Lexical analysis, which translates a stream of input characters into a stream of tokens.

3. Syntactic analysis, which translates the stream of tokens into executable code.

Conforming implementations must accept scripts encoded with the UTF-8 encoding form (as defined by the Unicode standard), and transform them into a sequence of characters. Implementations can choose to accept and transform additional character encoding schemes.

Grammars

This specification shows the syntax of the PHP programming language using two grammars. The *lexical grammar* defines how source characters are combined to form white space, comments, and tokens. The *syntactic grammar* defines how the resulting tokens are combined to form PHP programs.

The grammars are presented using *grammar productions*, with each one defining a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In productions, non-terminal symbols are shown in slanted type *like this*, and terminal symbols are shown in a fixed-width font `like this`.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by one colon for a syntactic grammar production, and two colons for a lexical grammar production. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

```
single-line-comment::
  // input-charactersopt
  #  input-charactersopt
```

defines the lexical grammar production *single-line-comment* as being the terminals `//` or `#`, followed by an optional *input-characters*. Each expansion is listed on a separate line.

Although alternatives are usually listed on separate lines, when there is a large number, the shorthand phrase “one of” may precede a list of expansions given on a single line. For example,

```
hexadecimal-digit: one of
  0   1   2   3   4   5   6   7   8   9
  a   b   c   d   e   f
  A   B   C   D   E   F
```

Lexical analysis

General

The production *input-file* is the root of the lexical structure for a script. Each script must conform to this production.

Syntax:

```
input-file::
  input-element
  input-file   input-element
input-element::
  comment
  white-space
  token
```

comment is defined in §§; white-space is defined in §§, and token is defined in §§.

Semantics:

The basic elements of a script are comments, white space, and tokens.

The lexical processing of a script involves the reduction of that script into a sequence of tokens (§§) that becomes the input to the syntactic analysis. Tokens can be separated by white space (§§) and delimited comments (§§).

Lexical processing always results in the creation of the longest possible lexical element. (For example, `$a+++++$b` must be parsed as `$a++ ++ +$b`, which syntactically is invalid).

Comments

Two forms of comments are supported: delimited comments and single-line comments.

Syntax:

```
comment::
  single-line-comment
  delimited-comment

single-line-comment::
  //   input-charactersopt
    #    input-charactersopt

input-characters::
  input-character
  input-characters   input-character

input-character::
  Any source character except new-line

new-line::
  Carriage-return character (U+000D)
  Line-feed character (U+000A)
  Carriage-return character (U+000D) followed by line-feed character (U+000A)

delimited-comment::
  /*   No characters or any source character sequence except /*   */
```

Semantics:

Except within a string literal or a comment, the characters / start a delimited comment, which ends with the characters */. Except within a string literal or a comment, the characters // or # start a single-line comment, which ends with a new line. That new line is not part of the comment. However, if the single-line comment is the last source element in an embedded script, the trailing new line can be omitted. (Note: this allows for uses like `<?php ... // ... ?>`).*

A delimited comment can occur in any place in a script in which white space (§§) can occur. (For example; `/*...*/$c/*...*/=/*...*/567/*...*/;/*...*/` is parsed as `$c=567;`, and `$k = $i+++/*...*/++$j;` is parsed as `$k = $i+++ ++$j;`).

Implementation Notes

During tokenizing, an implementation can treat a delimited comment as though it was white space.

White Space

White space consists of an arbitrary combination of one or more new-line, space, horizontal tab, vertical tab, and form-feed characters.

Syntax:

```
white-space::
  white-space-character
  white-space   white-space-character

white-space-character::
  new-line
  Space character (U+0020)
  Horizontal-tab character (U+0009)
```

new-line is defined in §§.

Semantics:

The space and horizontal tab characters are considered horizontal white-space characters.

Tokens

General

There are several kinds of source tokens:

Syntax:

```
token::
  variable-name
  name
  keyword
  literal
  operator-or-punctuator
```

variable-name and name are defined in §§; keyword is defined in §§; literal is defined in §§; and operator-or-punctuator is defined in §§.

Names

Syntax:

```
variable-name::
  $   name

namespace-name::
  name
  namespace-name   \   name

namespace-name-as-a-prefix::
  \
  \opt   namespace-name   \
```

```
    namespace   \
    namespace   \   namespace-name   \

  qualified-name::
    namespace-name-as-a-prefix_opt   name

  name::
    name-nondigit
    name   name-nondigit
    name   digit

  name-nondigit::
    nondigit
    one of the characters U+007f–U+00ff

  nondigit:: one of
    _
    a   b   c   d   e   f   g   h   i   j   k   l   m
    n   o   p   q   r   s   t   u   v   w   x   y   z
    A   B   C   D   E   F   G   H   I   J   K   L   M
    N   O   P   Q   R   S   T   U   V   W   X   Y   Z
```

digit is defined in §§

Semantics:

Names are used to identify the following: constants (§§), variables (§§), labels (§§), functions (§§), classes (§§), class members (§§), interfaces (§§), traits (§§), namespaces (§§), and names in heredoc (§§) and nowdoc comments (§§).

*A name begins with an underscore (_), name-nondigit, or extended name character in the range U+007f– ** U+00ff. Subsequent characters can also include digits. A variable name is a name with a leading dollar ($).*

Unless stated otherwise (§§, §§, §§, §§, §§, §§), names are case-sensitive, and every character in a name is significant.

Function and method names beginning with two underscores (__) are reserved by the PHP language.

Variable names and function names (when used in a function-call context) need not be defined as source tokens; they can also be created at runtime using the variable name-creation operator (§§). (For example, given `$a = "Total"; $b = 3; $c = $b + 5;`, `${$a.$b.$c} =TRUE;` is equivalent to `$Total38 = TRUE;`, and `${$a.$b.$c}()` is equivalent to `Total38()`).

Examples

```
const MAX_VALUE = 100;
function getData() { ... }
class Point { ... }
interface ICollection { ... }
```

Implementation Notes

An implementation is discouraged from placing arbitrary restrictions on name length or length of significance.

Keywords

A keyword is a name-like sequence of characters that is reserved, and cannot be used as a name.

Syntax:

```
  keyword:: one of
    abstract   and   as   break   callable   case   catch   class   clone
    const   continue   declare   default   do   echo   else   elseif
    enddeclare   endfor   endforeach   endif   endswitch   endwhile
```

```
  extends   final   finally   for   foreach   function   global
  goto   if   implements   include   include_once   instanceof
  insteadof   interface   namespace   new or   print   private
  protected   public   require   require_once   return static   switch
  throw   trait   try   use   var   while   xor   yield
```

Semantics:

Keywords are not case-sensitive.

Literals

General

The source code representation of a value is called a literal.

Syntax:

```
literal::
  boolean-literal
  integer-literal
  floating-literal
  string-literal
  null-literal
```

boolean-literal is defined in §§; integer-literal is defined in §§; floating-literal is defined in §§; string-literal is defined in §§; and null-literal is defined in §§.

Boolean Literals

Syntax:

```
boolean-literal::
  TRUE (written in any case combination)
  FALSE (written in any case combination)
```

Semantics:

The type of a boolean-literal is bool. The values `TRUE` and `FALSE` represent the Boolean values True and False, respectively.

Examples

```
$done = FALSE;
computeValues($table, TRUE);
```

Integer Literals

Syntax:

```
integer-literal::
  decimal-literal
  octal-literal
  hexadecimal-literal
  binary-literal

  decimal-literal::
    nonzero-digit
    decimal-literal   digit
```

```
octal-literal::
  0
  octal-literal   octal-digit

hexadecimal-literal::
  hexadecimal-prefix   hexadecimal-digit
  hexadecimal-literal   hexadecimal-digit

hexadecimal-prefix:: one of
  0x  0X

binary-literal::
  binary-prefix   binary-digit
  binary-literal   binary-digit

binary-prefix:: one of
  0b  0B

digit:: one of
  0  1  2  3  4  5  6  7  8  9

nonzero-digit:: one of
  1  2  3  4  5  6  7  8  9

octal-digit:: one of
  0  1  2  3  4  5  6  7

hexadecimal-digit:: one of
  0  1  2  3  4  5  6  7  8  9
        a  b  c  d  e  f
        A  B  C  D  E  F

binary-digit:: one of
    0  1
```

Semantics:

The value of a decimal integer literal is computed using base 10; that of an octal integer literal, base 8; that of a hexadecimal integer literal, base 16; and that of a binary integer literal, base 2.

If the value of an integer-literal can be represented in type int, that is its type; otherwise, its type is float, as described below.

*Using a twos-complement system, can the smallest negative value (-2147483648 for 32 bits and -9223372036854775808 for 64 bits) be represented as a decimal integer literal? No. Consider the expression -5. This is made up of two tokens: a unary minus followed by the integer literal 5. As such, **there is no such thing as a negative-valued decimal integer literal in PHP**. Instead, there is the non-negative value, which is then negated. However, if the non-negative value is too large to represent as an `int`, it becomes `float`, which is then negated. Literals written using hexadecimal, octal, or binary notations are considered to have non-negative values.*

Examples

```
$count = 10      // decimal 10

0b101010 >> 4    // binary 101010 and decimal 4

0XAF << 023      // hexadecimal AF and octal 23
```

On an implementation using 32-bit int representation

```
2147483648 -> 2147483648 (too big for int, so is a float)
```

```
-2147483648 -> -2147483648 (too big for int, so is a float, negated)

-2147483647 - 1 -> -2147483648 fits in int

0x80000000 -> 2147483648 (too big for int, so is a float)
```

Floating-Point Literals

Syntax:

```
floating-literal::
  fractional-literal   exponent-part_opt
  digit-sequence   exponent-part

fractional-literal::
  digit-sequence_opt . digit-sequence
  digit-sequence .

exponent-part::
  e  sign_opt   digit-sequence
  E  sign_opt   digit-sequence

sign:: one of
  +  -

digit-sequence::
  digit
  digit-sequence   digit
```

digit is defined in §§.

Constraints

The value of a floating-point literal must be representable by its type.

Semantics:

The type of a *floating-literal* is `float`.

The constants `INF` (§6.3) and `NAN` (§6.3) provide access to the floating-point values for infinity and Not-a-Number, respectively.

Examples

```
$values = array(1.23, 3e12, 543.678E-23);
```

String Literals

Syntax:

```
string-literal::
  single-quoted-string-literal
  double-quoted-string-literal
  heredoc-string-literal
  nowdoc-string-literal
```

single-quoted-string-literal is defined in §§; *double-quoted-string-literal* is defined in §§; *heredoc-string-literal* is defined in §§; and *nowdoc-string-literal* is defined in §§.

Note: By conventional standards, calling *heredoc-string-literals* (§) and *nowdoc-string-literals* (§§) literals is a stretch, as each *is hardly a single token.*

Semantics:

A string literal is a sequence of zero or more characters delimited in some fashion. The delimiters are not part of the literal's content.

The type of a string literal is string.

Single-Quoted String Literals

Syntax:

```
  single-quoted-string-literal::
    b_opt  ' sq-char-sequence_opt  '

  sq-char-sequence::
    sq-char
    sq-char-sequence   sq-char

  sq-char::
    sq-escape-sequence
    \_opt   any member of the source character set except single-quote (') or backslash (\)

  sq-escape-sequence:: one of
    \'  \\
```

Semantics:

A single-quoted string literal is a string literal delimited by single-quotes ('). The literal can contain any source character except single-quote (') and backslash (\), which can only be represented by their corresponding escape sequence.

The optional `b` prefix is reserved for future use in dealing with so-called binary strings. For now, a single-quoted-string-literal with a `b` prefix is equivalent to one without.

A single-quoted string literal is a c-constant (§§).

Examples

```
'This text is taken verbatim'

'Can embed a single quote (\') and a backslash (\\) like this'
```

Double-Quoted String Literals

Syntax:

```
  double-quoted-string-literal::
    b_opt  " dq-char-sequence_opt  "

  dq-char-sequence::
    dq-char
    dq-char-sequence   dq-char

  dq-char::
    dq-escape-sequence
    any member of the source character set except double-quote (") or backslash (\)
    \  any member of the source character set except "\$efnrtvxX or
octal-digit

  dq-escape-sequence::
```

```
    dq-simple-escape-sequence
    dq-octal-escape-sequence
    dq-hexadecimal-escape-sequence

  dq-simple-escape-sequence:: one of
    \"   \\   \$   \e   \f   \n   \r   \t   \v

  dq-octal-escape-sequence::
    \   octal-digit
    \   octal-digit   octal-digit
    \   octal-digit   octal-digit   octal-digit

  dq-hexadecimal-escape-sequence::
    \x  hexadecimal-digit   hexadecimal-digit_opt
    \X  hexadecimal-digit   hexadecimal-digit_opt
```

octal-digit and hexadecimal-digit are defined in §§.

Semantics:

A double-quoted string literal is a string literal delimited by double-quotes ("). The literal can contain any source character except double-quote (") and backslash (\), which can only be represented by their corresponding escape sequence. Certain other (and sometimes non-printable) characters can also be expressed as escape sequences.

The optional `b` prefix is reserved for future use in dealing with so-called binary strings. For now, a double-quoted-string-literal with a `b` prefix is equivalent to one without.

An escape sequence represents a single-character encoding, as described in the table below:

Escape sequence	Character name
\$	Dollar sign
\"	Double quote
\	Backslash
\e	Escape
\f	Form feed
\n	New line
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\ooo	1–3-digit octal digit value ooo
\xhh or \Xhh	1–2-digit hexadecimal digit value hh

Within a double-quoted string literal, except when recognized as the start of an escape sequence, a backslash (\) is retained verbatim.

Within a double-quoted string literal a dollar ($) character not escaped by a backslash (\) is handled, as follows:

- *If that dollar ($) character plus the character sequence following spells a longest-possible variable name:*
- *For a scalar type, that variable name is replaced by the string representation of that variable's value, if such a variable exists. This is known as variable substitution. If no such variable is currently defined, the value substituted is the empty string. (For the purposes of variable substitution, the string representation is produced as if the library function `sprintf` was used. In the case of a floating-point value, the conversion specifier used is `%.nG`, where the precision `n` is*

implementation-defined.

- *For a variable that designates an array, if that variable name is followed by characters of the form "`[index]`" without any intervening white space, the variable name and these following characters are presumed to refer to the corresponding element of that array, in which case, the value of that element is substituted. If `index` is itself a variable having scalar type, that variable's value is substituted. If `index` is an integer literal, it must be a decimal-integer literal. `index` must not be a character sequence that itself looks like an array subscript or a class property.*
- *For a variable that designates an array, but no subscript-like character sequence follows that variable name, the value substituted is "Array".*
- *For a variable that designates an instance of a class, if that variable name is followed by characters of the form "`->name`" without any intervening white space, the variable name and these following characters are presumed to refer to the corresponding property of that instance, in which case, the value of that property is substituted.*
- *Otherwise, the dollar ($) is retained verbatim.*

Variable substitution also provides limited support for the evaluation of expressions. This is done by enclosing an expression in a pair of matching braces ({...}). The opening brace must be followed immediately by a dollar ($) without any intervening white space, and that dollar must begin a variable name. If this is not the case, braces are treated verbatim. An opening brace ({) cannot be escaped.

A double-quoted string literal is a c-constant (§§) if it does not contain any variable substitution.

Examples

```
$x = 123;
echo ">\$x.$x"."<"; // → >$x.123<
// -----------------------------------------
$colors = array("red", "white", "blue");
$index = 2;
echo "\$colors[$index] contains >$colors[$index]<\n";
  // → $colors[2] contains >blue<
// -----------------------------------------
class C {
    public $p1 = 2;
}
$myC = new C();
echo "\$myC->p1 = >$myC->p1<\n";  // → $myC->p1 = >2<
```

Heredoc String Literals

Syntax:

```
  heredoc-string-literal::
    <<<  hd-start-identifier   new-line   hd-char-sequence_opt  new-line hd-end-identifier  ;_opt   new-line

  hd-start-identifier::
    name

  hd-end-identifier::
    name

  hd-char-sequence::
    hd-char
    hd-char-sequence   hd-char

  hd-char::
    hd-escape-sequence
    any member of the source character set except backslash (\)
    \  any member of the source character set except \$efnrtvxX or
octal-digit

  hd-escape-sequence::
```

```
    hd-simple-escape-sequence
    dq-octal-escape-sequence
    dq-hexadecimal-escape-sequence

  hd-simple-escape-sequence:: one of
    \\   \$   \e   \f   \n   \r   \t   \v
```

name is defined in §§; new-line is defined in §§; and dq-octal-escape-sequence and dq-hexadecimal-escape-sequence are defined in §§.

Constraints

The start and end identifier must be the same. Only horizontal white space is permitted between `<<<` and the start identifier. No white space is permitted between the start identifier and the new-line that follows. No white space is permitted between the new-line and the end identifier that follows. Except for an optional semicolon (`;`), no characters—not even comments or white space—are permitted between the end identifier and the new-line that terminates that source line.

Semantics:

A heredoc string literal is a string literal delimited by "`<<< name`" and "`name`". The literal can contain any source character. Certain other (and sometimes non-printable) characters can also be expressed as escape sequences.

A heredoc literal supports variable substitution as defined for double-quoted string literals (§§).

A heredoc string literal is a c-constant (§§) if it does not contain any variable substitution.

Examples

```
$v = 123;
$s = <<<    ID
S'o'me "\"t e\txt; \$v = $v"
Some more text
ID;
echo ">$s<";
→ >S'o'me "\"t e  xt; $v = 123"
Some more text<
```

Nowdoc String Literals

Syntax:

```
  nowdoc-string-literal::
    <<<  '  hd-start-identifier  '  new-line  hd-char-sequence_opt   new-line hd-end-identifier  ;_opt   new-line
```

hd-start-identifier, hd-char-sequence, and hd-end-identifier are defined in §§; and new-line is defined in §§.

Constraints

No white space is permitted between the start identifier and its enclosing single quotes ('). See also §§.

Semantics:

A nowdoc string literal looks like a heredoc string literal (§§) except that in the former the start identifier name is enclosed in single quotes ('). The two forms of string literal have the same semantics and constraints except that a nowdoc string literal is not subject to variable substitution.

A nowdoc string literal is a c-constant (§§).

Examples

```
$v = 123;
$s = <<<    'ID'
S'o'me "\"t e\txt; \$v = $v"
Some more text
ID;
echo ">$s<\n\n";
→ >S'o'me "\"t e\txt; \$v = $v"
Some more text<
```

The Null Literal

There is one null-literal value, `NULL`. Its spelling is case-insensitive. (Note: Throughout this specification, the convention is to use all uppercase).

```
null-literal::
  NULL (written in any case combination)
```

A null-literal has the null type.

Operators and Punctuators

Syntax

```
operator-or-punctuator:: one of
  [   ]    (    )   {    }   .   ->   ++   --   **   *   +   -   ~   !
  $   /   % <<    >>   <   >   <=   >=   ==   ===   !=   !==   ^   |
  &   &&   ||   ?   :   ; =   **=   *=   /=   %=   +=   -=   .=   <<=
  >>=   &=   ^=   |=   ,
```

Semantics:

Operators and punctuators are symbols that have independent syntactic and semantic significance. Operators are used in expressions to describe operations involving one or more operands, and that yield a resulting value, produce a side effect, or some combination thereof. Punctuators are used for grouping and separating.

Expressions

General

An expression involves one or more terms and zero or more operators.

A full expression is an expression that is not part of another expression.

A side effect is an action that changes the state of the execution environment. (Examples of such actions are modifying a variable, writing to a device or file, or calling a function that performs such operations).

When an expression is evaluated, it produces a result. It might also produce a side effect. Only a few operators produce side effects. (For example, given the expression statement (§§) `$v = 10`; the expression 10 is evaluated to the result 10, and there is no side effect. Then the assignment operator is executed, which results in the side effect of `$v` being modified. The result of the whole expression is the value of `$v` after the assignment has taken place. However, that result is never used. Similarly, given the expression statement `++$v`; the expression is evaluated to the result incremented-value-of-`$v`, and the side effect is that `$v` is actually incremented. Again, the result is never used).

The occurrence of value computation and side effects is delimited by sequence points, places in a program's execution at *which all the computations and side effects previously promised are complete, and no computations or side effects of future operations have yet begun. There is a sequence point at the end of each full expression. The logical and (§§), logical or (§10.15), conditional (§10.15), and function-call (§§) operators each contain a sequence point. (For example, in the following series of expression statements, `$a = 10; ++$a; $b = $a;`, there is sequence point at the end of each full expression, so the assignment to $a is completed before `$a` is incremented, and the increment is completed before the assignment to `$b`).*

When an expression contains multiple operators, the precedence of those operators controls the order in which those operators are applied. (For example, the expression `$a - $b / $c` is evaluated as `$a - ($b / $c)` because the / operator has higher precedence than the binary - operator). The precedence of an operator is determined by the definition of its associated grammar production.

If an operand occurs between two operators having the same precedence, the order in which the operations are performed is determined by those operators' associativity. With left-associative operators, operations are performed left-to-right. (For example, `$a + $b - $c` is evaluated as `($a + $b) - $c`). With right-associative operators, operations are performed right-to-left. (For example, `$a = $b = $c` is evaluated as `$a = ($b = $c)`).

Precedence and associativity can be controlled using grouping parentheses. (For example, in the expression `($a - $b) / $c`, the subtraction is done before the division. Without the grouping parentheses, the division would take place first).

While precedence, associativity, and grouping parentheses control the order in which operators are applied, they do not control the order of evaluation of the terms themselves. Unless stated explicitly in this specification, the order in which the operands in an expression are evaluated relative to each other is unspecified. See the discussion above about the operators that contain sequence points. (For example, in the full expression `$list1[$i] = $list2[$i++]`, whether the value of `$i` on the left-hand side is the old or new `$i`, is unspecified. Similarly, in the full expression `$j = $i + $i++`, whether the value of `$i` is the old or new `$i`, is unspecified. Finally, in the full expression `f() + g() \* h()`, the order in which the three functions are called, is unspecified).

Implementation Notes

An expression that contains no side effects and whose resulting value is not used need not be evaluated. For example, the expression statements `6;, $i + 6;`, and `$i/$j`; are well formed, but they contain no side effects and their results are not used.

A side effect need not be executed if it can be determined that no other program code relies on its having happened. (For example, in the cases of return `$a++`; and return `++$a`;, it is obvious what value must be returned in each case, but if `$a` is a variable local to the enclosing function, `$a` need not actually be incremented.

Primary Expressions

General

Syntax

```
primary-expression:
  variable-name
  qualified-name
  literal
  const-expression
  intrinsic
  anonymous-function-creation-expression
  (  expression  )
  $this
```

variable-name and qualified-name are defined in §§; literal is defined in §§; const-expression is defined in §§; intrinsic is defined in §§; anonymous-function-creation-expression is defined in §§; and expression is defined in §§.

Semantics

When the name of a function is used as an expression without the function-call operator `()` (§§), that name is treated as a string containing that function's name.

The type and value of parenthesized expression are identical to those of the un-parenthesized expression.

The variable `$this` is predefined inside any instance method or constructor when that method is called from within an object context. `$this` is a handle (§§) that points to the calling object or to the object being constructed. The type of `$this` is the type of the class within which the usage of `$this` occurs. However, at run time, the type of the object referred to by `$this` may be the type of the enclosing class or any type derived from that class.

Intrinsics

General

Syntax

```
intrinsic:
  array-intrinsic
  echo-intrinsic
  empty-intrinsic
  eval-intrinsic
  exit-intrinsic
  isset-intrinsic
  list-intrinsic
  print-intrinsic
  unset-intrinsic
```

array-intrinsic is defined in §§; *echo-intrinsic* is defined in §§; *empty-intrinsic* is defined in §§; *eval-intrinsic* is defined in §§; *exit-intrinsic* is defined in §§; *isset-intrinsic* is defined in §§; *list-intrinsic* is defined in §§; *print-intrinsic* is defined in §§; and *unset-intrinsic* is defined in §§.

Semantics

The names in this series of subclauses have special meaning and are called intrinsics, but they are not keywords; nor are they functions.

array

Syntax

```
array-intrinsic:
  array ( array-initializer_opt  )
```

array-initializer is defined in §§.

Semantics

This intrinsic creates and initializes an array. It is equivalent to the array-creation operator `[]` (§§).

echo

Syntax

```
echo-intrinsic:
  echo  expression
  echo  (  expression  )
```

```
    echo  expression-list-two-or-more

  expression-list-two-or-more:
    expression  ,  expression
    expression-list-two-or-more  ,  expression
```

expression is defined in §§.

Constraints

expression must not designate an array nor an instance of a type not having a `__toString` method (§§).

Semantics

After converting each of its *expression*s' values to strings, if necessary, `echo` concatenates them in lexical order, and writes the resulting string to `STDOUT` (§§). Unlike `print` (§§), it does not produce a result.

For value substitution in string literals, see §§ and §§. For conversion to string, see §§.

Examples

```
$v1 = TRUE;
$v2 = 123;
echo  '>>' . $v1 . '|' . $v2 . "<<\n";    // outputs ">>1|123<<"
echo  '>>' , $v1 , '|' , $v2 , "<<\n";    // outputs ">>1|123<<"
echo ('>>' . $v1 . '|' . $v2 . "<<\n");   // outputs ">>1|123<<"
$v3 = "qqq{$v2}zzz";
echo "$v3\n";
```

empty

Syntax

```
  empty-intrinsic:
    empty ( expression  )
```

expression is defined in §§.

Semantics

This intrinsic returns `TRUE` if the variable or value designated by *expression* is empty, where *empty* means that the variable does not exist, or it exists and its value compares equal to `FALSE`. Otherwise, the intrinsic returns `FALSE`.

The following values are considered empty: `FALSE`, `0`, `0.0`, "", "`0`", `NULL`, `[]` / `array()`, and any uninitialized variable.

If this intrinsic is used with an expression that designate a dynamic property (§§), then if the class of that property has an `__isset` method (§§), that method is called.

Examples

```
empty("0")  // results in TRUE
empty("00") // results in FALSE
$v = [10, 20];
empty($v)   // results in FALSE
```

eval

Syntax

```
eval-intrinsic:
  eval (  expression  )
```

expression is defined in §§.

Constraints

expression must designate a string, the contents of which must be valid PHP source code.

The PHP source code in the string must not be delimited by opening and closing PHP tags.

Semantics

This intrinsic evaluates the contents of the string designated by *expression*, as PHP source code.

Execution of a `return` statement (§§) from within the source code terminates the intrinsic, and the value returned becomes the value returned by eval. If the source code is ill formed, eval returns `FALSE`; otherwise, eval returns `NULL`.

The source code is executed in the scope of that from which `eval` is called.

Examples

```
$str = "Hello";
eval("echo \$str . \"\\n\";");  // → echo $str . "\n";
```

exit/die

Syntax

```
exit-intrinsic:
  exit  expression_opt
  exit  (  expression_opt  )
  die   expression_opt
  die   (   expression_opt )
```

expression is defined in §§.

Constraints

When *expression* designates an integer, its value must be in the range 0–254.

Semantics

`exit` and `die` are equivalent.

This intrinsic terminates the current script. If *expression* designates a string, that string is written to `STDOUT` (§§). If *expression* designates an integer, that represents the script's *exit status code*. Code 255 is reserved by PHP. Code 0 represents "success". The exit status code is made available to the execution environment. If *expression* is omitted or is a string, the exit status code is zero. `exit` does not have a resulting value.

`exit` performs the following operations, in order:

- Writes the optional string to `STDOUT` (§§).
- Calls any functions registered via the library function `register_shutdown_function` (§xx) in their order of registration.
- Invokes destructors (§§) for all remaining instances.

Examples

```
exit ("Closing down");
exit (1);
exit;
```

isset

Syntax

```
  isset-intrinsic:
    isset  (  expression-list-one-or-more  )

  expression-list-one-or-more:
    expression
    expression-list-one-or-mor  ,  expression
```

expression is defined in §§.

Constraints

Each *expression* must designate a variable.

Semantics

This intrinsic returns `TRUE` if all the variables designated by *expression*s are set and their values are not `NULL`. Otherwise, it returns `FALSE`.

If this intrinsic is used with an expression that designates a dynamic property (§§), then if the class of that property has an `__isset` method (§§), that method is called.

Examples

```
$v = TRUE;
isset($v)     // results in TRUE
$v = NULL;
isset($v)     // results in FALSE
$v1 = TRUE; $v2 = 12.3; $v1 = NULL;
isset($v1, $v2, $v3)  // results in FALSE
```

list

Syntax

```
  list-intrinsic:
    list  (  list-expression-listopt  )

  list-expression-list:
  list-or-variable
  ,
  list-expression-list  ,  list-or-variableopt

  list-or-variable:
    list-intrinsic
    expression
```

expression is defined in §§.

Constraints

list-intrinsic must be used as the left-hand operand in a simple-assignment-expression (§§) of which the right-hand operand must be an expression that designates an array (the "source array").

Each expression in expression-list-one-or-more must designate a variable (the "target variable").

Semantics

This intrinsic assigns zero or more elements of the source array to the target variables. On success, it returns a copy of the source array. If the source array is actually the value `NULL`, this is consider a failure, and the return value from `list` is undefined.

All elements in the source array having keys of type `string` are ignored. The element having an `int` key of 0 is assigned to the first target variable, the element having an `int` key of 1 is assigned to the second target variable, and so on, until all target variables have been assigned. Any elements having an `int` key outside the range 0–(n-1), where n is the number of target variables, are ignored. If there are fewer element candidates having int keys than there are target variables, the unassigned target variables are unset (§§).

Any target variable may be a list, in which case, the corresponding element is expected to be an array.

If the source array elements and the target variables overlap in any way, the behavior is unspecified.

Examples

```
list($min, $max, $avg) = array(0, 100, 67);
  // $min is 0, $max is 100, $avg is 67
list($min, $max, $avg) = array(2 => 67, 1 => 100, 0 => 0);
  // same as example above
list($min, , $avg) = array(0, 100, 67);
  // $min is 0, $avg is 67
list($min, $max, $avg) = array(0, 2 => 100, 4 => 67);
  // $min is 0, $max is unset, $avg is 100
list($min, list($max, $avg)) = [0, [1 => 67, 99, 0 => 100], 33];
  // $min is 0, $max is 100, $avg is 67
```

print

Syntax

```
  print-intrinsic:
    print  expression
    print  (  expression  )
```

expression is defined in §§.

Constraints

expression must not designate an array or an instance of a type not having a `__toString` method.

Semantics

After converting its expression's value to a string, if necessary, `print` writes the resulting string to `STDOUT` (§§). Unlike `echo` (§§), `print` can be used in any context allowing an expression. It always returns the value 1.

For value substitution in string literals, see §§ and §§. For conversion to string, see §§.

Examples

```
$v1 = TRUE;
$v2 = 123;
```

```
print  '>>' . $v1 . '|' . $v2 . "<<\n";   // outputs ">>1|123<<"
print ('>>' . $v1 . '|' . $v2 . "<<\n");  // outputs ">>1|123<<"
$v3 = "qqq{$v2}zzz";
print "$v3\n";            // outputs "qqq123zzz"
$a > $b ? print "..." : print "...";
```

unset

Syntax

```
unset-intrinsic:
  unset  (  expression-list-one-or-more  )
```

expression-list-one-or-more is defined in §§.

Constraints

Each expression must designate a variable.

Semantics

This intrinsic unsets (§§) the variables designated by each expression in expression-list-one-or-more. No value is returned. An attempt to unset a non-existent variable (such as a non-existent element in an array) is ignored.

When called from inside a function, this intrinsic behaves, as follows:

- *For a variable declared `global` in that function, `unset` removes the alias to that variable from the scope of the current call to that function. Once the function returns, the global variable is still set. (To unset the global variable, use unset on the corresponding `$GLOBALS` array entry (§§)).*
- *For a variable passed byRef to that function, `unset` removes the alias to that variable from the scope of the current call to that function. Once the function returns, the passed-in argument variable is still set.*
- *For a variable declared static in that function, `unset` removes the alias to that variable from the scope of the current call to that function. In subsequent calls to that function, the static variable is still set and retains its value from call to call.*

Any visible instance property may be unset, in which case, the property is removed from that instance.

If this intrinsic is used with an expression that designate a dynamic property (§§), then if the class of that property has an `__unset` method (§§), that method is called.

Examples

```
unset($v);
unset($v1, $v2, $v3);
unset($x->m); // if m is a dynamic property, $x's __unset("m") is called
```

Anonymous Function-Creation

Syntax

```
anonymous-function-creation-expression:
function  &opt (  parameter-declaration-listopt  )  anonymous-function-use-clauseopt
    compound-statement

anonymous-function-use-clause:
  use  (  use-variable-name-list  )

use-variable-name-list:
  &opt   variable-name
```

```
    use-variable-name-list  ,  &opt  variable-name
```

parameter-declaration-list is defined in §§; compound-statement is defined in §§; variable-name is defined in §§.

Semantics

This operator returns an object of type `Closure` (§§), or a derived type thereof, that encapsulates the anonymous function (§§) defined within. An anonymous function is defined like, and behaves like, a named function (§§) except that the former has no name and has an optional anonymous-function-use-clause.

An expression that designates an anonymous function is compatible with the type hint `callable` (§§).

The use-variable-name-list is a list of variables from the enclosing scope, which are to be made available by name to the body of the anonymous function. Each of these may be passed by value or byRef, as needed. The values used for these variables are those at the time the `Closure` object is created, not when it is used to call the function it encapsulates.

An anonymous function defined inside an instance method has access to the variable `$this`.

Examples

```
function doit($value, callable $process)  // return type is "Closure"
{
  return $process($value);
}
$result = doit(5, function ($p) { return $p * 2; });  // doubles a value
$result = doit(5, function ($p) { return $p * $p; }); // squares a value
// -----------------------------------------
class C
{
  public function compute(array $values)
  {
    $count = 0;
          $callback1 = function () use (&$count) // called C::{closure}
    {
      ++$count;
      ...
    };
    ...
    $callback2 = function()   // also called C::{closure}
    {
      ...
    };
    ...
  }
  ...
}
```

Postfix Operators

General

Syntax

```
  postfix-expression:
    primary-expression
    clone-expression
    object-creation-expression
    array-creation-expression
    subscript-expression
```

```
    function-call-expression
    member-selection-expression
    postfix-increment-expression
    postfix-decrement-expression
    scope-resolution-expression
    exponentiation-expression
```

primary-expression is defined in §§; clone-expression is defined in §§; object-creation-expression is defined in §§; array-creation-expression is defined in §§; subscript-expression is defined in §§; function-call-expression is defined in §§; member-selection-expression is defined in §§; postfix-increment-expression and postfix-decrement-expression are defined in §§; scope-resolution-expression is defined in §§; and exponetiation-expression is defined in §§..

Semantics

These operators associate left-to-right.

The `clone` Operator

Syntax

```
  clone-expression:
    clone  expression
```

expression is defined in §§.

Constraints

expression must designate an object.

Semantics

The `clone` operator creates a new object that is a shallow copy of the object designated by *expression*. Then, if the class type of *expression* has a method called `__clone` (§§), that is called to perform a deep copy. The result is a handle that points to the new object.

Examples

Consider a class `Employee`, from which is derived a class `Manager`. Let us assume that both classes contain properties that are objects. clone is used to make a copy of a Manager object, and behind the scenes, the `Manager` object uses clone to copy the properties for the base class, `Employee`.

```
class Employee
{
  ...
  public function __clone()
  {
    // make a deep copy of Employee object
  }
}
class Manager extends Employee
{
  ...
  public function __clone()
  {
    $v = parent::__clone();
    // make a deep copy of Manager object

  }
}
$obj1 = new Manager("Smith", 23);
```

```
$obj2 = clone $obj1;  // creates a new Manager that is a deep copy
```

The `new` Operator

Syntax

```
  object-creation-expression:
    new  class-type-designator  (  argument-expression-list_opt  )
    new  class-type-designator

  class-type-designator:
    static
    qualified-name
    expression
```

argument-expression-list is defined in §§; qualified-name is defined in §§; and expression is defined in §§.

Constraints

qualified-name must name a class.

expression must be a value of type `string` (but not be a string literal) that contains the name of a class.

class-type-designator must not designate an abstract class (§§).

The number of arguments in argument-expression-list must be at least as many as the number of parameters defined for the class's constructor.

Semantics

The `new` operator allocates memory for an object that is an instance of the class specified by class-type-designator.

The object is initialized by calling the class's constructor (§§) passing it the optional argument-expression-list. If the class has no constructor, the constructor that class inherits (if any) is used. Otherwise, each instance property takes on the value `NULL`.

The result of an object-creation-expression is a handle to an object of the type specified by class-type-designator.

From within a method, the use of `static` corresponds to the class in the inheritance context in which the method is called.

Because a constructor call is a function call, the relevant parts of §§ also apply.

Examples

```
class Point
{
  public function __construct($x = 0, $y = 0)
  {
    ...
  }
  ...
}
$p1 = new Point;     // create Point(0, 0)
$p1 = new Point(12);   // create Point(12, 0)
$cName = 'Point';
$p1 = new $cName(-1, 1); // create Point(-1, 1)
```

Array Creation Operator

An array is created and initialized by one of two equivalent ways: via the array-creation operator `[]`, as described below, or *the intrinsic `array` (§§).*

Syntax

```
array-creation-expression:
  array  (  array-initializer_opt  )
  [ array-initializer_opt ]

array-initializer:
  array-initializer-list  ,opt

array-initializer-list:
  array-element-initializer
  array-element-initializer  ,  array-initializer-list

array-element-initializer:
  &opt   element-value
  element-key  =>  &opt   element-value

element-key:
  expression

element-value
  expression
```

expression is defined in §§.

Constraints

If array-element-initializer contains &, element-value's expression must be a variable name (§§).

Semantics

If array-initializer is omitted, the array has zero elements. For convenience, an array-initializer may have a trailing comma; however, this comma has no purpose. An array-initializer-list consists of a comma-separated list of one or more array-element-initializers, each of which is used to provide an element-value and an optional element-key.

If the value of element-key is neither `int` nor `string`, keys with `float` or `bool` values, or strings whose contents match exactly the pattern of decimal-literal (§§), are converted to `int` (§§), and values of all other key types are converted to `string` (§§).

If element-key is omitted from an array-element-initializer, an element key of type `int` is associated with the corresponding element-value. The key associated is one more than the previously assigned `int` key for this array, regardless of whether that key was provided explicitly or by default. However, if this is the first element with an `int` key, key zero is associated.

Once the element keys have been converted to `int` or `string`, and omitted element keys have each been associated by default, if two or more array-element-initializers in an array-initializer contain the same key, the lexically right-most one is the one whose element-value is used to initialize that element.

The result of this operator is a handle to the set of array elements.

If array-element-initializer contains &, element-value's value is stored using byRef assignment (§§).

Examples

```
$v = [];      // array has 0 elements
$v = array(TRUE);   // array has 1 element, the Boolean TRUE
$v = [123, -56];  // array of two ints, with implicit int keys 0 and 1
$v = [0 => 123, 1 => -56]; // array of two ints, with explicit int keys 0 and 1
$i = 10;
$v = [$i - 10 => 123, $i - 9 => -56]; // key can be a runtime expression
```

```
$v = [NULL, 1 => FALSE, 123, 3 => 34e12, "Hello"];  // implicit & explicit keys
$i = 6; $j = 12;
$v = [7 => 123, 3 => $i, 6 => ++$j];  // keys are in arbitrary order
$v[4] = 99;   // extends array with a new element
$v = [2 => 23, 1 => 10, 2 => 46, 1.9 => 6];
     // array has 2, with keys 2 and 1, values 46 and 6, respectively
$v = ["red" => 10, "4" => 3, 9.2 => 5, "12.8" => 111, NULL => 1];
     // array has 5 elements, with keys “red”, 4, 9, “12.8”, and “”.
$c = array("red", "white", "blue");
$v = array(10, $c, NULL, array(FALSE, NULL, $c));
$v = array(2 => TRUE, 0 => 123, 1 => 34.5, -1 => "red");
foreach($v as $e) { ... } // iterates over keys 2, 0, 1, -1
for ($i = -1; $i <= 2; ++$i) { ... $v[$i] } // retrieves via keys -1, 0, 1, 2
```

Subscript Operator

Syntax

```
subscript-expression:
  postfix-expression  [  expression_opt  ]
  postfix-expression  {  expression_opt  }   [Deprecated form]
```

postfix-expression is defined in §§; and expression is defined in §§.

Constraints

If postfix-expression designates a string, expression must not designate a string.

expression can be omitted only if subscript-expression is used in a modifiable-lvalue context and postfix-expression does not designate a string.

If subscript-expression is used in a non-lvalue context, the element being designated must exist.

Semantics

A subscript-expression designates a (possibly non-existent) element of an array or string. When subscript-expression designates an object of a type that implements `ArrayAccess` (§§), the minimal semantics are defined below; however, they can be augmented by that object's methods `offsetGet` (§15.6.1) and `offsetSet` (§15.6.1).

The element key is designated by expression. If the value of element-key is neither `int` nor `string`, keys with `float` or `bool` values, or strings whose contents match exactly the pattern of decimal-literal (§§), are converted to `int` (§§), and values of all other key types are converted to `string` (§§).

If both postfix-expression and expression designate strings, expression is treated as if it specified the `int` key zero instead.

A subscript-expression designates a modifiable lvalue if and only if postfix-expression designates a modifiable lvalue.

postfix-expression designates an array

If expression is present, if the designated element exists, the type and value of the result is the type and value of that element; otherwise, the result is `NULL`.

*If expression is omitted, a new element is inserted. Its key has type `int` and is one more than the highest, previously assigned, non-negative `int` key for this array. If this is the first element with a non-negative `int` key, key zero is used. However, if the highest, previously assigned `int` key for this array is `PHP_INT_MAX` (§§), **no new element is inserted**. The type and value of the result is the type and value of the new element.*

- *If the usage context is as the left-hand side of a simple-assignment-expression (§§): The value of the new element is the value of the right-hand side of that simple-assignment-expression.*

- *If the usage context is as the left-hand side of a compound-assignment-expression (§§): The expression `e1 op= e2` is evaluated as `e1 = NULL op (e2)`.*
- *If the usage context is as the operand of a postfix- or prefix-increment or decrement operator (§§, §§): The value of the new element is `NULL`.*

postfix-expression designates a string

If the designated element exists, the type and value of the result is the type and value of that element; otherwise, the result is an empty string.

postfix-expression designates an object of a type that implements `ArrayAccess`

If expression is present,

- *If subscript-expression is used in a non-lvalue context, the object's method `offsetGet` is called with an argument of expression. The type and value of the result is the type and value returned by `offsetGet`.*
- *If the usage context is as the left-hand side of a simple-assignment-expression: The object's method `offsetSet` is called with a first argument of expression and a second argument that is the value of the right-hand side of that simple-assignment-expression. The type and value of the result is the type and value of the right-hand side of that simple-assignment-expression.*
- *If the usage context is as the left-hand side of a compound-assignment-expression: The expression `e1 op= e2` is evaluated as `e1 = offsetGet(expression) op (e2)`, which is then processed according to the rules for simple assignment immediately above.*
- *If the usage context is as the operand of a postfix- or prefix-increment or decrement operator (§§, §§): The object's method `offsetGet` is called with an argument of expression. However, this method has no way of knowing if an increment or decrement operator was used, or whether it was a prefix or postfix operator. The type and value of the result is the type and value returned by `offsetGet`.*

If expression is omitted,

- *If the usage context is as the left-hand side of a simple-assignment-expression: The object's method `offsetSet` (§§) is called with a first argument of `NULL` and a second argument that is the value of the right-hand side of that simple-assignment-expression. The type and value of the result is the type and value of the right-hand side of that simple-assignment-expression.*
- *If the usage context is as the left-hand side of a compound-assignment-expression: The expression `e1 op= e2` is evaluated as `e1 = offsetGet(NULL) op (e2)`, which is then processed according to the rules for simple assignment immediately above.*
- *If the usage context is as the operand of a postfix- or prefix-increment or decrement operator (§§, §§): The object's method `offsetGet` is called with an argument of `NULL`. However, this method has no way of knowing if an increment or decrement operator was used, or whether it was a prefix or postfix operator. The type and value of the result is the type and value returned by `offsetGet`.*

Note: The brace (`{...}`) form of this operator has been deprecated.

Examples

```
$v = array(10, 20, 30);
$v[1] = 1.234;    // change the value (and type) of element [1]
$v[-10] = 19;   // insert a new element with int key -10
$v["red"] = TRUE; // insert a new element with string key "red"
[[2,4,6,8], [5,10], [100,200,300]][0][2]  // designates element with value 6
["black", "white", "yellow"][1][2]  // designates substring "i" in "white"
function f() { return [1000, 2000, 3000]; }
f()[2]      // designates element with value 3000
"red"[1.9]    // designates [1]
"red"[0][0][0]    // designates [0]
// -----------------------------------------
class MyVector implements ArrayAccess { ... }
```

```
$vect1 = new MyVector(array(10, 'A' => 2.3, "up"));
$vect1[10] = 987; // calls Vector::offsetSet(10, 987)
$vect1[] = "xxx"; // calls Vector::offsetSet(NULL, "xxx")
$x = $vect1[1];   // calls Vector::offsetGet(1)
```

Function Call Operator

Syntax

```
function-call-expression:
  postfix-expression  (  argument-expression-list_opt  )

argument-expression-list:
  assignment-expression
  argument-expression-list  ,  assignment-expression
```

postfix-expression is defined in §§; and assignmment-expression is defined in §§.

Constraints

postfix-expression must designate a function, either by being its name, by being a value of type string (but not a string literal) that contains the function's name, or by being a variable whose type is `Closure` (§§) or a derived type thereof.

The number of arguments present in a function call must be at least as many as the number of parameters defined for that function.

No calls can be made to a conditionally defined function (§§) until that function exists.

Any argument that matches a parameter passed byRef should (but need not) designate an lvalue.

Semantics

An expression of the form function-call-expression is a function call. The postfix expression designates the called function, and argument-expression-list specifies the arguments to be passed to that function. An argument can have any type. In a function call, postfix-expression is evaluated first, followed by each assignment-expression in the order left-to-right. There is a sequence point (§§) right before the function is called. For details of the type and value of a function call see §§. The value of a function call is a modifiable lvalue only if the function returns a byRef that aliases a modifiable lvalue.

When postfix-expression designates an instance method or constructor, the instance used in that designation is used as the value of `$this` in the invoked method or constructor. However, if no instance was used in that designation (for example, in the call `C::instance_method()`) the invoked instance has no `$this` defined.

When a function is called, the value of each argument passed to it is assigned to the corresponding parameter in that function's definition, if such a parameter exists. The assignment of argument values to parameters is defined in terms of simple (§§) or byRef assignment (§§), depending on how the parameter was declared. There may be more arguments than parameters, in which case, the library functions `func_num_args` (§xx), `func_get_arg` (§xx), and `func_get_args` (§xx) can be used to get access to the complete argument list that was passed. If the number of arguments present in a function call is fewer than the number of parameters defined for that function, any parameter not having a corresponding argument is considered undefined if it has no default argument value (§§); otherwise, it is considered defined with that default argument value.

If an undefined variable is passed using byRef, that variable becomes defined, with a default value of `NULL`.

Direct and indirect recursive function calls are permitted.

If postfix-expression is a string, this is a variable function call (§§).

Examples

```
function square($v) { return $v * $v; }
square(5)     // call square directly; it returns 25
$funct = square;  // assigns the string "square" to $funct
$funct(-2.3)    // call square indirectly; it returns 5.29
strlen($lastName) // returns the # of bytes in the string
// -----------------------------------------
function f1() { ... }  function f2() { ... }  function f3() { ... }
for ($i = 1; $i <= 2; ++$i) { $f = 'f' . $i;  $f(); }
// -----------------------------------------
function f($p1, $p2, $p3, $p4, $p5) { ... }
function g($p1, $p2, $p3, $p4, $p5) { ... }
function h($p1, $p2, $p3, $p4, $p5) { ... }
$funcTable = array(f, g, h);  // list of 3 function designators
$i = 1;
$funcTable[$i++]($i, ++$i, $i, $i = 12, --$i); // calls g(2,3,3,12,11)
// -----------------------------------------
function f4($p1, $p2 = 1.23, $p3 = "abc") { ... }
f4(); // inside f4, $p1 is undefined, $p2 is 1.23, $p3 is "abc"
// -----------------------------------------
function f(&$p) { ... }
$a = array(10, 20, 30);
f($a[5]); // non-existent element going in, but element exists afterwards
// -----------------------------------------
function factorial($int)  // contains a recursive call
{
  return ($int > 1) ? $int * factorial($int - 1) : $int;
}
// -----------------------------------------
$anon = function () { ... };  // store a Closure in $anon
$anon();  // call the anonymous function encapsulated by that object
```

Member-Selection Operator

Syntax

```
  member-selection-expression:
    postfix-expression  ->  member-selection-designator

  member-selection-designator:
    name
    expression
```

postfix-expression is defined in §§; name is defined in §§; and expression is defined in §§.

Constraints

postfix-expression must designate an object or be `NULL`, `FALSE`, or an empty string.

name must designate an instance property, or an instance or static method of postfix-expression's class type.

*expression must be a value of type `string` (but not a string literal) that contains the name of an instance property (**without** the leading `$`) or an instance or static method of that instance's class type.*

Semantics

A member-selection-expression designates an instance property or an instance or static method of the object designated by postfix-expression. For a property, the value is that of the property, and is a modifiable lvalue if postfix-expression is a modifiable lvalue.

When the `->` operator is used in a modifiable lvalue context and name or expression designate a property that is not visible, *the property is treated as a dynamic property (§§). If postfix-expression's class type defines a `__set` method (§§), it is called to store the property's value. When the `->` operator is used in a non-lvalue context and name or expression designate a property that is not visible, the property is treated as a dynamic property. If postfix-expression's class type defines a `__get` method (§§), it is called to retrieve the property's value.*

If postfix-expression is `NULL`, `FALSE`, or an empty string, an expression of the form `$p->x = 10` causes an instance of `stdClass` (§§) to be created with a dynamic property x having a value of 10. `$p` is then made to refer to this instance.

Examples

```
class Point
{
  private $x;
  private $y;
  public function move($x, $y)
  {
    $this->x = $x;  // sets private property $x
    $this->y = $y;  // sets private property $x
  }
  public function __toString()
  {
    return '(' . $this->x . ',' . $this->y . ')';
  }     // get private properties $x and $y
    public function __set($name, $value) { ... }
    public function __get($name) { ... }
}
$p1 = new Point;
$p1->move(3, 9);  // calls public instance method move by name
$n = "move";
$p1->$n(-2, 4);   // calls public instance method move by variable
$p1->color = "red"; // turned into $p1->__set("color", "red");
$c = $p1->color;  // turned into $c = $p1->__get("color");
```

Postfix Increment and Decrement Operators

Syntax

```
  postfix-increment-expression:
    unary-expression  ++

  postfix-decrement-expression:
    unary-expression  --
```

unary-expression is defined in §§.

Constraints

The operand of the postfix ++ and -- operators must be a modifiable lvalue that has scalar type.

Semantics

These operators behave like their prefix counterparts (§§) except that the value of a postfix ++ or -- expression is the value before any increment or decrement takes place.

Examples

```
$i = 10; $j = $i-- + 100;   // old value of $i (10) is added to 100
$a = array(100, 200); $v = $a[1]++; // old value of $ia[1] (200) is assigned
```

Scope-Resolution Operator

Syntax

```
scope-resolution-expression:
  scope-resolution-qualifier  ::  member-selection-designator
  scope-resolution-qualifier  ::  class

scope-resolution-qualifier:
  qualified-name
  expression
  self
  parent
  static
```

member-selection-designator is defined in §§.

Constraints

qualified-name must be the name of a class or interface type.

expression must be a value of type string (but not a string literal) that contains the name of a class or interface type.

Semantics

From inside or outside a class or interface, operator `::` allows the selection of a constant. From inside or outside a class, this operator allows the selection of a static property, static method, or instance method. From within a class, it also allows the selection of an overridden property or method. For a property, the value is that of the property, and is a modifiable lvalue if member-selection-designator is a modifiable lvalue.

From within a class, `self::m` refers to the member `m` in that class, whereas `parent::m` refers to the closest member `m` in the base-class hierarchy, not including the current class. From within a method, `static::m` refers to the member `m` in the class that corresponds to the class inheritance context in which the method is called. This allows late static binding. Consider the following scenario:

```
class Base
{
  public function b()
  {
    static::f();  // calls the most appropriate f()
  }
  public function f() { ... }
}
class Derived extends Base
{
  public function f() { ... }
}
$b1 = new Base;
$b1->b(); // as $b1 is an instance of Base, Base::b() calls Base::f()
$d1 = new Derived;
$d1->b(); // as $d1 is an instance of Derived, Base::b() calls Derived::f()
```

The value of the form of scope-resolution-expression ending in `::class` is a string containing the fully qualified name of the current class, which for a static qualifier, means the current class context.

Examples

```
final class MathLibrary
{
```

```
  public static function sin() { ... }
  ...
}
$v = MathLibrary::sin(2.34);  // call directly by class name
$clName = 'MathLibrary';
$v = $clName::sin(2.34);    // call indirectly via string
// -----------------------------------------
class MyRangeException extends Exception
{
  public function __construct($message, ...)
  {
    parent::__construct($message);
    ...
  }
  ...
}
// -----------------------------------------
class Point
{
  private static $pointCount = 0;
  public static function getPointCount()
  {
    return self::$pointCount;
  }
  ...
}
```

Exponentiation Operator

Syntax

```
  exponentiation-expression:
    expression  **  expression
```

expression is defined in §§.

Semantics

The `**` operator produces the result of raising the value of the left-hand operand to the power of the right-hand one. If either or both operands have non-numeric types, their values are converted to type `int` or `float`, as appropriate. If both operands have non-negative integer values and the result can be represented as an `int`, the result has type `int`; otherwise, the result has type `float`.

Examples

```
2**3;   // int with value 8
2**3.0;   // float with value 8.0
"2.0"**"3"; // float with value 8.0
```

Unary Operators

General

Syntax

```
  unary-expression:
    postfix-expression
```

```
    prefix-increment-expression
    prefix-decrement-expression
    unary-op-expression
    error-control-expression
    shell-command-expression
    cast-expression
    variable-name-creation-expression
```

postfix-expression is defined in §§; prefix-increment-expression and prefix-decrement-expression are defined in §§; unary-op-expression is defined in §§; error-control-expression is defined in §§; shell-command-expression is defined in §§; cast-expression is defined in §§ and variable-name-creation-expression is defined in §§.

Semantics

These operators associate right-to-left.

Prefix Increment and Decrement Operators

Syntax

```
  prefix-increment-expression:
    ++ unary-expression

  prefix-decrement-expression:
    -- unary-expression
```

unary-expression is defined in §§.

Constraints

The operand of the prefix `++` or `--` operator must be a modifiable lvalue that has scalar type.

Semantics

Arithmetic Operands

For a prefix `++` operator used with an arithmetic operand, the side effect (§§) of the operator is to increment by 1, as appropriate, the value of the operand. The result is the value of the operand after it has been incremented. If an int operand's value is the largest representable for that type, the type and value of the result is implementation-defined (§§).

For a prefix `--` operator used with an arithmetic operand, the side effect of the operator is to decrement by 1, as appropriate, the value of the operand. The result is the value of the operand after it has been decremented. If an int operand's value is the smallest representable for that type, the type and value of the result is implementation-defined (§§).

For a prefix `++` or `--` operator used with an operand having the value `INF`, `-INF`, or `NAN`, there is no side effect, and the result is the operand's value.

Boolean Operands

For a prefix `++` or `--` operator used with a Boolean-valued operand, there is no side effect, and the result is the operand's value.

NULL-valued Operands

For a prefix -- operator used with a `NULL`-valued operand, there is no side effect, and the result is the operand's value. For a prefix `++` operator used with a `NULL`-valued operand, the side effect is that the operand's type is changed to int, the operand's value is set to zero, and that value is incremented by 1. The result is the value of the operand after it has been incremented.

String Operands

For a prefix `--` operator used with an operand whose value is an empty string, the side effect is that the operand's type is changed to `int`, the operand's value is set to zero, and that value is decremented by 1. The result is the value of the operand after it has been incremented.

For a prefix `++` operator used with an operand whose value is an empty string, the side effect is that the operand's value is changed to the string "1". The type of the operand is unchanged. The result is the new value of the operand.

For a prefix `--` or `++` operator used with a numeric string, the numeric string is treated as the corresponding `int` or `float` value.

For a prefix `--` operator used with a non-numeric string-valued operand, there is no side effect, and the result is the operand's value.

For a non-numeric string-valued operand that contains only alphanumeric characters, for a prefix `++` operator, the operand is considered to be a pseudo-base-36 number (i.e., with digits 0–9 followed by A–Z or a–z) in which letter case is ignored for value purposes. The right-most digit is incremented by 1. For the digits 0–8, that means going to 1–9. For the letters "A"–"Y" (or "a"–"y"), that means going to "B"–"Z" (or "b"–"z"). For the digit 9, the digit becomes 0, and the carry is added to the next left-most digit, and so on. For the digit "Z" (or "z"), the resulting string has an extra digit "A" (or "a") appended. For example, when incrementing, "a" -> "b", "X" -> "AA", "AA" -> "AB", "F29" -> "F30", "FZ9" -> "GA0", and "ZZ9" -> "AAA0". A digit position containing a number wraps modulo-10, while a digit position containing a letter wraps modulo-26.

For a non-numeric string-valued operand that contains any non-alphanumeric characters, for a prefix `++` operator, all characters up to and including the right-most non-alphanumeric character is passed through to the resulting string, unchanged. Characters to the right of that right-most non-alphanumeric character are treated like a non-numeric string-valued operand that contains only alphanumeric characters, except that the resulting string will not be extended. Instead, a digit position containing a number wraps modulo-10, while a digit position containing a letter wraps modulo-26.

Examples

```
$i = 10; $j = --$i + 100;   // new value of $i (9) is added to 100
$a = array(100, 200); $v = ++$a[1]; // new value of $ia[1] (201) is assigned
```

Unary Arithmetic Operators

Syntax

```
  unary-op-expression:
    unary-operator cast-expression

  unary-operator: one of
    +  -  !  \
```

cast-expression is defined in §§.

Constraints

The operand of the unary `+`, unary `-`, and unary `!` operators must have scalar type.

The operand of the unary `~` operator must have arithmetic type.

Semantics

Arithmetic Operands

For a unary `+` operator used with an arithmetic operand, the type and value of the result is the type and value of the *operand.*

For a unary `-` operator used with an arithmetic operand, the value of the result is the negated value of the operand. However, if an int operand's original value is the smallest representable for that type, the type and value of the result is implementation-defined (§§).

For a unary `!` operator used with an arithmetic operand, the type of the result is `bool`. The value of the result is `TRUE` if the value of the operand is non-zero; otherwise, the value of the result is `FALSE`. For the purposes of this operator, `NAN` is considered a non-zero value. The expression `!E` is equivalent to `(E == 0)`.

For a unary `~` operator used with an `int` operand, the type of the result is `int`. The value of the result is the bitwise complement of the value of the operand (that is, each bit in the result is set if and only if the corresponding bit in the operand is clear). For a unary `~` operator used with a `float` operand, the value of the operand is first converted to `int` before the bitwise complement is computed.

Boolean Operands

For a unary `+` operator used with a `TRUE`-valued operand, the value of the result is 1 and the type is `int`. When used with a `FALSE`-valued operand, the value of the result is zero and the type is `int`.

For a unary `-` operator used with a `TRUE`-valued operand, the value of the result is -1 and the type is `int`. When used with a `FALSE`-valued operand, the value of the result is zero and the type is `int`.

For a unary `!` operator used with a `TRUE`-valued operand, the value of the result is `FALSE` and the type is `bool`. When used with a `FALSE`-valued operand, the value of the result is `TRUE` and the type is `bool`.

NULL-valued Operands

For a unary `+` or unary `-` operator used with a `NULL`-valued operand, the value of the result is zero and the type is `int`.

For a unary `!` operator used with a `NULL`-valued operand, the value of the result is `TRUE` and the type is `bool`.

String Operands

For a unary `+` or `-` operator used with a numeric string or a leading-numeric string, the string is first converted to an `int` or `float`, as appropriate, after which it is handled as an arithmetic operand. The trailing non-numeric characters in leading-numeric strings are ignored. With a non-numeric string, the result has type `int` and value 0.

For a unary `!` operator used with a string, the string is first converted to `bool`, after which its value is negated.

Examples

```
$v = +10;
if ($v1 > -5) ...
$t = TRUE;
if (!$t) ...
$v = ~0b1010101;
```

Error Control Operator

Syntax

```
  error-control-expression:
    @   expression
```

expression is defined in §§.

Semantics

Operator `@` suppresses any error messages generated by the evaluation of expression.

If a custom error-handler has been established using the library function `set_error_handler` (§xx), that handler is still called.

Examples

```
$infile = @fopen("NoSuchFile.txt", 'r');
```

On open failure, the value returned by `fopen` is `FALSE`, which is sufficient to know to handle the error. There is no need to have any error message displayed.

Implementation Notes

Given the following example:

```
function f() {
  $ret = $y;
  return $ret;
}

$x = @f();  // without @, get "Undefined variable: y"
```

The following code shows how this statement is handled:

```
$origER = error_reporting();
error_reporting(0);
$tmp = f();
$curER = error_reporting();
if ($curER === 0) error_reporting($origER);
$x = $tmp;
```

Shell Command Operator

Syntax

```
  shell-command-expression:
    `  dq-char-sequence_opt  `
```

where ` is the GRAVE ACCENT character U+0060, commonly referred to as a backtick.

dq-char-sequence is described in §§.

Semantics

This operator passes dq-char-sequence to the command shell for execution, as though it was being passed to the library function `shell_exec` (§xx). If the output from execution of that command is written to `STDOUT` (§§), that output is the result of this operator as a string. If the output is redirected away from `STDOUT`, or dq-char-sequence is empty or contains only white space, the result of the operator is `NULL`.

If `shell_exec` (§xx) is disabled, this operator is disabled.

Examples

```
$result = `ls`;           // result is the output of command ls
$result = `ls >dirlist.txt`;  // result is NULL
$d = "dir"; $f = "*.*";
$result = `$d {$f}`;      // result is the output of command dir *.*
```

Cast Operator

Syntax

```
cast-expression:
  unary-expression
  (  cast-type  ) cast-expression

cast-type: one of
  array  binary  bool  boolean  double  int  integer  float  object
  real  string  unset
```

unary-expression is defined in §§.

Constraints

For binary, cast-expression must designate a string.

Semantics

With the exception of the cast-types unset and binary (see below), the value of the operand cast-expression is converted to the type specified by cast-type, and that is the type and value of the result. This construct is referred to a cast, and is used as the verb, "to cast". If no conversion is involved, the type and value of the result are the same as those of cast-expression.

A cast can result in a loss of information.

A cast-type of `array` results in a conversion to type array. See §§ for details.

A cast-type of `binary` is reserved for future use in dealing with so-called binary strings. Casting a string to binary results in the same string.

A cast-type of `bool` or `boolean` results in a conversion to type `bool`. See §§ for details.

A cast-type of `int` or `integer` results in a conversion to type `int`. See §§ for details.

A cast-type of `float`, `double`, or `real` results in a conversion to type `float`. See §§ for details.

A cast-type of `object` results in a conversion to type `object`. See §§ for details.

A cast-type of `string` results in a conversion to type `string`. See §§ for details.

A cast-type of `unset` always results in a value of `NULL`. (This use of `unset` should not be confused with the `unset` intrinsic (§§)).

Examples

```
(int)(10/3)          // results in the int 3 rather than the float 3.333...
(array)(16.5)      // results in an array of 1 float; [0] = 16.5
(int)(float)"123.87E3" // results in the int 123870
```

Variable-Name Creation Operator

Syntax

```
variable-name-creation-expression:
  $   expression
  $  {  expression  }
```

expression is defined in §§.

Constraints

In the non-brace form, expression must be a variable-name-creation-expression or a variable-name that designates a scalar value.

In the brace form, expression must be a variable-name-creation-expression or an expression that designates a scalar value.

Semantics

The result of this operator is a variable name spelled using the textual representation of the value of expression even though such a name might not be permitted as a variable-name (§§) source code token.

This specification documents existing practice rather than ideal language design, and ***there is one aspect of this operator that behaves in a manner that violates the precedence rules****. Consider `o` to be an object of some class that has an instance property called `pr`. How is the non-brace-form expression `$$o->pr` handled with respect to precedence? As the operator `->` has higher precedence, the answer would seem to be, "`->` wins over `$`"; however, that is not the case. In fact, the expression is treated as `${$o}->pr`.*

Examples

```
$color = "red";
$$color = 123;    // equivalent to $red = 123
// -----------------------------------------
$x = 'ab'; $ab = 'fg'; $fg = 'xy';
$$ $ $x = 'Hello';  // equivalent to $xy = Hello
// -----------------------------------------
$v1 = 3;
$$v1 = 22;        // equivalent to ${3} = 22
$v2 = 9.543;
$$v2 = TRUE;    // equivalent to ${9.543} = TRUE
$v3 = NULL;
$$v3 = "abc";   // equivalent to ${NULL} = "abc"
// -----------------------------------------
function f1 () { return 2.5; }
${1 + f1()} = 1000;   // equivalent to ${3.5} = 1000
// -----------------------------------------
$v = array(10, 20); $a = 'v';
$$a[0] = 5;       // [] has higher precedence than $
$v = array(10, 20); $a = 'v';
${$a[0]} = 5;   // equivalent to above
$v = array(10, 20); $a = 'v';
${$a}[0] = 5;   // $ gets first shot at $a
```

`instanceof` Operator

Syntax

```
  instanceof-expression:
    unary-expression
    instanceof-subject  instanceof   instanceof-type-designator

  instanceof-subject:
    expression

  instanceof-type-designator:
    qualified-name
    expression
```

unary-expression is defined in §§; *expression* is defined in §§; and *qualified-name* is defined in §§.

Constraints

The *expression* in *instanceof-subject* must designate a variable.

The *expression* in *instanceof-type-designator* must not be any form of literal.

qualified-name must be the name of a class or interface type.

Semantics

Operator `instanceof` returns `TRUE` if the variable designated by *expression* in *instanceof-subject* is an object having type *qualified-name*, is an object whose type is derived from type *qualified-name*, or is an object whose type implements interface *qualified-name*. Otherwise, it returns `FALSE`. When the *expression* form of *instanceof-type-designator* is used, *expression* may be a string that contains a class or interface name. Alternatively, *expression* can designate an instance variable, in which case, operator `instanceof` returns `TRUE` if the variable designated by the left-hand *expression* is an instance of the `class` type, or of a derived type, of the right-hand *expression*.

If either *expression* is not an instance, `FALSE` is returned.

Note: This operator supersedes the library function `is_a` (§xx), which has been deprecated.

Examples

```
class C1 { ... } $c1 = new C1;
class C2 { ... } $c2 = new C2;
class D extends C1 { ... } $d = new D;
$d instanceof C1      // TRUE
$d instanceof C2      // FALSE
$d instanceof D       // TRUE
// -----------------------------------------
interface I1 { ... }
interface I2 { ... }
class E1 implements I1, I2 { ... }
$e1 = new E1;
$e1 instanceof I1       // TRUE
$iName = "I2";
$e1 instanceof $iName     // TRUE
```

Multiplicative Operators

Syntax

```
  multiplicative-expression:
    instanceof-expression
    multiplicative-expression  *  multiplicative-expression
    multiplicative-expression  /  multiplicative-expression
    multiplicative-expression  %  multiplicative-expression
```

instanceof-expression is defined in §§.

Constraints

The right-hand operand of operator `/` and operator `%` must not be zero.

Semantics

The binary `*` operator produces the product of its operands. If either or both operands have non-numeric types, their values are converted to type `int` or `float`, as appropriate. Then if either operand has type `float`, the other is converted to that type, and the result has type `float`. Otherwise, both operands have type `int`, in which case, if the resulting value can be represented in type `int` that is the result type. Otherwise, the type and value of the result is implementation-defined (§§).

Division by zero results in a diagnostic followed by a `bool` result having value `FALSE`. (The values +/- infinity and NaN cannot be generated via this operator; instead, use the predefined constants `INF` and `NAN`).

The binary `/` operator produces the quotient from dividing the left-hand operand by the right-hand one. If either or both operands have non-numeric types, their values are converted to type `int` or `float`, as appropriate. Then if either operand has type `float`, the other is converted to that type, and the result has type `float`. Otherwise, both operands have type `int`, in which case, if the mathematical value of the computation can be preserved using type `int`, that is the result type; otherwise, the type of the result is `float`.

The binary `%` operator produces the remainder from dividing the left-hand operand by the right-hand one. If the type of both operands is not `int`, their values are converted to that type. The result has type `int`.

These operators associate left-to-right.

Examples

```
-10 * 100;       // int with value -1000
100 * -3.4e10;   // float with value -3400000000000
"123" * "2e+5;   // float with value 24600000
100 / 100;       // int with value 1
100  / "123";    // float with value 0.8130081300813
"123" % 100;     // int with value 23
```

Additive Operators

Syntax

```
  additive-expression:
    multiplicative-expression
    additive-expression  +  multiplicative-expression
    additive-expression  -  multiplicative-expression
    additive-expression  .  multiplicative-expression
```

multiplicative-expression is defined in §§.

Constraints

If either operand has array type, the other operand must also have array type.

Semantics

For non-array operands, the binary `+` operator produces the sum of those operands, while the binary `-` operator produces the difference of its operands when subtracting the right-hand operand from the left-hand one. If either or both operands have non-array, non-numeric types, their values are converted to type `int` or `float`, as appropriate. Then if either operand has type `float`, the other is converted to that type, and the result has type `float`. Otherwise, both operands have type `int`, in which case, if the resulting value can be represented in type `int` that is the result type. Otherwise, the type and value of the result is implementation-defined (§§).

If both operands have array type, the binary `+` operator produces a new array that is the union of the two operands. The result is a copy of the left-hand array with elements inserted at its end, in order, for each element in the right-hand array whose key does not already exist in the left-hand array. Any element in the right-hand array whose key exists in the left-hand *array is ignored.*

The binary `.` operator creates a string that is the concatenation of the left-hand operand and the right-hand operand, in that order. If either or both operands have types other than `string`, their values are converted to type `string`. The result has type `string`.

These operators associate left-to-right.

Examples

```
-10 + 100;        // int with value 90
100 + -3.4e10;    // float with value -33999999900
"123" + "2e+5";   // float with value 200123
100 - "123";      // int with value 23
-3.4e10 - "abc";  // float with value -34000000000
// ----------------------------------------
[1, 5 => FALSE, "red"] + [4 => -5, 1.23]; // [1, 5 => FALSE, "red", 4 => -5]
  // dupe key 5 (value 1.23) is ignored
[NULL] + [1, 5 => FALSE, "red"];          // [NULL, 5 => FALSE, "red"]
  // dupe key 0 (value 1) is ignored
[4 => -5, 1.23] + [NULL];                 // [4 => -5, 1.23, 0 => NULL]
// ----------------------------------------
-10 . NAN;        // string with value "-10NAN"
INF . "2e+5";     // string with value "INF2e+5"
TRUE . NULL;      // string with value "1"
10 + 5 . 12 . 100 - 50;  // int with value 1512050; ((((10 + 5).12).100)-50)
```

Bitwise Shift Operators

Syntax

```
  shift-expression:
    additive-expression
    shift-expression  <<  additive-expression
    shift-expression  >>  additive-expression
```

additive-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

Given the expression `e1 << e2`, the bits in the value of `e1` are shifted left by `e2` positions. Bits shifted off the left end are discarded, and zero bits are shifted on from the right end. Given the expression `e1 >> e2`, the bits in the value of `e1` are shifted right by `e2` positions. Bits shifted off the right end are discarded, and the sign bit is propagated from the left end.

If either operand does not have type `int`, its value is first converted to that type.

The type of the result is `int`, and the value of the result is that after the shifting is complete. The values of `e1` and `e2` are unchanged.

If the shift count is negative, the actual shift applied is `n - (-shift count % n)`, where `n` is the number of bits per `int`. If the shift count is greater than the number of bits in an `int`, the actual shift applied is shift count `% n`.

These operators associate left-to-right.

Examples

```
1000 >> 2   // 3E8 is shifted right 2 places
-1000 << 2      // FFFFFC18 is shifted left 5 places
123 >> 128      // adjusted shift count = 0
123 << 33   // For a 32-bit int, adjusted shift count = 1; otherwise, 33
```

Relational Operators

Syntax

```
  relational-expression:
    shift-expression
    relational-expression  <   shift-expression
    relational-expression  >   shift-expression
    relational-expression  <=  shift-expression
    relational-expression  >=  shift-expression
```

shift-expression is defined in §§.

Semantics

Operator `<` represents less-than, operator `>` represents greater-than, operator `<=` represents less-than-or-equal-to, and operator `>=` represents greater-than-or-equal-to.

The type of the result is `bool`.

The operands are processed using the following steps, in order:

1. If either operand has the value `NULL`, then if the other operand has type string, the `NULL` is converted to the empty string (""); otherwise, the `NULL` is converted to type `bool`.
2. If both operands are non-numeric strings or one is a numeric string and the other a leading-numeric string, the result is the lexical comparison of the two operands. Specifically, the strings are compared byte-by-byte starting with their first byte. If the two bytes compare equal and there are no more bytes in either string, the strings are equal and the comparison ends; otherwise, if this is the final byte in one string, the shorter string compares less-than the longer string and the comparison ends. If the two bytes compare unequal, the string having the lower-valued byte compares less-than the other string, and the comparison ends. If there are more bytes in the strings, the process is repeated for the next pair of bytes.
3. If either operand has type `bool`, the other operand is converted to that type. The result is the logical comparison of the two operands after conversion, where `FALSE` is defined to be less than `TRUE`.
4. If the operands both have arithmetic type, string type, or are resources, they are converted to the corresponding arithmetic type (§§ and §§). The result is the numerical comparison of the two operands after conversion.
5. If both operands have array type, if the arrays have different numbers of elements, the one with the fewer is considered less-than the other one—regardless of the keys and values in each—, and the comparison ends. For arrays having the same numbers of elements, if the next key in the left-hand operand exists in the right-hand operand, the corresponding values are compared. If they are unequal, the array containing the lesser value is considered less-than the other one, and the comparison ends; otherwise, the process is repeated with the next element. If the next key in the left-hand operand does not exist in the right-hand operand, the arrays cannot be compared and `FALSE` is returned. For array comparison, the order of insertion of the elements into those arrays is irrelevant.
6. If only one operand has object type, that compares greater-than any other operand type.
7. If only one operand has array type, that compares greater-than any other operand type.
8. If the operands have different object types, the result is always `FALSE`.
9. If the operands have the same object type, the result is determined by comparing the lexically first-declared instance property in each object. If those properties have object type, the comparison is applied recursively.

These operators associate left-to-right.

Examples

```
"" < "ab"       // result has value TRUE
"a" > "A"       // result has value TRUE
"a0" < "ab"     // result has value TRUE
"aA <= "abc"    // result has value TRUE
// ----------------------------------------
NULL < [10,2.3] // result has value TRUE
TRUE > -3.4     // result has value FALSE
TRUE < -3.4     // result has value FALSE
TRUE >= -3.4    // result has value TRUE
FALSE < "abc"   // result has value TRUE
// ----------------------------------------
10 <= 0         // result has value FALSE
10 >= "-3.4"    // result has value TRUE
"-5.1" > 0      // result has value FALSE
// ----------------------------------------
[100] < [10,20,30] // result has value TRUE (LHS array is shorter)
[10,20] >= ["red"=>0,"green"=>0] // result has value FALSE, (key 10 does not exists in RHS)
["red"=>0,"green"=>0] >= ["green"=>0,"red"=>0] // result has value TRUE (order is irrelevant)
```

Equality Operators

Syntax

```
  equality-expression:
    relational-expression
    equality-expression  ==  relational-expression
    equality-expression  !=  relational-expression
    equality-expression  <>  relational-expression
    equality-expression  ===  relational-expression
    equality-expression  !==  relational-expression
```

relational-expression is defined in §§.

Semantics

Operator `==` represents value-equality, operators `!=` and `<>` are equivalent and represent value-inequality, operator `===` represents same-type-and-value-equality, and operator `!==` represents not-same-type-and-value-equality. However, when comparing two objects, operator `===` represents identity and operator `!==` represents non-identity. Specifically, in this context, these operators check to see if the two operands are the exact same object, not two different objects of the same type and value.

The type of the result is `bool`.

The operands are processed using the following steps, in order:

1. *For operators `==`, `!=`, and `<>`, if either operand has the value `NULL`, then if the other operand has type string, the `NULL` is converted to the empty string (""); otherwise, the `NULL` is converted to type bool.*
2. *If both operands are non-numeric strings or one is a numeric string and the other a leading-numeric string, the result is the lexical comparison of the two operands. Specifically, the strings are compared byte-by-byte starting with their first byte. If the two bytes compare equal and there are no more bytes in either string, the strings are equal and the comparison ends; otherwise, if this is the final byte in one string, the shorter string compares less-than the longer string and the comparison ends. If the two bytes compare unequal, the string having the lower-valued byte compares less-than the other string, and the comparison ends. If there are more bytes in the strings, the process is repeated for the next pair of bytes.*
3. *If either operand has type bool, for operators `==`, `!=`, and `<>`, the other operand is converted to that type. The result is*

the logical comparison of the two operands after any conversion, where `FALSE` is defined to be less than `TRUE`.

4. If the operands both have arithmetic type, string type, or are resources, for operators `==`, `!=`, and `<>`, they are converted to the corresponding arithmetic type (§§ and §§). The result is the numerical comparison of the two operands after any conversion.
5. If both operands have array type, for operators `==`, `!=`, and `<>`, the arrays are equal if they have the same set of key/value pairs, after element type conversion, without regard to the order of insertion of their elements. For operators `===` and `!==` the arrays are equal if they have the same set of key/value pairs, the corresponding values have the same type, and the order of insertion of their elements are the same.
6. If only one operand has object type, the two operands are never equal.
7. If only one operand has array type, the two operands are never equal.
8. If the operands have different object types, the two operands are never equal.
9. If the operands have the same object type, the two operands are equal if the instance properties in each object have the same values. Otherwise, the objects are unequal. The instance properties are compared, one at a time, in the lexical order of their declaration. For properties that have object type, the comparison is applied recursively.

These operators associate left-to-right.

Examples

```
"a" <> "aa" // result has value TRUE
// ----------------------------------------
NULL == 0   // result has value TRUE
NULL === 0  // result has value FALSE
TRUE != 100  // result has value FALSE
TRUE !== 100  // result has value TRUE
// ----------------------------------------
"10" != 10  // result has value FALSE
"10" !== 10 // result has value TRUE
// ----------------------------------------
[10,20] == [10,20.0]  // result has value TRUE
[10,20] === [10,20.0] // result has value FALSE
["red"=>0,"green"=>0] === ["red"=>0,"green"=>0] // result has value TRUE
["red"=>0,"green"=>0] === ["green"=>0,"red"=>0] // result has value FALSE
```

Bitwise AND Operator

Syntax

```
  bitwise-AND-expression:
    equality-expression
    bit-wise-AND-expression  &  equality-expression
```

equality-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type `int`, its value is first converted to that type.

The result of this operator is the bitwise-AND of the two operands, and the type of that result is `int`.

This operator associates left-to-right.

Examples

```
0b101111 & 0b101          // 0b101
$lLetter = 0x73;          // letter 's'
$uLetter = $lLetter & ~0x20;  // clear the 6th bit to make letter 'S'
```

Bitwise Exclusive OR Operator

Syntax

```
  bitwise-exc-OR-expression:
    bitwise-AND-expression
    bitwise-exc-OR-expression  ^   bitwise-AND-expression
```

bitwise-AND-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type `int`, its value is first converted to that type.

The result of this operator is the bitwise exclusive-OR of the two operands, and the type of that result is `int`.

This operator associates left-to-right.

Examples

```
0b101111 ^ 0b101    // 0b101010
$v1 = 1234; $v2 = -987; // swap two integers having different values
$v1 = $v1 ^ $v2;
$v2 = $v1 ^ $v2;
$v1 = $v1 ^ $v2;    // $v1 is now -987, and $v2 is now 1234
```

Bitwise Inclusive OR Operator

Syntax

```
  bitwise-inc-OR-expression:
    bitwise-exc-OR-expression
    bitwise-inc-OR-expression  |  bitwise-exc-OR-expression
```

bitwise-exc-OR-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type `int`, its value is first converted to that type.

The result of this operator is the bitwise inclusive-OR of the two operands, and the type of that result is `int`.

This operator associates left-to-right.

Examples

```
0b101111 | 0b101      // 0b101111
$uLetter = 0x41;      // letter 'A'
$lLetter = $upCaseLetter | 0x20;  // set the 6th bit to make letter 'a'
```

Logical AND Operator (form 1)

Syntax

```
  logical-AND-expression-1:
    bitwise-incl-OR-expression
    logical-AND-expression-1  &&  bitwise-inc-OR-expression
```

bitwise-incl-OR-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type bool, its value is first converted to that type.

Given the expression `e1 && e2, e1` is evaluated first. If `e1` is `FALSE`, `e2` is not evaluated, and the result has type `bool`, value `FALSE`. Otherwise, `e2` is evaluated. If `e2` is `FALSE`, the result has type bool, value `FALSE`; otherwise, it has type `bool`, value `TRUE`. There is a sequence point after the evaluation of `e1`.

This operator associates left-to-right.

Except for the difference in precedence, operator `&&` has exactly the same semantics as operator `and` (§§).

Examples

```
if ($month > 1 && $month <= 12) ...
```

Logical Inclusive OR Operator (form 1)

Syntax

```
  logical-inc-OR-expression-1:
    logical-AND-expression-1
    logical-inc-OR-expression-1  ||  logical-AND-expression-1
```

logical-exc-OR-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type bool, its value is first converted to that type.

Given the expression `e1 || e2`, `e1` is evaluated first. If `e1` is TRUE, `e2` is not evaluated, and the result has type `bool`, *value `TRUE`. Otherwise, `e2` is evaluated. If `e2` is `TRUE`, the result has type `bool`, value `TRUE`; otherwise, it has type `bool`, value `FALSE`. There is a sequence point after the evaluation of `e1`.*

This operator associates left-to-right.

Examples

```
if ($month < 1 || $month > 12) ...
```

Conditional Operator

Syntax

```
  conditional-expression:
    logical-inc-OR-expression-1
    logical-inc-OR-expression-1  ?  expression_opt  :  conditional-expression
```

logical-OR-expression is defined in §§; and expression is defined in §§.

Constraints

The first operand must have scalar type.

Semantics

Given the expression `e1 ? e2 : e3`, if `e1` is `TRUE`, then and only then is `e2` evaluated, and the result and its type become the result and type of the whole expression. Otherwise, then and only then is `e3` evaluated, and the result and its type become the result and type of the whole expression. There is a sequence point after the evaluation of `e1`. If `e2` is omitted, the result and type of the whole expression is the value and type of `e1` when it was tested.

This operator associates left-to-right.

Examples

```
for ($i = -5; $i <= 5; ++$i)
  echo "$i is ".(($i & 1 == TRUE) ? "odd\n" : "even\n");
// ----------------------------------------
$a = 10 ? : "Hello";  // result is int with value 10
$a = 0 ? : "Hello";     // result is string with value "Hello"
$i = PHP_INT_MAX;
$a = $i++ ? : "red";  // result is int with value 2147483647 (on a 32-bit
                // system) even though $i is now the float 2147483648.0
// ----------------------------------------
$i++ ? f($i) : f(++$i); // the sequence point makes this well-defined
// ----------------------------------------
function factorial($int)
{
  return ($int > 1) ? $int * factorial($int - 1) : $int;
}
```

Assignment Operators

General

Syntax

```
assignment-expression:
  conditional-expression
  simple-assignment-expression
  byref-assignment-expression
  compound-assignment-expression
```

conditional-expression is defined in §§; *simple-assignment-expression* is defined in §§; *byref-assignment-expression* is defined in §§; and *compound-assignment-expression* is defined in §§.

Constraints

The left-hand operand of an assignment operator must be a modifiable lvalue.

Semantics

These operators associate right-to-left.

Simple Assignment

Syntax

```
simple-assignment-expression:
  unary-expression  =  assignment-expression
```

unary-expression is defined in §§; *assignment-expression* is defined in §§.

Constraints

If the location designated by the left-hand operand is a string element, the key must not be a negative-valued `int`, and the right-hand operand must have type `string`.

Semantics

If *assignment-expression* designates an expression having value type, see §§. If *assignment-expression* designates an expression having handle type, see §§. If *assignment-expression* designates an expression having array type, see §§.

The type and value of the result is the type and value of the left-hand operand after the store (if any [see below]) has taken place. The result is not an lvalue.

If the location designated by the left-hand operand is a non-existent array element, a new element is inserted with the designated key and with a value being that of the right-hand operand.

If the location designated by the left-hand operand is a string element, then if the key is a negative-valued `int`, there is no side effect. Otherwise, if the key is a non-negative-valued `int`, the left-most single character from the right-hand operand is stored at the designated location; all other characters in the right-hand operand string are ignored. If the designated location is beyond the end of the destination string, that string is extended to the new length with spaces (U+0020) added as padding beyond the old end and before the newly added character. If the right-hand operand is an empty string, the null character \0 (U+0000) is stored.

Examples

```
$a = $b = 10    // equivalent to $a = ($b = 10)
$v = array(10, 20, 30);
$v[1] = 1.234;    // change the value (and type) of an existing element
$v[-10] = 19;   // insert a new element with int key -10
$v["red"] = TRUE; // insert a new element with string key "red"
$s = "red";
$s[1] = "X";    // OK; "e" -> "X"
```

```
$s[-5] = "Y";   // warning; string unchanged
$s[5] = "Z";    // extends string with "Z", padding with spaces in [3]-[5]
$s = "red";
$s[0] = "DEF";    // "r" -> "D"; only 1 char changed; "EF" ignored
$s[0] = "";       // "D" -> "\0"
$s["zz"] = "Q";   // warning; defaults to [0], and "Q" is stored there
// -----------------------------------------
class C { ... }
$a = new C; // make $a point to the allocated object
```

byRef Assignment

Syntax

```
  byref-assignment-expression:
    unary-expression  =  &  assignment-expression
```

unary-expression is defined in §§; assignment-expression is defined in §§.

Constraints

unary-expression must be a variable name.

assignment-expression must be an lvalue, a call to a function that returns a value byRef, or a new-expression (see comment below regarding this).

Semantics

unary-expression becomes an alias for assignment-expression. If assignment-expression designates an expression having value type, see §§. If assignment-expression designates an expression having handle type, see §§. If assignment-expression designates an expression having array type, see §§.

Examples

```
$a = 10;
$b =& $a;   // make $b an alias of $a
++$a;       // increment $a/$b to 11
$b = -12;   // sets $a/$b to -12
$a = "abc";     // sets $a/$b to "abc"
unset($b);      // removes $b's alias to $a
// -----------------------------------------
function &g2() { $t = "xxx"; return $t; } // return byRef
$b =& g2();     // make $b an alias to "xxx"
```

Compound Assignment

Syntax

```
  compound-assignment-expression:
    unary-expression   compound-assignment-operator   assignment-expression

  compound-assignment-operator: one of
    **=  *=  /=  %=  +=  -=  .=  <<=  >>=  &=  ^=  |=
```

unary-expression is defined in §§; assignment-expression is defined in §§.

Constraints

Any constraints that apply to the corresponding postfix or binary operator apply to the compound-assignment form as well.

Semantics

The expression `e1 op= e2` is equivalent to `e1 = e1 op (e2)`, except that `e1` is evaluated once only.

Examples

```
$v = 10;
$v += 20;   // $v = 30
$v -= 5;    // $v = 25
$v .= 123.45  // $v = "25123.45"
$a = [100, 200, 300];
$i = 1;
$a[$i++] += 50; // $a[1] = 250, $i → 2
```

Logical AND Operator (form 2)

Syntax

```
logical-AND-expression-2:
  assignment-expression
  logical-AND-expression-2  and  assignment-expression
```

assignment-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

Except for the difference in precedence, operator and has exactly the same semantics as operator `&&` (§§).

Logical Exclusive OR Operator

Syntax

```
logical-exc-OR-expression:
  logical-AND-expression-2
  logical-exc-OR-expression  xor  logical-AND-expression-2
```

logical-AND-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

If either operand does not have type `bool`, its value is first converted to that type.

Given the expression `e1 xor e2`, `e1` is evaluated first, then `e2`. If either `e1` or `e2` is `TRUE`, but not both, the result has type `bool`, value `TRUE`. Otherwise, the result has type `bool`, value `FALSE`. There is a sequence point after the evaluation of `e1`.

This operator associates left-to-right.

Examples

```
f($i++) XOR g($i) // the sequence point makes this well-defined
```

Logical Inclusive OR Operator (form 2)

Syntax

```
logical-inc-OR-expression-2:
  logical-exc-OR-expression
  logical-inc-OR-expression-2  or  logical-exc-OR-expression
```

logical-exc-OR-expression is defined in §§.

Constraints

Each of the operands must have scalar type.

Semantics

Except for the difference in precedence, operator and has exactly the same semantics as operator `||` (§§).

`yield` *Operator*

Syntax

```
yield-expression:
  logical-inc-OR-expression-2
  yield  array-element-initializer
```

logical-inc-OR-expression is defined in §§; array-element-initializer is defined in §§.

Semantics

Any function containing a yield-expression is a generator function. A generator function generates a collection of zero or more key/value pairs where each pair represents the next in some series. For example, a generator might yield random numbers or the series of Fibonacci numbers. When a generator function is called explicitly, it returns an object of type `Generator` (§§), which implements the interface `Iterator` (§§). As such, this allows that object to be iterated over using the `foreach` statement (§§). During each iteration, the Engine calls the generator function implicitly to get the next key/value pair. Then the Engine saves the state of the generator for subsequent key/value pair requests.

This operator produces the result `NULL` unless the method `Generator->send` (§§) was called to provide a result value. This operator has the side effect of generating the next value in the collection.

Before being used, an element-key must have, or be converted to, type `int` or `string`. Keys with `float` or `bool` values, or strings whose contents match exactly the pattern of decimal-literal (§§), are converted to `int` (§§). Values of all other key types are converted to `string` (§§).

If element-key is omitted from an array-element-initializer, an element key of type `int` is associated with the corresponding element-value. The key associated is one more than the previously assigned int key for this collection. However, if this is the first element in this collection with an `int` key, key zero is used. If element-key is provided, it is associated with the corresponding element-value. The resulting key/value pair is made available by `yield`.

If array-element-initializer is omitted, default int-key assignment is used and each value is `NULL`.

If the generator function definition declares that it returns byRef, each value in a key/value pair is yielded byRef.

Examples

```
function getTextFileLines($filename)
{
  $infile = fopen($filename, 'r');
  if ($infile == FALSE) { /* deal with the file-open failure */ }

  try
  {
    while ($textLine = fgets($infile))  // while not EOF
    {
      $textLine = rtrim($textLine, "\r\n"); // strip off terminator
      yield $textLine;
    }
  }
  finally
  {
    fclose($infile);
  }
}
foreach (getTextFileLines("Testfile.txt") as $line) { /* process each line */ }
// -----------------------------------------
function series($start, $end, $keyPrefix = "")
{
  for ($i = $start; $i <= $end; ++$i)
  {
    yield $keyPrefix . $i => $i;  // generate a key/value pair
  }
}
foreach (series(1, 5, "X") as $key => $val) { /* process each key/val pair */ }
```

Script Inclusion Operators

General

Syntax

```
  expression:
    yield-expression
    include-expression
    include-once-expression
    require-expression
    require-once-expression
```

yield-expression is described in §§; include-expression is described in §§; include-once-expression is described in §§; require-expression is described in §§; and require-once-expression is described in §§.

Semantics:

When creating large applications or building component libraries, it is useful to be able to break up the source code into small, manageable pieces each of which performs some specific task, and which can be shared somehow, and tested, maintained, and deployed individually. For example, a programmer might define a series of useful constants and use them in numerous and possibly unrelated applications. Likewise, a set of class definitions can be shared among numerous applications needing to create objects of those types.

An include file is a script that is suitable for inclusion by another script. The script doing the including is the including file, *while the one being included is the included file. A script can be an including file and an included file, either, or neither.*

Using the series-of-constants example, an include file called `Positions.php` might define the constants `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`, in their own namespace (§§), Positions. Using the set-of-classes example, to support two-dimensional geometry applications, an include file called `Point.php` might define the class `Point`. An include file called `Line.php` might define the class Line (where a `Line` is represented as a pair of Points).An include file, called `Circle.php` might define the class `Circle` (where a `Circle` is represented as a `Point` for the origin, and a radius).

If a number of the scripts making up an application each use one or more of the Position constants, they can each include the corresponding include file via the `include` operator (§§). However, most include files behave the same way each time they are included, so it is generally a waste of time including the same include file more than once into the same scope. In the case of the geometry example, any attempt to include the same include file more than once will result in a fatal "attempted class type redefinition" error. However, this can be avoided by using the `include_once` operator (§§) instead.

The `require` operator (§§) is a variant of the `include` operator, and the `require_once` operator (§§) is a variant of the `include_once` operator.

It is important to understand that unlike the C/C++ (or similar) preprocessor, script inclusion in PHP is not a text substitution process. That is, the contents of an included file are not treated as if they directly replaced the inclusion operation source in the including file.

An inclusion expression can be written to look like a function call; however, that is not the case, even though an included file can return a value to its including file.

The name used to specify an include file may contain an absolute or relative path. In the latter case, an implementation may use the configuration directive `include_path` (§xx) to resolve the include file's location.

The `include` Operator

Syntax

```
include-expression:
  include  (  include-filename  )
  include  include-filename

include-filename:
  expression
```

expression is defined in §§.

Constraints:

expression must be a string that designates a file that exists, is accessible, and whose format is suitable for inclusion (that is, starts with a PHP start-tag, and optionally ends with a PHP end-tag). However, if the designated file is not accessible, execution may continue.

Semantics:

When an included file is opened, parsing immediately drops out of PHP mode and into HTML mode at the beginning, and switches back again when the end of the included file is reached.

Variables defined in an included file take on scope of the source line on which the inclusion occurs in the including file. However, functions and classes defined in the included file are given global scope.

If inclusion occurs inside a function definition within the including file, the complete contents of the included file are treated as though it were defined inside that function.

Operator `include` has a side effect of including the designated include file. The result produced by this operator is one of the *following: `FALSE`, which indicates the inclusion attempt failed; the `int` 1, which indicates the default value for inclusion attempt succeeded; or some other value, as returned from the included file (§§).*

The library function `get_included_files` (§xx) provides the names of all files included or required.

Examples:

```
$fileName = 'limits' . '.php'; include $fileName;
$inc = include('limits.php');
If ((include 'Positions.php') == 1) ...
```

The `include_once` Operator

Syntax

```
  include-once-expression:
    include_once  (  include-filename  )
    include_once  include-filename
```

include-filename is defined in §§.

Semantics:

This operator is identical to operator `include` (§§) except that in the case of `include_once`, the include file is included once only during program execution.

Once an include file has been included, a subsequent use of `include_once` on that include file results in a return value of `TRUE`.

Examples:

Point.php:

```
\\ Point.php:
<?php ...
class Point { ... }

\\ Circle.php:
<?php ...
include_once 'Point.php';
class Circle { /* uses Point somehow */ }

\\ MyApp.php
include_once 'Point.php';   // Point.php included directly
include_once 'Circle.php';    // Point.php now not included indirectly
$p1 = new Point(10, 20);
$c1 = new Circle(9, 7, 2.4);
```

The `require` Operator

Syntax

```
  require-expression:
    require  (  include-filename  )
    require  include-filename
```

include-filename is defined in §§.

Semantics:

This operator is identical to operator `include` (§§) except that in the case of `require`, failure to find/open the designated include file terminates program execution.

The library function `get_included_files` (§xx) provides the names of all files included or required.

The `require_once` Operator

Syntax

```
require-once-expression:
  require_once  (  include-filename  )
  require_once  include-filename
```

include-filename is defined in §§.

Semantics:

This operator is identical to operator `require` (§§) except that in the case of `require_once`, the include file is included once only during program execution.

Once an include file has been included, a subsequent use of `require_once` on that include file results in a return value of TRUE.

Constant Expressions

Syntax

```
constant-expression:
  array-creation-expression
  const-expression

const-expression:
  expression
```

array-creation-expression is defined in §§ and expression is defined in §§.

Constraints:

All of the element-key and element-value expressions in array-creation-expression (§§) must be literals.

expression must have a scalar type, and be a literal or the name of an existing c-constant (§§), that is currently in scope.

Semantics:

A const-expression is the value of a c-constant. A const-expression is required in several contexts, such as in initializer values in a const-declaration (§§) and default initial values in a function definition (§§).

An initializer in a property-declaration (§§) is less restrictive than one in a const-declaration.

Statements

General

Syntax

```
statement:
  compound-statement
  labeled-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  declare-statement
  const-declaration
  function-definition
  class-declaration
  interface-declaration
  trait-declaration
  namespace-definition
  namespace-use-declaration
  global-declaration
  function-static-declaration
```

compound-statement is defined in §§; labeled-statement is defined in §§; expression-statement is defined in §§; selection-statement is defined in §§; iteration-statement is defined in §§; jump-statement is defined in §§; declare-statement is defined in §§; const-declaration is defined in §§; function-definition is defined in §§; class-declaration is defined in §§; interface-declaration is defined in §§; trait-declaration is defined in §§; namespace-definition is defined in §§; namespace-use-declaration is defined in §§; global-declaration is defined in §§; and function-static-declaration is defined in §§.

Compound Statements

Syntax

```
compound-statement:
  {   statement-list_opt   }

statement-list:
  statement
  statement-list   statement
```

statement is defined in §§.

Semantics

A compound statement allows a group of zero of more statements to be treated syntactically as a single statement. A compound statement is often referred to as a block.

Examples

```
if (condition)
{ // braces are needed as the true path has more than one statement
  // statement-1
  // statement-2
}
else
{ // braces are optional as the false path has only one statement
  // statement-3
}
// -----------------------------------------
while (condition)
{ // the empty block is equivalent to a null statement
}
```

Labeled Statements

Syntax

```
labeled-statement:
  named-label
  case-label
  default-label

named-label:
  name  :  statement

case-label:
  case   expression   case-default-label-terminator   statement

default-label:
  default  case-default-label-terminator   statement

case-default-label-terminator:
  :
  ;
```

name is defined in §§; *statement* is defined in §§; and *expression* is defined in §§.

Constraints

A named label must only be used as the target of a `goto` statement (§§).

Named labels must be unique within a function.

A case and default label must only occur inside a `switch` statement (§§).

Semantics

Any statement may be preceded by a token sequence that declares a name as a label name. The presence of a label does not alter the flow of execution.

Expression Statements

Syntax

```
expression-statement:
  expression_opt  ;
```

expression is defined in §§.

Semantics

If present, *expression* is evaluated for its side effects, if any, and any resulting value is discarded. If *expression* is omitted, the statement is a *null statement*, which has no effect on execution.

Examples

```
$i = 10;  // $i is assigned the value 10; result (10) is discarded
++$i; // $i is incremented; result (11) is discarded
$i++; // $i is incremented; result (11) is discarded
DoIt(); // function DoIt is called; result (return value) is discarded
```

```
// -----------------------------------------
$i;   // no side effects, result is discarded. Vacuous but permitted
123;  // likewise for this one and the two statements following
34.5 * 12.6 + 11.987;
TRUE;
// -----------------------------------------
function findValue($table, $value)  // where $table is 2x3 array
{
  for ($row = 0; $row <= 1; ++$row)
  {
    for ($colm = 0; $colm <= 2; ++$colm)
    {
      if ($table[$row][$colm] == $value)
      {
        // ...
        goto done;
      }
    }
  }
  // ...
done:
  ;     // null statement needed as a label must precede a statement
}
```

Selection Statements

General

Syntax

```
  selection-statement:
    if-statement
    switch-statement
```

if-statement is defined in §§ and switch-statement is defined in §§.

Semantics

Based on the value of a controlling expression, a selection statement selects among a set of statements.

The `if` Statement

Syntax

```
  if-statement:
    if   (   expression   )   statement   elseif-clauses-1opt   else-clause-1opt
    if   (   expression   )   :   statement-list   elseif-clauses-2opt   else-clause-2opt   endif   ;

  elseif-clauses-1:
    elseif-clause-1
    elseif-clauses-1   elseif-clause-1

  elseif-clause-1:
    elseif   (   expression   )   statement

  else-clause-1:
    else   statement

  elseif-clauses-2:
```

```
    elseif-clause-2
    elseif-clauses-2   elseif-clause-2

  elseif-clause-2:
    elseif   (   expression   )   :   statement-list

  else-clause-2:
    else   :   statement-list
```

expression is defined in §§; *statement* is defined in §§; and *statement-list* is defined in §§.

Constraints

The controlling expression *expression* must have type `bool` or be implicitly convertible to that type.

Semantics

The two forms of the `if` statement are equivalent; they simply provide alternate styles.

If *expression* tests `TRUE`, the *statement* that follows immediately is executed. Otherwise, if an `elseif` clause is present the *statement* immediately following the `elseif` is executed. Otherwise, any other `elseif` *expressions* are evaluated. If none of those tests `TRUE`, if an `else` clause is present the *statement* immediately following the `else` is executed.

An `else` clause is associated with the lexically nearest preceding `if` or `elseif` that is permitted by the syntax.

Examples

```
if ($count > 0)
{
  ...
  ...
  ...
}
// ----------------------------------------
goto label1;
echo "Unreachable code\n";

if ($a)
{
label1:
  ...
}
else
{
  ...
}
// ----------------------------------------
if (1)
  ...
  if (0)
    ...
else  // this else does NOT go with the outer if
  ...

if (1)
{
  ...
  if (0)
    ...
}
else  // this else does go with the outer if
    ...
```

The `switch` Statement

Syntax

```
switch-statement:
  switch  (  expression  )  compound-statement
  switch  (  expression  )  :   statement-list  endswitch;
```

expression is defined in §§; and *compound-statement* and *statement-list* are defined in §§.

Constraints

The controlling expression *expression* must have scalar type.

The *statement-list* must not contain any *compound-statement*s.

There must be at most one default label.

Semantics

The two forms of the `switch` statement are equivalent; they simply provide alternate styles.

Based on the value of its *expression*, a `switch` statement transfers control to a case label (§11.3); to a default label (§11.3), if one exists; or to the statement immediately following the end of the `switch` statement. A case or default label is only reachable directly within its closest enclosing `switch` statement.

On entry to the `switch` statement, the controlling expression is evaluated and then compared with the value of the case-label-expression values, in lexical order. If one matches, control transfers to the statement following the corresponding case label. If there is no match, then if there is a default label, control transfers to the statement following that; otherwise, control transfers to the statement immediately following the end of the `switch` statement. If a `switch` contains more than one case label whose values compare equal to the controlling expression, the first in lexical order is consider the match.

An arbitrary number of statements can be associated with any case or default label. In the absence of a `break` statement (§§) at the end of a set of such statements, control drops through into any following case or default label. Thus, if all cases and the default end in break and there are no duplicate-valued case labels, the order of case and default labels is insignificant.

Case-label values can be runtime expressions, and the types of sibling case-label values need not be the same.

Switches may nested, in which case, each `switch` has its own set of `switch` clauses.

Examples

```
$v = 10;
switch ($v)
{
default:
  echo "default case: \$v is $v\n";
  break;    // break ends "group" of default statements
case 20:
  echo "case 20\n";
  break;    // break ends "group" of case 20 statements
case 10:
  echo "case 10\n"; // no break, so control drops into next label's "group"
case 30:
  echo "case 30\n"; // no break, but then none is really needed either
}
// -----------------------------------------
$v = 30;
switch ($v)
```

```
{
case 30.0:  // <===== this case matches with 30
  echo "case 30.0\n";
  break;
default:
  echo "default case: \$v is $v\n";
  break;
case 30:    // <===== rather than this case matching with 30
  echo "case 30\n";
  break;
}
// -----------------------------------------
switch ($v)
{
case 10 + $b: // non-constant expression
  // ...
case $v < $a:   // non-constant expression
  // ...
// ...
}
```

Iteration Statements

General

Syntax

```
iteration-statement:
  while-statement
  do-statement
  for-statement
  foreach-statement
```

while-statement is defined in §§; do-statement is defined in §§; for-statement is defined in §§; and foreach-statement is defined in §§.

The `while` Statement

Syntax

```
while-statement:
  while  (  expression  )  statement
  while  (  expression  )  :   statement-list  endwhile ;
```

expresion is defined in §§; statement is defined in §§; and statement-list is defined in §§.

Constraints

The controlling expression expression must have type `bool` or be implicitly convertible to that type.

Semantics

The two forms of the `while` statement are equivalent; they simply provide alternate styles.

If expression tests `TRUE`, the statement that follows immediately is executed, and the process is repeated. If expression tests `FALSE`, control transfers to the point immediately following the end of the `while` statement. The loop body, statement, is executed zero or more times.

Examples

```
$i = 1;
while ($i <= 10):
  echo "$i\t".($i * $i)."\n"; // output a table of squares
  ++$i;
endwhile;
// ----------------------------------------
while (TRUE)
{
  // ...
  if ($done)
    break;  // break out of the while loop
  // ...
}
```

The `do` Statement

Syntax

```
  do-statement:
    do  statement  while  (  expression  )  ;
```

statement is defined in §§ and *expression* is defined in §§.

(Note: There is no `:/enddo` alternate syntax).

Constraints

The controlling expression *expression* must have type `bool` or be implicitly convertible to that type.

Semantics

First, *statement* is executed and then *expression* is tested. If its value is `TRUE`, the process is repeated. If *expression* tests `FALSE`, control transfers to the point immediately following the end of the `do` statement. The loop body, *statement*, is executed one or more times.

Examples

```
$i = 1;
do
{
  echo "$i\t".($i * $i)."\n"; // output a table of squares
  ++$i;
}
while ($i <= 10);
```

The `for` Statement

Syntax

```
  for-statement:
    for   (   for-initializeropt   ;   for-controlopt   ;   for-end-of-loopopt   )   statement
    for   (   for-initializeropt   ;   for-controlopt   ;   for-end-of-loopopt   )   :   statement-list   endfor   ;

  for-initializer:
    for-expression-group
```

```
  for-control:
    for-expression-group

  for-end-of-loop:
    for-expression-group

  for-expression-group:
    expression
    for-expression-group   ,   expression
```

statement is defined in §§; statement-list is defined in §§; and expression is defined in §§.

Note: Unlike C/C++, PHP does not support a comma operator, per se. However, the syntax for the `for` statement has been extended from that of C/C++ to achieve the same results in this context.

Constraints

The controlling expression—the right-most expression in for-control—must have type `bool` or be implicitly convertible to that type.

Semantics

The two forms of the `for` statement are equivalent; they simply provide alternate styles.

The group of expressions in for-initializer is evaluated once, left-to-right, for their side effects. Then the group of expressions in for-control is evaluated left-to-right (with all but the right-most one for their side effects only), with the right-most expression's value being tested. If that tests `TRUE`, statement is executed, and the group of expressions in for-end-of-loop is evaluated left-to-right, for their side effects only. Then the process is repeated starting with for-control. If the right-most expression in for-control tests `FALSE`, control transfers to the point immediately following the end of the `for` statement. The loop body, statement, is executed zero or more times.

If for-initializer is omitted, no action is taken at the start of the loop processing. If for-control is omitted, this is treated as if for-control was an expression with the value `TRUE`. If for-end-of-loop is omitted, no action is taken at the end of each iteration.

Examples

```
for ($i = 1; $i <= 10; ++$i)
{
  echo "$i\t".($i * $i)."\n"; // output a table of squares
}
// -----------------------------------------
// omit 1st and 3rd expressions

$i = 1;
for (; $i <= 10;):
  echo "$i\t".($i * $i)."\n"; // output a table of squares
  ++$i;
endfor;
// -----------------------------------------
// omit all 3 expressions

$i = 1;
for (;;)
{
  if ($i > 10)
    break;
  echo "$i\t".($i * $i)."\n"; // output a table of squares
  ++$i;
}
// -----------------------------------------
```

```
//  use groups of expressions

for ($a = 100, $i = 1; ++$i, $i <= 10; ++$i, $a -= 10)
{
  echo "$i\t$a\n";
}
```

The `foreach` Statement

Syntax

```
foreach-statement:
  foreach  (  foreach-collection-name  as  foreach-key_opt  foreach-value  )   statement
  foreach  (  foreach-collection-name  as  foreach-key_opt   foreach-value  )  :   statement-list  endforeach  ;

foreach-collection-name:
  expression

foreach-key:
  expression  =>

foreach-value:
  &_opt   expression
  list-intrinsic
```

statement is defined in §§; statement-list is defined in §§; variable-name is defined in §§; list-intrinsic is defined in §§; and expression is defined in §§.

Constraints

The variable designated by foreach-collection-name must be a collection.

Each expression must designate a variable name.

Semantics

The two forms of the `foreach` statement are equivalent; they simply provide alternate styles.

The foreach statement iterates over the set of elements in the collection designated by foreach-collection-name, starting at the beginning, executing statement each iteration. On each iteration, if the `&` is present in foreach-value, the variable designated by the corresponding expression is made an alias to the current element. If the `&` is omitted, the value of the current element is assigned to the corresponding variable. The loop body, statement, is executed zero or more times. After the loop terminates, expression in foreach-value has the same meaning it had after the final iteration, if any.

If foreach-key is present, the variable designated by its expression is assigned the current element's key value.

In the list-intrinsic case, a value that is an array is split into individual elements.

Examples

```
$colors = array("red", "white", "blue");
foreach ($colors as $color):
    // ...
endforeach;
// ----------------------------------------
foreach ($colors as $key => $color)
{
    // ...
}
// ----------------------------------------
```

```
// Modify the local copy of an element's value

foreach ($colors as $color)
{
  $color = "black";
}
// -----------------------------------------
// Modify the the actual element itself

foreach ($colors as &$color)  // note the &
{
  $color = "black";
}
```

Jump Statements

General

Syntax

```
  jump-statement:
    goto-statement
    continue-statement
    break-statement
    return-statement
    throw-statement
```

goto-statement is defined in §§; continue-statement is defined in §§; break-statement is defined in §§; return-statement is defined in §§; and throw-statement is defined in §§.

The `goto` Statement

Syntax

```
  goto-statement:
    goto  name  ;
```

name is defined in §§.

Constraints

The name in a `goto` statement must be that of a named label located somewhere in the current script. Control must not be transferred into or out of a function, or into an iteration statement (§§) or a `switch` statement (§§).

A `goto` statement must not attempt to transfer control out of a finally-block (§§).

Semantics

A `goto` statement transfers control unconditionally to the named label (§§).

A `goto` statement may break out of a construct that is fully contained within a finally-block.

Examples

```
function findValue($table, $v)  // where $table is 2x3 array
{
  for ($row = 0; $row <= 1; ++$row)
```

```
  {
    for ($colm = 0; $colm <= 2; ++$colm)
    {
      if ($table[$row][$colm] == $v)
      {
        echo "$v was found at row $row, column $colm\n";
        goto done; // not quite the same as break 2!
      }
    }
  }
  echo "$v was not found\n";
done:
  ; // note that a label must always precede a statement
}
```

The `continue` Statement

Syntax

```
  continue-statement:
    continue   breakout-level_opt  ;

  breakout-level:
    integer-literal
```

integer-literal is defined in §§.

Constraints

The breakout level must not be zero, and it must not exceed the level of actual enclosing iteration and/or `switch` statements.

A `continue` statement must not attempt to break out of a finally-block (§§).

Semantics

A `continue` statement terminates the execution of the innermost enclosing iteration (§§) or `switch` (§§) statement.

A `continue` statement terminates the execution of one or more enclosing iteration (§§) or `switch` (§§) statements. If breakout-level is greater than one, the next iteration (if any) of the next innermost enclosing iteration or switch statement is started; however, if that statement is a `for` statement and it has a for-end-of-loop, its expression group for the current iteration is evaluated first. If breakout-level is 1, the behavior is the same as for `break 1`. If breakout-level is omitted, a level of 1 is assumed.

A `continue` statement may break out of a construct that is fully contained within a finally-block.

Examples

```
for ($i = 1; $i <= 5; ++$i)
{
  if (($i % 2) == 0)
    continue;
  echo "$i is odd\n";
}
```

The `break` Statement

Syntax

```
break-statement:
  break  breakout-level_opt  ;
```

breakout-level is defined in §§.

Constraints

The breakout level must not be zero, and it must not exceed the level of actual enclosing iteration and/or `switch` statements.

A `break` statement must not attempt to break out of a finally-block (§§).

Semantics

A `break` statement terminates the execution of one or more enclosing iteration (§§) or `switch` (§§) statements. The number of levels broken out is specified by breakout-level. If breakout-level is omitted, a level of 1 is assumed.

A `break` statement may break out of a construct that is fully contained within a finally-block.

Examples

```
$i = 1;
for (;;)
{
  if ($i > 10)
    break;
  // ...
  ++$i;
}
// ----------------------------------------
for ($row = 0; $row <= 1; ++$row)
{
  for ($colm = 0; $colm <= 2; ++$colm)
  {
    if (some-condition-set)
    {
      break 2;
    }
    // ...
  }
}
// ----------------------------------------
for ($i = 10; $i <= 40; $i +=10)
{
        switch($i)
        {
        case 10: /* ... */; break;    // breaks to the end of the switch
        case 20: /* ... */; break 2;  // breaks to the end of the for
        case 30: /* ... */; break;    // breaks to the end of the switch
        }
}
```

The `return` Statement

Syntax

```
return-statement:
  return  expression_opt  ;
```

expression is defined in §§.

Constraints

The expression in a return-statement in a generator function (§§) must be the literal `NULL` or be omitted.

Semantics

A `return` statement from within a function terminates the execution of that function normally, and depending on how the function was defined (§§), it returns the value of expression to the function's caller by value or byRef. If expression is omitted the value `NULL` is used.

If execution flows into the closing brace (`}`) of a function, `return NULL;` is implied.

A function may have any number of `return` statements, whose returned values may have different types.

If an undefined variable is returned byRef, that variable becomes defined, with a value of `NULL`.

A `return` statement is permitted in a try-block (§§) and a catch-block (§§). However, it is unspecified whether a `return` statement is permitted in a finally-block (§§), and, if so, the semantics of that.

Using a `return` statement inside a finally-block will override any other `return` statement or thrown exception from the try-block and all its catch-blocks. Code execution in the parent stack will continue as if the exception was never thrown.

If an uncaught exception exists when a finally-block is executed, if that finally-block executes a `return` statement, the uncaught exception is discarded.

In an included file (§§) a `return` statement may occur outside any function. This statement terminates processing of that script and returns control to the including file. If expression is present, that is the value returned; otherwise, the value `NULL` is returned. If execution flows to the end of the script, `return 1;` is implied. However, if execution flows to the end of the top level of a script, `return 0;` is implied. Likewise, if expression is omitted at the top level. (See exit (§§)).

Returning from a constructor or destructor behaves just like returning from a function.

A `return` statement inside a generator function causes the generator to terminate.

Return statements can also be used in the body of anonymous functions.

`return` terminates the execution of source code given to the intrinsic `eval` (§§).

Examples

```
function f() { return 100; }  // f explicitly returns a value
function g() { return; }   // g explicitly returns an implicit NULL
function h() { }      // h implicitly returns NULL
// -----------------------------------------
// j returns one of three dissimilarly-typed values
function j($x)
{
  if ($x > 0)
  {
    return "Positive";
  }
  else if ($x < 0)
  {
    return -1;
  }
  // for zero, implied return NULL
}
function &compute() { ...; return $value; } // returns $value byRef
// -----------------------------------------
class Point
{
  private static $pointCount = 0;
```

```
  public static function getPointCount()
  {
    return self::$pointCount;
  }
  ...
}
```

Implementation Notes

Although expression is a full expression (§§), and there is a sequence point (§§) at the end of that expression, as stated in §§, a side effect need not be executed if it can be determined that no other program code relies on its having happened. (For example, in the cases of `return $a++;` and `return ++$a;`, it is obvious what value must be returned in each case, but if `$a` is a variable local to the enclosing function, `$a` need not actually be incremented.

The `throw` Statement

Syntax

```
  throw-statement:
    throw  expression  ;
```

expression is defined in §§.

Constraints

The type of expression must be Exception (§§) or a subclass of that class.

expression must be such that an alias to it can be created.

Semantics

A `throw` statement throws an exception immediately and unconditionally. Control never reaches the statement immediately following the throw. See §§ and §§ for more details of throwing and catching exceptions, and how uncaught exceptions are dealt with.

Rather than handle an exception, a catch-block may (re-)throw the same exception that it caught, or it can throw an exception of a different type.

Examples

```
throw new Exception;
throw new Exception("Some message", 123);
class MyException extends Exception { ... }
throw new MyException;
```

The `try` Statement

Syntax

```
  try-statement:
    try  compound-statement   catch-clauses
    try  compound-statement   finally-clause
    try  compound-statement   catch-clauses   finally-clause

  catch-clauses:
    catch-clause
    catch-clauses   catch-clause
```

```
catch-clause:
  catch  (  parameter-declaration-list  )  compound-statement

finally-clause:
  finally   compound-statement
```

compound-statement is defined in §§ and parameter-declaration-list is defined in §§.

Constraints

In a catch-clause, parameter-declaration-list must contain only one parameter, and its type must be `Exception` *(§§) or a type derived from that class, and that parameter must not be passed byRef.*

Semantics

In a catch-clause, identifier designates an exception variable passed in by value. This variable corresponds to a local variable with a scope that extends over the catch-block. During execution of the catch-block, the exception variable represents the exception currently being handled.

Once an exception is thrown, the Engine searches for the nearest catch-block that can handle the exception. The process begins at the current function level with a search for a try-block that lexically encloses the throw point. All catch-blocks associated with that try-block are considered in lexical order. If no catch-block is found that can handle the run-time type of the exception, the function that called the current function is searched for a lexically enclosing try-block that encloses the call to the current function. This process continues until a catch-block is found that can handle the current exception.

If a matching catch-block is located, the Engine prepares to transfer control to the first statement of that catch-block. However, before execution of that catch-block can start, the Engine first executes, in order, any finally-blocks associated with try-blocks nested more deeply than the one that caught the exception.

If no matching catch-block is found, the behavior is implementation-defined.

Examples

```
function getTextLines($filename)
{
  $infile = fopen($filename, 'r');
  if ($infile == FALSE) { /* deal with an file-open failure */ }
  try
  {
    while ($textLine = fgets($infile))  // while not EOF
    {
      yield $textLine;  // leave line terminator attached
    }
  }
  finally
  {
    fclose($infile);
  }
}
// ----------------------------------------
class DeviceException extends Exception { ... }
class DiskException extends DeviceException { ... }
class RemovableDiskException extends DiskException { ... }
class FloppyDiskException extends RemovableDiskException { ... }

try
{
  process(); // call a function that might generate a disk-related exception
}
catch (FloppyDiskException $fde) { ... }
catch (RemovableDiskException $rde) { ... }
```

```
catch (DiskException $de) { ... }
catch (DeviceException $dve) { ... }
finally { ... }
```

The `declare` Statement

Syntax

```
declare-statement:
  declare  (  declare-directive  )  statement
  declare  (  declare-directive  )  :  statement-list  enddeclare  ;
  declare  (  declare-directive  )  ;

declare-directive:
  ticks  =  declare-tick-count
  encoding  =  declare-character-encoding

declare-tick-count
  expression

declare-character-encoding:
  expression
```

statement is defined in §§; *statement-list* is defined in §§; and *expression* is defined in §§.

Constraints

tick-count must designate a value that is, or can be converted, to an integer having a non-negative value.

character-encoding must designate a string whose value names an 8-bit-character encoding.

Except for white space, a *declare-statement* in a script that specifies *character-encoding* must be the first thing in that script.

Semantics

The first two forms of the `declare` statement are equivalent; they simply provide alternate styles.

The `declare` statement sets an *execution directive* for its *statement* body, or for the `;`-form, for the remainder of the script or until the statement is overridden by another *declare-statement*, whichever comes first. As the parser is executing, certain statements are considered *tickable*. For every *tick-count* ticks, an event occurs, which can be serviced by the function previously registered by the library function `register_tick_function` (§xx). Tick event monitoring can be disabled by calling the library function `unregister_tick_function` (§xx). This facility allows a profiling mechanism to be developed.

Character encoding can be specified on a script-by-script basis using the encoding directive. The joint ISO and IEC standard ISO/IEC 8859 standard series (http://en.wikipedia.org/wiki/ISO/IEC_8859) specifies a number of 8-bit-character encodings whose names can be used with this directive.

Examples

```
declare(ticks = 1) { ... }
declare(encoding = 'ISO-8859-1'); // Latin-1 Western European
declare(encoding = 'ISO-8859-5'); // Latin/Cyrillic
```

Arrays

General

An array is a data structure that contains a collection of zero or more elements. The elements of an array need not have the same type, and the type of an array element can change over its lifetime. An array element can have any type (which allows for arrays of arrays). However, PHP does not support multidimensional arrays.

An array is represented as an ordered map in which each entry is a key/value pair that represents an element. An element key can be an expression of type `int` or `string`. Duplicate keys are not permitted. The order of the elements in the map is the order in which the elements were inserted into the array. An element is said to exist once it has been inserted into the array with a corresponding key. An array is extended by initializing a previously non-existent element using a new key. Elements can be removed from an array via the intrinsic unset (§§).

The `foreach` statement (§§) can be used to iterate over the collection of elements in an array, in the order in which the elements were inserted. This statement provides a way to access the key and value for each element.

Each array has its own current element pointer that designates the current array element. When an array is created, the current element is the first element inserted into the array.

Numerous library functions are available to create and/or manipulate arrays. See §xx.

(Note: Arrays in PHP are quite different from arrays in numerous mainstream languages. Specifically, in PHP, array elements need not have the same type, the subscript index need not be an integer (so there is no concept of a base index of zero or 1), and there is no concept of consecutive elements occupying physically adjacent memory locations).

Array Creation and Initialization

An array is created and initialized by one of two equivalent ways: via the array-creation operator `[]` (§§) or the intrinsic array (§§).

Element Access and Insertion

The value (and possibly the type) of an existing element is changed, and new elements are inserted, using the subscript operator `[]` (§§).

Functions

General

When a function is called, information may be passed to it by the caller via an argument list, which contains one or more argument expressions, or more simply, arguments. These correspond by position to the parameters in a parameter list in the called function's definition (§§).

An unconditionally defined function is a function whose definition is at the top level of a script. A conditionally defined function is a function whose definition occurs inside a compound statement (which is inside a function definition); that is, it is a nested function. There is no limit on the depth of levels of function nesting. Consider the case of an outer function, and an inner function defined within it. Until the outer function is called at least once, its inner function cannot exist. Even if the outer function is called, if its runtime logic bypasses the definition of the inner function, that inner function still does not exist.

Any function containing `yield` (§§) is a generator function.

Examples

```
ucf1(); // can call ucf1 before its definition is seen
function ucf1() { ... }
ucf1(); // can call ucf1 after its definition is seen
cf1(); // Error; call to non-existent function
$flag = TRUE;
if ($flag) { function cf1() { ... } } // cf1 now exists
if ($flag) { cf1(); } // can call cf1 now
// -----------------------------------------
function ucf2() { function cf2() { ... } }
cf2(); // Error; call to non-existent function
ucf2(); // now cf2 exists
cf2(); // so we can call it
```

Function Calls

A function is called via the function-call operator `()` (§§).

Function Definitions

Syntax

```
function-definition:
  function-definition-header   compound-statement

function-definition-header:
  function   &opt   name   (   parameter-declaration-listopt   )

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list   ,   parameter-declaration

parameter-declaration:
  type-hintopt   &opt   variable-name   default-argument-specifieropt

type-hint:
  array
  callable
  qualified-name

default-argument-specifier:
  =  const-expression
```

const-expression is defined in §§. qualified-name is defined in §§.

Constraints

Each parameter name in a function-definition must be distinct.

A conditionally defined function (§§) must exist before any calls are made to that function.

parameter-declaration must not contain `&` if type-hint is `array` or `callable`.

Semantics

*A function-definition defines a function called name. Function names are **not** case-sensitive. A function can be defined with zero or more parameters, each of which is specified in its own parameter-declaration in a parameter-declaration-list. Each parameter has a name, variable-name, and optionally, a default-argument-specifier. An `&` in parameter-declaration indicates that parameter is passed byRef (§§) rather than by value. An `&` before name indicates that the value returned from this* *function is to be returned byRef. Function-value returning is described in §§.*

When the function is called, if there exists a parameter for which there is a corresponding argument, the argument is assigned to the parameter variable using value assignment, while for passing-byRef, the argument is assigned to the parameter variable using byRef assignment (§§, §§). If that parameter has no corresponding argument, but the parameter has a default argument value, for passing-by-value or passing-byRef, the default value is assigned to the parameter variable using value assignment. Otherwise, if the parameter has no corresponding argument and the parameter does not have a default value, the parameter variable is non-existent and no corresponding VSlot (§§) exists. After all possible parameters have been assigned initial values or aliased to arguments, the body of the function, compound-statement, is executed. This execution may terminate normally (§4.3, §§) or abnormally (§4.3).

Each parameter is a variable local to the parent function, and is a modifiable lvalue.

A function-definition may exist at the top level of a script, inside any compound-statement, in which case, the function is conditionally defined (§§), or inside a method-declaration (§§).

By default, a parameter will accept an argument of any type. However, by specifying a type-hint, the types of argument accepted can be restricted. By specifying `array`, only an argument designating an array type is accepted. By specifying `callable`, only an argument designating a function is accepted. By specifying qualified-name, only an instance of a class having that type, or being derived from that type, are accepted, or only an instance of a class that implements that interface type directly or indirectly is accepted.

Variable Functions

If a variable name is followed by the function-call operator `()` (§§), and the value of that variable is a string containing the name of a function currently defined and visible, that function will be executed.

The library function `is_callable` (§xx) reports whether the contents of a variable can be called as a function.

Anonymous Functions

An anonymous function, also known as a closure, is a function defined with no name. As such, it must be defined in the context of an expression whose value is used immediately to call that function, or that is saved in a variable for later execution. An anonymous function is defined via the anonymous function-creation operator (§§).

For both `__FUNCTION__` and `__METHOD__` (§§), an anonymous function's name is `{closure}`. All anonymous functions created in the same scope have the same name.

Classes

General

A class is a type that may contain zero or more explicitly declared members, which can be any combination of class constants (§§); data members, called properties (§§); and function members, called methods (§§). (The ability to add properties and methods to an instance at runtime is described in §§). An object (often called an instance) of a class type is created (i.e., instantiated) via the new operator (§§).

*PHP supports inheritance (§§), a means by which a derived class can extend and specialize a single base class. However, unlike numerous other languages, classes in PHP are **not** all derived from a common ancestor. An abstract class (§§) is a base type intended for derivation, but which cannot be instantiated directly. A concrete class is a class that is not abstract. A final class (§§) is one from which other classes cannot be derived.*

A class may implement one or more interfaces (§§, §§), each of which defines a contract.

A class can use one or more traits (§§), which allows a class to have some of the benefits of multiple inheritance.

A constructor (§§) is a special method that is used to initialize an instance immediately after it has been created. A destructor (§§) is a special method that is used to free resources when an instance is no longer needed. Other special methods exist; they are described in (§§).

The members of a class each have a default or explicitly declared visibility, which determines what source code can access them. A member with `private` visibility may be accessed only from within its own class. A member with `protected` visibility may be accessed only from within its own class and from classes derived from that class. Access to a member with `public` visibility is unrestricted.

The signature of a method is a combination of the parent class name, that method's name, and its parameter list, including type hints and indication for arguments passed using byRef, and whether the resulting value is returned byRef.

Methods and properties from a base class can be overridden in a derived class by redeclaring them with the same signature defined in the base class.

When an instance is allocated, new returns a handle that points to that object. As such, assignment of a handle does not copy the object itself. (See §§ for a discussion of shallow and deep copying).

Class Declarations

Syntax

```
class-declaration:
  class-modifier_opt  class  name   class-base   clause_opt  class-interface-clause_opt   {   trait-use-clauses_opt   clas

class-modifier:
  abstract
  final

class-base-clause:
  extends  qualified-name

class-interface-clause:
  implements  qualified-name
  class-interface-clause   ,  qualified-name
```

qualified-name is defined in §§. class-member-declarations is defined in §§. trait-use-clauses ~~ is defined in §§

Constraints

A class must not be derived directly or indirectly from itself.

A class-declaration containing any class-member-declarations that have the modifier `abstract` must itself have an `abstract` class-modifier.

class-base-clause must not name a final class.

qualified-name in class-base-clause must name a class type, and must not be `parent`, `self`, or `static`.

A concrete class must implement each of the methods from all the interfaces (§§) specified in class-interface-clause, using the exact same signature as defined in each interface.

qualified-name in class-interface-clause must name an interface type.

Semantics

A class-declaration defines a class type by the name name. Class names are case-insensitive.

The `abstract` modifier declares a class usable only as a base class; the class cannot be instantiated directly. An abstract class may contain one or more abstract members, but it is not required to do so. When a concrete class is derived from an abstract class, the concrete class must include an implementation for each of the abstract members it inherits.

The `final` modifier prevents a class from being used as a base class.

The optional class-base-clause specifies the one base class from which the class being defined is derived. In such a case, the derived class inherits all the members from the base class.

The optional class-interface-clause specifies the one or more interfaces that are implemented by the class being defined.

A class can use one or more traits via a trait-use-clauses; see §§ and §§.

Examples

```
abstract class Vehicle
{
  public abstract function getMaxSpeed();
  ...
}
abstract class Aircraft extends Vehicle
{
  public abstract function getMaxAltitude();
  ...
}
class PassengerJet extends Aircraft
{
  public function getMaxSpeed()
  {
    // implement method
  }
  public function getMaxAltitude()
  {
    // implement method
  }
  ...
}
$pj = new PassengerJet(...);
echo "\$pj's maximum speed: " . $pj->getMaxSpeed() . "\n";
echo "\$pj's maximum altitude: " . $pj->getMaxAltitude() . "\n";
// -----------------------------------------
final class MathLibrary
{
  private function MathLibrary() {} // disallows instantiation
  public static function sin() { ... }
  // ...
}
$v = MathLibrary::sin(2.34);
// -----------------------------------------
interface MyCollection
{
        function put($item);
        function get();
}
class MyList implements MyCollection
{
  public function put($item)
  {
    // implement method
  }
```

```
  public function get()
  {
    // implement method
  }
  ...
}
```

Class Members

Syntax

```
  class-member-declarations:
    class-member-declaration
    class-member-declarations   class-member-declaration

   class-member-declaration:
     const-declaration
     property-declaration
     method-declaration
     constructor-declaration
     destructor-declaration
```

const-declaration is defined in §§; property-declaration is defined in §§; method-declaration is defined in §§; constructor-declaration is defined in §§; and destructor-declaration is defined in §§.

Semantics

The members of a class are those specified by its class-member-declarations, and the members inherited from its base class. (A class may also contain dynamic members, as described in §§. However, as these have no compile-time names, they can only be accessed via method calls).

A class may contain the following members:

- *Constants – the constant values associated with the class (§§).*
- *Properties – the variables of the class (§§).*
- *Methods – the computations and actions that can be performed by the class (§§, §§).*
- *Constructor – the actions required to initialize an instance of the class (§§).*
- *Destructor – the actions to be performed when an instance of the class is no longer needed (§§).*

A number of names are reserved for methods with special semantics, which user-defined versions must follow. These are described in (§§).

Methods and properties can either be static or instance members. A static member is declared using `static`. An instance member is one that is not static. The name of a static method or property can never be used on its own; it must always be used as the right-hand operand of the scope resolution operator (§§). The name of an instance method or property can never be used on its own; it must always be used as the right-hand operand of the member selection operator (§§).

Each instance of a class contains its own, unique set of instance properties of that class. An instance member is accessed via the `->` operator (§§). In contrast, a static property designates exactly one VSlot for its class, which does not belong to any instance, per se. A static property exists whether or not any instances of that class exist. A static member is accessed via the `::` operator (§§).

When any instance method operates on a given instance of a class, within that method that object can be accessed via `$this` (§§). As a static method does not operate on a specific instance, it has no `$this`.

Examples

```
class Point
{
  private static $pointCount = 0;     // static property

  private $x;             // instance property
  private $y;             // instance property

  public static function getPointCount()    // static method
  {
    return self::$pointCount;     // access static property
  }
  public function move($x, $y)        // instance method
  {
    $this->x = $x;
    $this->y = $y;
  }
  public function __construct($x = 0, $y = 0) // instance method
  {
    $this->x = $x;          // access instance property
    $this->y = $y;          // access instance property
    ++self::$pointCount;    // access static property
  }
  public function __destruct()      // instance method
  {
    --self::$pointCount;        // access static property
    ...
  }
  ...
}
echo "Point count = " . Point::getPointCount() . "\n";
$cName = 'Point';
echo "Point count = " . $cName::getPointCount() . "\n";
```

Dynamic Members

*Ordinarily, all of the instance properties and methods of a class are declared explicitly in that class's definition. However, other members—dynamic properties and, under certain circumstances, dynamic methods—can be added to a particular instance of a class or to the class as a whole at runtime. A dynamic property can also be removed from an instance at runtime. In the case of dynamic properties, if a class makes provision to do so by defining a series of special methods, it can deal with the allocation and management of storage for those properties, by storing them in another object or in a database, for example. (The default behavior is for the Engine to allocate a VSlot for each one). This is called class-specific dynamic allocation. Otherwise, the Engine takes care of the storage in some unspecified manner. Dynamic method handling is only possible when ** class-specific dynamic allocation is used.*

Consider the following scenario, which involves dynamic properties:

```
class Point { ... } // has no public property "color", but has made
                    // provision to support dynamic properties.
$p = new Point(10, 15);
$p->color = "red"; // create/set the dynamic property "color"
$v = $p->color;    // get the dynamic property "color"
isset($p->color);  // test if the dynamic property "color" exists
unset($p->color);  // remove the dynamic property "color"
```

*For the ** class-specific dynamic allocation scenario, when a property name that is not currently visible (because it is hidden or it does not exist) is used in a modifiable lvalue context (as with the assignment of "red"), the Engine generates a call to the instance method `__set` (§§). This method treats that name as designating a dynamic property of the instance being operated on, and sets its value to "red", creating the property, if necessary. Similarly, in a non-lvalue context, (as with the assignment* *of color to $v), the Engine generates a call to the instance method `__get` (§§), which treats that name as designating a dynamic property of the instance being operated on, and gets its value. In the case of the call to the intrinsic `isset` (§§), this generates a call to the instance method `__isset` (§§), while a call to the intrinsic `unset` (§§) generates a call to the instance method `__unset` (§§). By defining these four special methods, the implementer of a class can control how dynamic properties are handled. For the non-class-specific dynamic allocation scenario, the process is like that above except that no special `__*` methods are called.*

In the case of a dynamic method, no method is really added to the instance or the class. However, the illusion of doing that is achieved by allowing a call to an instance or static method, but one which is not declared in that instance's class, to be accepted, intercepted by a method called `__call` (§§) or `__callStatic` (§§), and dealt with under program control.

Consider the following code fragment, in which class Widget has neither an instance method called `iMethod` nor a static method called `sMethod`, but that class has made provision to deal with dynamic methods:

```
$obj = new Widget;
$obj->iMethod(10, TRUE, "abc");
Widget::sMethod(NULL, 1.234);
```

The call to `iMethod` is treated as if it were

```
$obj->__call('iMethod', array(10, TRUE, "abc"))
```

and the call to `sMethod` is treated as if it were

```
Widget::__callStatic('sMethod', array(NULL, 1.234))
```

Constants

Syntax

```
  const-declaration:
    const  name  =  const-expression   ;
```

name is defined in (§§). const-expression is defined in (§§).

Constraints:

A const-declaration must only appear at the top level of a script, be a class constant (inside a class-definition; §§) or be an interface constant (inside an interface-definition; §§).

A const-declaration must not redefine an existing c-constant (§§).

A class constant must not have an explicit visibility specifier (§§).

A class constant must not have an explicit `static` specifier.

Semantics:

A const-declaration defines a c-constant.

All class constants have public visibility.

All constants are implicitly `static`.

Examples:

```
const MIN_VAL = 20;
const LOWER = MIN_VAL;
// ----------------------------------------
class Automobile
{
  const DEFAULT_COLOR = "white";
  ...
}
$col = Automobile::DEFAULT_COLOR;
```

Properties

Syntax

```
  property-declaration:
    property-modifier   name   property-initializer_opt  ;

  property-modifier:
    var
    visibility-modifier   static-modifier_opt
    static-modifier   visibility-modifier_opt

  visibility-modifier:
    public
    protected
    private

  static-modifier:
    static

  property-initializer:
    =  constant-expression
```

name is described in §§ and *constant-expression* is described in §§.

Semantics

A *property-declaration* defines an instance or static property.

The visibility modifiers are described in §§. If *visibility-modifier* is omitted, public is assumed. The var modifier implies public visibility. The `static` modifier is described in §§.

The *property-initializer*s for instance properties are applied prior to the class's constructor being called.

An instance property that is visible may be unset (§§), in which case, the property is actually removed from that instance.

Examples

```
class Point
{
  private static $pointCount = 0; // static property with initializer

  private $x; // instance property
  private $y; // instance property
  ...

}
```

Methods

Syntax

```
method-declaration:
  method-modifiersopt   function-definition
  method-modifiers   function-definition-header   ;

method-modifiers:
  method-modifier
  method-modifiers   method-modifier

method-modifier:
  visibility-modifier
  static-modifier
  abstract
  final
```

visibility-modifier is described in §§; *static-modifier* is described in §§; and *function-definition* and *function-definition-header* are defined in §§.

Constraints

When defining a concrete class that inherits from an abstract class, the definition of each abstract method inherited by the derived class must have the same or a less-restricted visibility than in the corresponding abstract declaration. Furthermore, the signature of a method definition must match that of its abstract declaration.

The *method-modifiers* preceding a *function-definition* must not contain the `abstract` modifier.

The *method-modifiers* preceding a *function-definition-header* must contain the `abstract` modifier.

A method must not have the same modifier specified more than once. A method must not have more than one *visibility-modifier*. A method must not have both the modifiers `abstract` and `private`, or `abstract` and `final`.

Semantics

A *method-declaration* defines an instance or static method. A method is a function that is defined inside a class. However, the presence of `abstract` indicates an abstract method, in which case, no implementation is provided. The absence of `abstract` indicates a concrete method, in which case, an implementation is provided.

Method names are case-insensitive.

The presence of `final` indicates the method cannot be overridden in a derived class.

If *visibility-modifier* is omitted, `public` is assumed.

Examples

See §§ for examples of instance and static methods. See §§ for examples of abstract methods and their subsequent definitions.

Constructors

Syntax

```
constructor-definition:
  visibility-modifier  function &opt   __construct  (  parameter-declaration-listopt  )  compound-statement
  visibility-modifier  function &opt    name  (  parameter-declaration-listopt  )  compound-statement     [Deprecate
```

visibility-modifier is described in [§§](#); *parameter-declaration-list* is described in [§§](#); and *compound-statement* is described in [§§](#). *name* is described in [§§](#).

Constraints

An overriding constructor in a derived class must have the same or a less-restricted [visibility](#) than that being overridden in the base class.

name must be the same as that in the *class-declaration* ([§§](#)) that contains this *constructor-definition*.

Semantics

A constructor is a specially named instance method ([§§](#)) that is used to initialize an instance immediately after it has been created. Any instance properties not explicitly initialized by a constructor take on the value `NULL`. Like a method, a constructor can return a result by value or byRef. (Unlike a method, a constructor cannot be abstract or static).

If *visibility-modifier* is omitted, `public` is assumed. A `private` constructor inhibits the creation of an instance of the class type.

Constructors can be overridden in a derived class by redeclaring them. However, an overriding constructor need not have the same signature as defined in the base class.

Constructors are called by *object-creation-expression*s ([§§](#)) and from within other constructors.

If classes in a derived-class hierarchy have constructors, it is the responsibility of the constructor at each level to call the constructor in its base-class explicitly, using the notation `parent::__construct(...)`. If a constructor calls its base-class constructor, it should do so as the first statement in *compound-statement*, so the object hierarchy is built from the bottom-up. A constructor should not call its base-class constructor more than once. A call to a base-class constructor searches for the nearest constructor in the class hierarchy. Not every level of the hierarchy need have a constructor.

Prior to the addition of the `__construct` form of constructor, a class's constructor was called the same as its class name. For example, class `Point`'s constructor was called `Point`. Although this old-style form is supported, its use is deprecated. In any event, both `parent::__construct(...)` and `parent::name(...)` (where `name` is the name of the parent class type) will find an old- or a new-style constructor in the base class, if one exists. If both forms exist, the new-style one is used. The same is true of an *object-creation-expression* when searching for a base-class constructor.

Examples

```
class Point
{
  private static $pointCount = 0;
  private $x;
  private $y;
  public function __construct($x = 0, $y = 0)
  {
    $this->x = $x;
    $this->y = $y;
    ++self::$pointCount;
  }
  public function __destruct()
  {
    --self::$pointCount;
    ...
  }
  ...
}
// -----------------------------------------
```

```
class MyRangeException extends Exception
{
  public function __construct($message, ...)
  {
    parent::__construct($message);
    ...
  }
  ...
}
```

Destructors

Syntax

```
  destructor-definition:
    visibility-modifier  function  &opt  __destruct  ( ) compound-statement
```

visibility-modifier is described in §§ and compound-statement is described in §§.

Constraints

An overriding destructor in a derived class must have the same or a less-restricted visibility than that being overridden in the base class.

Semantics

A destructor is a special-named instance method (§§) that is used to free resources when an instance is no longer needed. The destructors for instances of all classes are called automatically once there are no handles pointing to those instances or in some unspecified order during program shutdown. Like a method, a destructor can return a result by value or byRef. (Unlike a method, a destructor cannot be abstract or static).

If visibility-modifier is omitted, `public` *is assumed.*

Destructors can be overridden in a derived class by redeclaring them.

Destructors are called by the Engine or from within other destructors.

If classes in a derived-class hierarchy have destructors, it is the responsibility of the destructor at each level to call the destructor in the base-class explicitly, using the notation `parent::__destruct()`*. If a destructor calls its base-class destructor, it should do so as the last statement in compound-statement, so the object hierarchy is destructed from the top-down. A destructor should not call its base-class destructor more than once. A call to a base-class destructor searches for the nearest destructor in the class hierarchy. Not every level of the hierarchy need have a destructor. A* `private` *destructor inhibits destructor calls from derived classes.*

Any dynamic properties (§14.4, §14.10.8) having an object type, and whose parent instances exist when the program terminates will have their destructors (if any) called as part of the cleanup of the parent instances, even if the parent class type has no destructor defined.

Examples

See §§ for an example of a constructor and destructor.

Methods with Special Semantics

General

If a class contains a definition for a method having one of the following names, that method must have the prescribed visibility, signature, and semantics:

Method Name	Description	Reference
`__call`	*Calls a dynamic method in the context of an instance-method call*	§§
`__callStatic`	*Calls a dynamic method in the context of a static-method call*	§§
`__clone`	*Typically used to make a deep copy (§§) of an object*	§§
`__construct`	*A constructor*	§§
`__destruct`	*A destructor*	§§
`__get`	*Retrieves the value of a given dynamic property*	§§
`__invoke`	*Called when a script calls an object as a function*	§§
`__isset`	*Reports if a given dynamic property exists*	§§
`__set`	*Sets the value of a given dynamic property*	§§
`__set_state`	*Called when a class is exported by `var_export` (§xx)*	§§
`__sleep`	*Executed before serialization (§§) of an instance of this class*	§§
`__toString`	*Returns a string representation of the instance on which it is called*	§§
`__unset`	*Removes a given dynamic property*	§§
`__wakeup`	*Executed after unserialization (§§) of an instance of this class*	§§

Method `__call`

Syntax

```
public  function  __call  (  $name  ,  $arguments  )  compound-statement
```

compound-statement is described in §§.

Constraints

The argument corresponding to `$name` must have type `string`, and that corresponding to `$arguments` must have type `array`.

The arguments passed to this method must not be passed byRef.

Semantics

This instance method is called to invoke the dynamic method (§§) designated by `$name` using the arguments specified by the elements of the array designated by `$arguments`. It can return any value deemed appropriate.

Typically, `__call` is called implicitly, when the `->` operator (§§) is used to call an instance method that is not visible. Now while `__call` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `p->m(...)`, where `p` is an instance and `m` is an instance-method name. If `m` is the name of a visible method, `p->m(...)` does not result in `__call`'s being called. Instead, the visible method is used. On the other hand, the expression `p->__call('m',array(...))` always calls the named dynamic method, ignoring the fact that a visible method having the same name might exist. If `m` is not the name of a visible method, the two expressions are equivalent; that is; when handling `p->m(...)`, if no visible method by that name is found, a dynamic method is assumed, and `__call` is called. (Note: While it would be unusual to create deliberately a dynamic method with the same name as a visible one, the visible method might be *added later. This name "duplication" is convenient when adding a dynamic method to a class without having to worry about a name clash with any method names that class inherits).*

While a method-name source token has a prescribed syntax, there are no restrictions on the spelling of the dynamic method name designated by $name. Any source character is allowed here.

Examples

```
class Widget
{
  public function __call($name, $arguments)
  {
    // using the method name and argument list, redirect/process
    // the method call, as desired.
  }
  ...
}
$obj = new Widget;
$obj->iMethod(10, TRUE, "abc"); // $obj->__call('iMethod', array(...))
```

Method `__callStatic`

Syntax

```
  public  static  function  __callStatic  (  $name  ,  $arguments  )   compound-statement
```

compound-statement is described in §§.

Constraints

The argument corresponding to `$name` must have type `string`, and that corresponding to `$arguments` must have type `array`.

The arguments passed to this method must not be passed byRef.

Semantics

This static method is called to invoke the dynamic method (§§) designated by `$name` using the arguments specified by the elements of the array designated by `$arguments`. It can return any value deemed appropriate.

Typically, `__callStatic` is called implicitly, when the `::` operator (§§) is used to call a static method that is not visible. Now while `__callStatic` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `C::m(...)`, where `C` is a class and `m` is a static-method name. If `m` is the name of a visible method, `C::m(...)` does not result in `__callStatic`'s being called. Instead, the visible method is used. On the other hand, the expression `C::__callStatic('m',array(...))` always calls the named dynamic method, ignoring the fact that a static visible method having the same name might exist. If m is not the name of a visible method, the two expressions are equivalent; that is; when handling `C::m(...)`, if no visible method by that name is found, a dynamic method is assumed, and `__callStatic` is called. (Note: While it would be unusual to create deliberately a static dynamic method with the same name as a static visible one, the visible method might be added later. This name "duplication" is convenient when adding a dynamic method to a class without having to worry about a name clash with any method names that class inherits).

While a method-name source token has a prescribed syntax, there are no restrictions on the spelling of the dynamic method name designated by `$name`. Any source character is allowed here.

Examples

```
class Widget
```

```
{
    public static function __callStatic($name, $arguments)
    {
      // using the method name and argument list, redirect/process\
      // the method call, as desired.
    }
    ...
}

Widget::sMethod(NULL, 1.234); // Widget::__callStatic('sMethod', array(...))
```

Method `__clone`

Syntax

```
  public  function  __clone  (  )  compound-statement
```

compound-statement is described in §§.

Semantics

This instance method is called by the `clone` operator (§§), (typically) to make a deep copy (§§) of the current class component of the instance on which it is called. (Method `__clone` cannot be called directly by the program).

Consider a class `Employee`, from which is derived a class `Manager`. Let us assume that both classes contain properties that are objects. To make a copy of a `Manager` object, its `__clone` method is called to do whatever is necessary to copy the properties for the `Manager` class. That method should, in turn, call the `__clone` method of its parent class, `Employee`, so that the properties of that class can also be copied (and so on, up the derived-class hierarchy).

To clone an object, the `clone` operator makes a shallow copy (§§) of the object on which it is called. Then, if the class of the instance being cloned has a method called `__clone`, that method is automatically called to make a deep copy. Method `__clone` cannot be called directly from outside a class; it can only be called by name from within a derived class, using the notation `self::__clone()`. This method can return a value; however, if it does so and control returns directly to the point of invocation via the `clone` operator, that value will be ignored. The value returned to a `self::__clone()` call can, however, be retrieved.

While cloning creates a new object, it does so without using a constructor, in which case, code may need to be added to the `__clone` method to emulate what happens in a corresponding constructor. (See the `Point` example below).

An implementation of `__clone` should factor in the possibility of an instance having dynamic properties (§§).

Examples

```
class Employee
{
  ...
  public function __clone()
  {
    // do what it takes here to make a copy of Employee object properties
  }
}
class Manager extends Employee
{
  ...
  public function __clone()
  {
    parent::__clone(); // request cloning of the Employee properties

    // do what it takes here to make a copy of Manager object properties
```

```
  }
  ...
}
// ----------------------------------------
class Point
{
  private static $pointCount = 0;
  public function __construct($x = 0, $y = 0)
  {
    ...
    ++self::$pointCount;
  }
  public function __clone()
  {
    ++self::$pointCount; // emulate the constructor
  }
  ...
}
$p1 = new Point;  // created using the constructor
$p2 = clone $p1;  // created by cloning
```

Method `__get`

Syntax

```
  public  function  &opt  __get  (  $name  )   compound-statement
```

compound-statement is described in §§.

Constraints

The argument passed to this method must have type `string` and be passed by value.

Semantics

This instance method gets the value of the dynamic property (§§) designated by `$name`. If no such dynamic property currently exists, NULL is returned.

Typically, `__get` is called implicitly, when the `->` operator (§§) is used in a non-lvalue context and the named property is not visible. Now while `__get` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `$v = $p->m`, where `p` is an instance and `m` is a property name. If `m` is the name of a visible property, `p->m` does not result in `__get`'s being called. Instead, the visible property is used. On the other hand, the expression `p->__get('m')` always gets the value of the named dynamic property, ignoring the fact that a visible property having the same name might exist. If `m` is not the name of a visible property, the two expressions are equivalent; that is; when handling `p->m` in a non-lvalue context, if no visible property by that name is found, a dynamic property is assumed, and `__get` is called.

Consider the expression $`v = $p->m = 5`, where `m` is a dynamic property. While `__set` (§§) is called to assign the value 5 to that property, `__get` is not called to retrieve the result after that assignment is complete.

If the dynamic property is an array, `__get` should return byRef, so subscripting can be done correctly on the result.

Examples

```
class Point
{
    private $dynamicProperties = array();
    private $x;
    private $y;
    public function __get($name)
    {
```

```
        if (array_key_exists($name, $this->dynamicProperties))
        {
            return $this->dynamicProperties[$name];
        }

        // no-such-property error handling goes here
        return NULL;
    }
  ...
}
```

Implementation Notes

*Consider the following class, which does **not** contain a property called prop:*

```
class C
{
  public function __get($name)
  {
    return $this->$name;    // must not recurse
  }
  ...
}
$c = new C;
$x = $c->prop;
```

As no property (dynamic or otherwise) by the name prop exists in the class and a `__get` method is defined, this looks look a recursive situation. However, the implementation must not allow that. The same applies to seemingly self-referential implementations of `__set` (§§), `__isset` (§§), and `__unset` (§§).

Method `__invoke`

Syntax

```
public  function  __invoke  ( parameter-declaration-list_opt  )  compound-statement
```

parameter-declaration-list is defined in §§; compound-statement is described in §§.

Semantics

This instance method allows an instance to be used with function-call notation. An instance whose class provides this method will return `TRUE` when passed to `is_callable` (§xx); otherwise, `FALSE` is returned.

When an instance is called as a function, the argument list used is made available to `__invoke`, whose return value becomes the value of the initial function call.

Examples

```
class C
{
  public function __invoke($p)
  {
    ...
    return ...;
  }
  ...
}
$c = new C;
is_callable($c) // returns TRUE
```

```
$r = $c(123);   // becomes $r = $c->__invoke(123);
```

Method `__isset`

Syntax

```
public  function  __isset  (  $name  )  compound-statement
```

compound-statement is described in §§.

Constraints

The argument passed to this method must have type `string` and be passed by value.

Semantics

If the dynamic property (§§) designated by `$name` exists, this instance method returns `TRUE`; otherwise, `FALSE` is returned.

Typically, `__isset` is called implicitly, when the intrinsic `isset` (§§) is called with an argument that designates a property that is not visible. (It can also be called by the intrinsic empty (§§)). Now while `__isset` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `isset($p->m)`, where `p` is an instance and `m` is a property name. If `m` is the name of a visible property, `__isset` is not called. Instead, the visible property is used. On the other hand, the expression `p->__isset('m')` always tests for the named dynamic property, ignoring the fact that a visible property having the same name might exist. If `m` is not the name of a visible property, the two expressions are equivalent; that is; when handling `p->m` in a non-lvalue context, if no visible property by that name is found, a dynamic property is assumed.

Examples

```
class Point
{
    private $dynamicProperties = array();
    private $x;
    private $y;
    public function __isset($name)
    {
        return isset($this->dynamicProperties[$name]);
    }
  ...
}
```

Implementation Notes

See the Implementation Notes for `__get` (§§).

Method `__set`

Syntax

```
public  function  __set  (  $name  ,  $value  )  compound-statement
```

compound-statement is described in §§.

Constraints

The arguments passed to this method must not be passed byRef.

The argument corresponding to `$name` must have type `string`.

Semantics

This instance method sets the value of the dynamic property (§§) designated by `$name` to `$value`. If no such dynamic property currently exists, it is created. No value is returned.

Typically, `__set` is called implicitly, when the `->` operator (§§) is used in a modifiable lvalue context and the named property is not visible. Now while `__set` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `p->m = 5`, where `p` is an instance and `m` is a property name. If `m` is the name of a visible property, `p->m` does not result in `__set`'s being called. Instead, the visible property is used. On the other hand, the expression `p->__set('m',5)` always sets the value of the named dynamic property, ignoring the fact that a visible property having the same name might exist. If `m` is not the name of a visible property, the two expressions are equivalent; that is; when handling `p->m`, if no visible property by that name is found, a dynamic property is assumed, and `__set` is called. (Note: While it would be unusual to create deliberately a dynamic property with the same name as a visible one, the visible property might be added later. This name "duplication" is convenient when adding a dynamic property to a class without having to worry about a name clash with any property names that class inherits).

The parameter `$value` can have any type including an object type, and that type could have a destructor. Any dynamic properties of such types, whose parent instances exist when the program terminates will have their destructors called as part of the cleanup of the parent instances, even if the parent class type has no destructor defined.

While a property-name source token has a prescribed syntax, there are no restrictions on the spelling of the dynamic property name designated by `$name`. Any source character is allowed here.

Examples

```
class Point
{
    private $dynamicProperties = array();
    private $x;
    private $y;
    public function __set($name, $value)
    {
        $this->dynamicProperties[$name] = $value;
    }
  ...
}
// ----------------------------------------
class X
{
    public function __destruct() { ... }
}
$p = new Point(5, 9);
$p->thing = new X;  // set dynamic property "thing" to instance with destructor
...
// at the end of the program, p->thing's destructor is called
```

Implementation Notes

See the Implementation Notes for `__get` (§§).

Method `__set_state`

Syntax

```
static public  function  __set_state  ( array  $properties  )  compound-statement
```

compound-statement is described in §§.

Constraints

`$properties` must contain a key/value pair for each instance property in the class and all its direct and indirect base classes, where each key is the name of a property in that class.

Semantics

This function supports the library function `var_export` (§xx) when it is given an instance of this class type. `var_export` takes a variable and produces a string representation of that variable as valid PHP code suitable for use with the intrinsic `eval` (§§).

For an object, the string returned by `var_export` has the following general format:

`classname::__set_state(array('prop1' => value, ..., 'propN' => value , ))`

where the property names `prop1` through `propN` do not include a leading dollar (`$`). This string contains a call to the `__set_state` method even if no such method is defined for this class or in any of its base classes, in which case, a subsequent call to `eval` using this string will fail. To allow the string to be used with eval, the method `__set_state` must be defined, and it must create a new instance of the class type, initialize its instance properties using the key/value pairs in `$properties`, and it must return that new object.

If a derived class does not define a `__set_state` method, a call to it will look for such a method in the base class hierarchy, and that method will return an instance of the appropriate base class, not of the class on which it was invoked. This is probably not what the programmer expected. If a derived class defines a `__set_state` method, but any base class has instance properties that are not visible within that method, that method must invoke parent's `__set_state` as well, but that can require support from a base class. See the second example below.

Examples

```
class Point
{
  private $x;
  private $y;
  static public function __set_state(array $properties)
  {
    $p = new Point;
    $p->x = $properties['x'];
    $p->y = $properties['y'];
    return $p;
  }
  ...
}
$p = new Point(3, 5);
$v = var_export($p, TRUE);  // returns string representation of $p
```

The string produced looks something like the following:

```
"Point::__set_state(array(
   'x' => 3,
   'y' => 5,
))"
eval('$z = ' . $v . ";"); // execute the string putting the result in $z
echo "Point \$z is $z\n"; // Point $z is (3,5)
// -----------------------------------------
class B // base class of D
{
  private $bprop;
  public function __construct($p)
```

```
  {
    $this->bprop = $p;
  }
  static public function __set_state(array $properties)
  {
    $b = new static($properties['bprop']);  // note the static
    return $b;
    // Because of the "new static", the return statement
    //   returns a B when called in a B context, and
    //   returns a D when called in a D context
  }
}
class D extends B
{
  private $dprop = 123;
  public function __construct($bp, $dp = NULL)
  {
    $this->dprop = $dp;
    parent::__construct($bp);
  }
  static public function __set_state(array $properties)
  {
    $d = parent::__set_state($properties); // expects back a D, NOT a B
    $d->dprop = $properties['dprop'];
    return $d;
  }
}
$b = new B(10);
$v = var_export($b, TRUE);
eval('$z = ' . $v . ";");
$d = new D(20, 30);
$v = var_export($d, TRUE);
eval('$z = ' . $v . ";");
```

Method `__sleep`

Syntax

```
public  function  __sleep  ( ) compound-statement
```

compound-statement is described in §§.

Semantics

The instance methods `__sleep` and `__wakeup` (§§) support serialization (§§).

If a class has a `__sleep` method, the library function `serialize` (§xx) calls that method to find out which visible instance properties it should serialize. (In the absence of a `__sleep` or `serialize` method, all such properties are serialized, including any dynamic properties (§§)). This information is returned by `__sleep` as an array of zero or more elements, where each element's value is distinct and is the name of a visible instance property. These properties' values are serialized in the order in which the elements are inserted in the array. If `__sleep` does not return a value explicitly, `NULL` is returned, and that value is serialized.

Besides creating the array of property names, `__sleep` can do whatever else might be needed before serialization occurs.

Consider a `Point` class that not only contains x- and y-coordinates, it also has an `id` property; that is, each distinct `Point` created during a program's execution has a unique numerical id. However, there is no need to include this when a `Point` is serialized. It can simply be recreated when that `Point` is unserialized. This information is transient and need not be preserved across program executions. (The same can be true for other transient properties, such as those that contain temporary results or run-time caches).

In the absence of methods `__sleep` and `__wakeup`, instances of derived classes can be serialized and unserialized. However, it is not possible to perform customize serialization using those methods for such instances. For that, a class must implement the interface Serializable (§§).

Examples

```
class Point
{
  private static $nextId = 1;
  private $x;
  private $y;
  private $id;
  public function __construct($x = 0, $y = 0)
  {
    $this->x = $x;
    $this->y = $y;
    $this->id = self::$nextId++;  // assign the next available id
  }
  public function __sleep()
  {
    return array('y', 'x'); // serialize only $y and $x, in that order
  }
  public function __wakeup()
  {
    $this->id = self::$nextId++;  // assign a new id
  }
  ...
}
$p = new Point(-1, 0);
$s = serialize($p);   // serialize Point(-1,0)
$v = unserialize($s); // unserialize Point(-1,0)
```

Method `__toString`

Syntax

```
public  function  __toString  ( )  compound-statement
```

compound-statement is described in §§.

Constraints

This function must return a string.

This function must not throw any exceptions.

Semantics

This instance method is intended to create a string representation of the instance on which it is called. If the instance's class is derived from a class that has or inherits a `__toString` method, the result of calling that method should be prepended to the returned string.

`__toString` is called by a number of language and library facilities, including `echo`, when an object-to-string conversion is needed. `__toString` can be called directly.

An implementation of `__toString` should factor in the possibility of an instance having dynamic properties (§§).

Examples

```
class Point
{
  private $x;
  private $y;
  public function __construct($x = 0, $y = 0)
  {
    $this->x = $x;
    $this->y = $y;
  }
  public function __toString()
  {
    return '(' . $this->x . ',' . $this->y . ')';
  }
  ...
}
$p1 = new Point(20, 30);
echo $p1 . "\n";  // implicit call to __toString() returns "(20,30)"
// -----------------------------------------
class MyRangeException extends Exception
{
  public function __toString()
  {
    return parent::__toString()
      . string-representation-of-MyRangeException
  }
  ...
}
```

Method `__unset`

Syntax

```
public  function  __unset  (  $name  )  compound-statement
```

compound-statement is described in §§.

Constraints

The argument passed to this method must have type `string` and be passed by value.

Semantics

If the dynamic property (§§) designated by `$name` exists, it is removed by this instance method; otherwise, the call has no effect. No value is returned.

Typically, `__unset` is called implicitly, when the intrinsic `unset` (§§) is called with an argument that designates a property that is not visible. Now while `__unset` can be called explicitly, the two scenarios do not necessarily produce the same result. Consider the expression `unset($p->m)`, where `p` is an instance and `m` is a property name. If `m` is the name of a visible property, `__unset` is not called. Instead, the visible property is used. On the other hand, the expression `p->__unset('m'))` always removes the named dynamic property, ignoring the fact that a visible property having the same name might exist. If `m` is not the name of a visible property, the two expressions are equivalent; that is; when handling `p->m` in a non-lvalue context, if no visible property by that name is found, a dynamic property is assumed.

Examples

```
class Point
{
    private $dynamicProperties = array();
    private $x;
```

```
    private $y;
    public function __unset($name)
    {
        unset($this->dynamicProperties[$name]);
    }
  ...
}
```

Implementation Notes

See the Implementation Notes for `__get` (§§).

Method `__wakeup`

Syntax

```
public  function  __wakeup  ( )  compound-statement
```

compound-statement is described in §§.

Constraints

Xx

Semantics

The instance methods `__sleep` (§§) and `__wakeup` support serialization (§§).

*When the library function `unserialize` (§xx) is called on the string representation of an object, as created by the library function `serialize` (§xx), `unserialize` creates an instance of that object's type **without calling a constructor**, and then calls that class's `__wakeup` method, if any, to initialize the instance. In the absence of a `__wakeup` method, all that is done is that the values of the instance properties encoded in the serialized string are restored.*

Consider a `Point` class that not only contains x- and y-coordinates, it also has an `id` property; that is, each distinct `Point` created during a program's execution has a unique numerical id. However, there is no need to include this when a `Point` is serialized. It can simply be recreated by `__wakeup` when that `Point` is unserialized. This means that `__wakeup` must emulate the constructor, as appropriate.

`__wakeup` does not return a value.

Examples

See §§.

Serialization

In PHP, variables can be converted into some external form suitable for use in file storage or inter-program communication. The process of converting to this form is known as serialization while that of converting back again is known as unserialization. These facilities are provided by the library functions `serialize` (§xx) and `unserialize` (§xx), respectively.

In the case of variables that are objects, on their own, these two functions serialize and unserialize all the instance properties, which may be sufficient for some applications. However, if the programmer wants to customize these processes, they can do so in one of two, mutually exclusive ways. The first approach is to define methods called `__sleep` and `__awake`, and have them get control before serialization and after serialization, respectively. For information on this approach, see §§ and §§. The second approach involves implementing the interface `Serializable` (§§) by defining two methods, `serialize` and `unserialize`.

Consider a `Point` class that not only contains x- and y-coordinates, it also has an `id` property; that is, each distinct `Point` created during a program's execution has a unique numerical id. However, there is no need to include this when a `Point` is serialized. It can simply be recreated when that `Point` is unserialized. This information is transient and need not be preserved across program executions. (The same can be true for other transient properties, such as those that contain temporary results or run-time caches). Furthermore, consider a class `ColoredPoint` that extends `Point` by adding a `color` property. The following code shows how these classes need be defined in order for both `Points` and `ColoredPoints` to be serialized and unserialized:

```
class Point implements Serializable // note the interface
{
  private static $nextId = 1;
  private $x;
  private $y;
  private $id;  // transient property; not serialized
  public function __construct($x = 0, $y = 0)
  {
    $this->x = $x;
    $this->y = $y;
    $this->id = self::$nextId++;
  }
  public function __toString()
  {
    return 'ID:' . $this->id . '(' . $this->x . ',' . $this->y . ')';
  }
  public function serialize()
  {
    return serialize(array('y' => $this->y, 'x' => $this->x));
  }
```

The custom method `serialize` calls the library function `serialize` to create a string version of the array, whose keys are the names of the instance properties to be serialized. The insertion order of the array is the order in which the properties are serialized in the resulting string. The array is returned.

```
  public function unserialize($data)
  {
    $data = unserialize($data);
    $this->x = $data['x'];
    $this->y = $data['y'];
    $this->id = self::$nextId++;
  }
}
```

The custom method `unserialize` converts the serialized string passed to it back into an array. Because a new object is being created, but without any constructor being called, the `unserialize` method must perform the tasks ordinarily done by a constructor. In this case, that involves assigning the new object a unique id.

```
$p = new Point(2, 5);
$s = serialize($p);
```

The call to the library function `serialize` calls the custom `serialize` method. Afterwards, the variable `$s` contains the serialized version of the `Point(2,5)`, and that can be stored in a database or transmitted to a cooperating program. The program that reads or receives that serialized string can convert its contents back into the corresponding variable(s), as follows:

```
$v = unserialize($s);
```

The call to the library function `unserialize` calls the custom `unserialize` method. Afterwards, the variable `$s` contains a new `Point(2,5)`.

```
class ColoredPoint extends Point implements Serializable
{
  const RED = 1;
  const BLUE = 2;

  private $color; // an instance property

  public function __construct($x = 0, $y = 0, $color = RED)
  {
    parent::__construct($x, $y);
    $this->color = $color;
  }

  public function __toString()
  {
    return parent::__toString() . $this->color;
  }

  public function serialize()
  {
    return serialize(array(
      'color' => $this->color,
      'baseData' => parent::serialize()
    ));
  }
```

As with class `Point`, this custom method returns an array of the instance properties that are to be serialized. However, in the case of the second element, an arbitrary key name is used, and its value is the serialized version of the base Point within the current `ColoredPoint` object. The order of the elements is up to the programmer.

```
    public function unserialize($data)
    {
    $data = unserialize($data);
    $this->color = $data['color'];
    parent::unserialize($data['baseData']);
    }
}
```

As `ColoredPoint` has a base class, it unserializes its own instance properties before calling the base class's custom method, so it can unserialize the `Point` properties.

```
$cp = new ColoredPoint(9, 8, ColoredPoint::BLUE);
$s = serialize($cp);
...
$v = unserialize($s);
```

Predefined Classes

Class `Closure`

The predefined class `Closure` is used for representing an anonymous function. It cannot be instantiated except by the Engine, as described below.

```
class Closure
```

```
{
  public static bind(Closure $closure, $newthis [, $newscope = "static" ]);
  public bindTo($newthis [, $newscope = "static" ]);
}
```

The class members are defined below:

Name	Purpose
`bind`	*Duplicates closure `$closure` with a specific bound object `$newthis` and class scope `$newscope`. Make `$newthis` `NULL` if the closure is to be unbound. `$newscope` is the class scope to which the closure is to be associated, or static to keep the current one. If an object is given, the type of the object will be used instead. This determines the visibility of protected and private methods of the bound object. Returns a new `Closure` object or `FALSE` on failure.*
`bindTo`	*Duplicates the closure designated by the current instance with a new-bound object and class scope. This method is an instance version of bind.*

When the anonymous function-creation operator (§§) is evaluated, the result is an object of type `Closure` (or some unspecified class derived from that type) created by the Engine. This object is referred to here as "the Closure object". This instance encapsulates the anonymous function defined in the corresponding anonymous-function-creation-expression.

The contents of a `Closure` object are determined based on the context in which an anonymous function is created. Consider the following scenario:

```
class C
{
  public function compute()
  {
    $count = 0;
    $values = array("red" => 3, 10);
    $callback = function ($p1, $p2) use (&$count, $values)
    {
      ...
    };
    ...
  }
}
```

A `Closure` object may contain the following, optional dynamic properties, in order: `static`, `this`, and `parameter`.

If an anonymous-function-creation-expression contains an anonymous-function-use-clause, a dynamic property called `static` is present. This property is an array having an element for each variable-name in the use-variable-name-list, inserted in lexical order of their appearance in the use clause. Each element's key is the corresponding variable-name, and each element value is the value of that variable at the time the time the `Closure` object is created (not when it is used to call the encapsulated function). In the scenario above, this leads to the following, shown as pseudo code:

```
$this->static = array(["count"]=>&0,["values"]=>array(["red"]=>3,[0]=>10));
```

If an anonymous-function-creation-expression is used inside an instance method, a dynamic property called `this` is present. This property is a handle that points to the current instance. In the scenario above, this leads to the following, shown as pseudo code:

```
$this->this = $this;
```

If an anonymous-function-creation-expression contains a parameter-declaration-list, a dynamic property called `parameter` is present. This property is an array of one or more elements, each of which corresponds to a parameter. The elements are inserted in that array in lexical order of their declaration. Each element's key is the corresponding parameter name, and each element value is some unspecified value. (These values are overridden by the argument values used when the anonymous function is called). In the scenario above, this leads to the following, shown as pseudo code:

```
$property = array("$p1" => ???, "$p2" => ???)
```

It is possible for all three dynamic properties to be absent, in which case, the `Closure` object is empty.

Class `Generator`

This class supports the `yield` operator (§§). This class cannot be instantiated directly. It is defined, as follows:

class Generator implements Iterator

```
class Generator implements Iterator
{
  public function current();
  public function key();
  public function next();
  public function rewind();
  public function send($value) ;
  public function throw(Exception $exception) ;
  public function valid();
  public function __wakeup();
}
```

The class members are defined below:

Name	Purpose
`current`	*An implementation of the instance method `Iterator::current` (§§).*
`key`	*An implementation of the instance method `Iterator::key` (§§).*
`next`	*An implementation of the instance method Iterator::next (§§).*
`rewind`	*An implementation of the instance method `Iterator::rewind` (§§).*
`send`	*This instance method sends the value designated by `$value` to the generator as the result of the current `yield` expression, and resumes execution of the generator. `$value` is the return value of the `yield` expression the generator is currently at. If the generator is not at a `yield` expression when this method is called, it will first be let to advance to the first `yield` expression before sending the value. This method returns the yielded value.*
`throw`	*This instance method throws an exception into the generator and resumes execution of the generator. The behavior is as if the current `yield` expression was replaced with throw `$exception`. If the generator is already closed when this method is invoked, the exception will be thrown in the caller's context instead. This method returns the yielded value.*
`valid`	*An implementation of the instance method `Iterator::valid` (§§).*
`__wakeup`	*An implementation of the special instance method `__wakeup` (§§). As a generator can't be serialized, this method throws an exception of an unspecified type. It returns no value.*

Class `__PHP_Incomplete_Class`

There are certain circumstances in which a program can generate an instance of this class, which on its own contains no members. One involves an attempt to unserialize (§§, §§) a string that encodes an instance of a class for which there is no definition in scope. Consider the following class, which supports a two-dimensional Cartesian point:

```
class Point
{
  private $x;
  private $y;
  ...
}
$p = new Point(2, 5);
$s = serialize($p); // properties $x and $y are serialized, in that order
```

Let us assume that the serialized string is stored in a database from where it is retrieved by a separate program. That program contains the following code, but does not contain a definition of the class Point:

```
$v = unserialize($s);
```

Instead of returning a point, `Point(2, 5`), an instance of `__PHP_Incomplete_Class` results, with the following contents:

```
__PHP_Incomplete_Class
{
   __PHP_Incomplete_Class_Name => "Point"
  x:Point:private => 2
  y:Point:private => 5
}
```

The three dynamic properties (§§) contain the name of the unknown class, and the name, visibility, and value of each property that was serialized, in order of serialization.

Class `stdClass`

This class contains no members. It can be instantiated and used as a base class. An instance of this type is automatically created when a non-object is converted to an object (§§), or the member-selection operator (§§) is applied to `NULL`, `FALSE`, or an empty string.

Interfaces

General

A class can implement a set of capabilities—herein called a contract—through what is called an interface. An interface is a set of method declarations and constants. Note that the methods are only declared, not defined; that is, an interface defines a type consisting of abstract methods, where those methods are implemented by client classes as they see fit. An interface allows unrelated classes to implement the same facilities with the same names and types without requiring those classes to share a common base class.

An interface can extend one or more other interfaces, in which case, it inherits all members from its base interface(s).

Interface Declarations

Syntax

```
interface-declaration:
  interface   name   interface-base-clause_opt {  interface-member-declarations_opt  }

interface-base-clause:
  extends   qualified-name
  interface-base-clause  ,  qualified-name
```

name and qualified-name are defined in §§. interface-member-declarations is defined in §§.

Constraints

An interface must not be derived directly or indirectly from itself.

qualified-name must name an interface type.

Semantics

An interface-declaration defines a contract that one or more classes can implement.

Interface names are case-insensitive.

The optional interface-base-clause specifies the base interfaces from which the interface being defined is derived. In such a case, the derived interface inherits all the members from the base interfaces.

Examples

```
interface MyCollection
{
  const MAX_NUMBER_ITEMS = 1000;
  function put($item);
  function get();
}
class MyList implements MyCollection
{
  public function put($item)  { /* implement method */ }
  public function get()   { /* implement method */ }
  ...
}
class MyQueue implements MyCollection
{
  public function put($item)  { /* implement method */ }
  public function get()   { /* implement method */ }
  ...
}
function processCollection(MyCollection $p1)
{
  ... /* can process any object whose class implements MyCollection */
}
processCollection(new MyList(...));
processCollection(new MyQueue(...));
```

Interface Members

Syntax

```
interface-member-declarations:
  interface-member-declaration
  interface-member-declarations   interface-member-declaration

interface-member-declaration:
```

```
    const-declaration
    method-declaration
```

const-declaration is defined in §§ and method-declaration is defined in §§.

Semantics

The members of an interface are those specified by its interface-member-declarations, and the members inherited from its base interfaces.

An interface may contain the following members:

- *Constants – the constant values associated with the interface (§§).*
- *Methods – placeholders for the computations and actions that can be performed by implementers of the interface (§§).*

Constants

Semantics:

An interface constant is just like a class constant (§§), except that an interface constant cannot be overridden by a class that implements it nor by an interface that extends it.

Examples:

```
interface MyCollection
{
  const MAX_NUMBER_ITEMS = 1000;
  function put($item);
  function get();
}
```

Methods

Constraints

All methods declared in an interface must be implicitly or explicitly public, and they must not be declared `abstract`.

Semantics:

An interface method is just like an abstract method (§§).

Examples:

```
interface MyCollection
{
  const MAX_NUMBER_ITEMS = 1000;
  function put($item);
  function get();
}
```

Predefined Interfaces

Interface `ArrayAccess`

This interface allows an instance of an implementing class to be accessed using array-like notation. This interface is defined, *as follows:*

```
interface ArrayAccess
{
  function offsetExists($offset);
  function offsetGet($offset);
  function offsetSet($offset, $value);
  function offsetUnset($offset);
}
```

The interface members are defined below:

Name	Purpose
`offsetExists`	*This instance method returns `TRUE` if the instance contains an element with key `$offset`, otherwise, `FALSE`.*
`offsetGet`	*This instance method gets the value having key `$offset`. It may return by value or byRef. (Ordinarily, this wouldn't be allowed because a class implementing an interface needs to match the interface's method signatures; however, the Engine gives special treatment to `ArrayAccess` and allows this). This method is called when an instance of a class that implements this interface is subscripted (§§) in a non-lvalue context.*
`offsetSet`	*This instance method sets the value having key `$offset` to $value. It returns no value. This method is called when an instance of a class that implements this interface is subscripted (§§) in a modifiable-lvalue context.*
`offsetUnset`	*This instance method unsets the value having key `$offset`. It returns no value.*

Interface `Iterator`

This interface allows instances of an implementing class to be treated as a collection. This interface is defined, as follows:

```
interface Iterator extends Traversable
{
  function current();
  function key();
  function next();
  function rewind();
  function valid();
}
```

The interface members are defined below:

Name	Purpose
`current`	*This instance method returns the element at the current position.*
`key`	*This instance method returns the key of the current element. On failure, it returns `NULL`; otherwise, it returns the scalar value of the key.*
`next`	*This instance method moves the current position forward to the next element. It returns no value. From within a `foreach` statement, this method is called after each loop.*
`rewind`	*This instance method resets the current position to the first element. It returns no value. From within a `foreach` statement, this method is called once, at the beginning.*
	This instance method checks if the current position is valid. It takes no arguments. It returns a bool value of

`valid`	*`TRUE` to indicate the current position is valid; `FALSE`, otherwise. This method is called after each call to `Iterator::rewind()` and `Iterator::next()`.*

Interface `IteratorAggregate`

This interface allows the creation of an external iterator. This interface is defined, as follows:

```
Interface IteratorAggregate extends Traversable
{
  function getIterator();
}
```

The interface members are defined below:

Name	Purpose
`getIterator`	*This instance method retrieves an iterator, which implements `Iterator` or `Traversable`. It throws an `Exception` on failure.*

Interface `Traversable`

This interface is intended as the base interface for all traversable classes. This interface is defined, as follows:

```
interface Traversable
{
}
```

This interface has no members.

Interface `Serializable`

This interface provides support for custom serialization. It is defined, as follows:

```
interface Serializable
{
  function serialize();
  function unserialize ($serialized);

}
```

The interface members are defined below:

Name	Purpose
`serialize`	*This instance method returns a string representation of the current instance. On failure, it returns `NULL`.*
`unserialize`	*This instance method constructs an object from its string form designated by `$serialized`. It does not return a value.*

Traits

General

PHP's class model allows single inheritance only (§§) with contracts being enforced separately via interfaces (§§). A trait can provide both implementation and contracts. Specifically, a class can inherit from a base class while getting implementation from one or more traits. At the same time, that class can implement contracts from one or more interfaces as well as from one or more traits. The use of a trait by a class does not involve any inheritance hierarchy, so unrelated classes can use the same trait. In summary, a trait is a set of methods and/or state information that can be reused.

Traits are designed to support classes; a trait cannot be instantiated directly.

The members of a trait each have visibility (§§), which applies once they are used by a given class. The class that uses a trait can change the visibility of any of that trait's members, by either widening or narrowing that visibility. For example, a private trait member can be made public in the using class, and a public trait member can be made private in that class.

Once implementation comes from both a base class and one or more traits, name conflicts can occur. However, trait usage provides a means of disambiguating such conflicts. Names gotten from a trait can also be given aliases.

A class member with a given name overrides one with the same name in any traits that class uses, which, in turn, overrides any such name from base classes.

Traits can contain both instance and static members, including both methods and properties. In the case of a trait with a static property, each class using that trait has its own instance of that property.

Methods in a trait have full access to all members of any class in which that trait is used.

Trait Declarations

Syntax

```
trait-declaration:
  trait   name   {   trait-use-clauses_opt   trait-member-declarations_opt   }

trait-use-clauses:
  trait-use-clause
  trait-use-clauses   trait-use-clause

trait-use-clause:
  use   trait-name-list   trait-use-terminator

trait-name-list:
  qualified-name
  trait-name-list   ,   qualified-name

trait-use-terminator:
  ;
  {   trait-select-and-alias-clauses_opt   }

trait-select-and-alias-clauses:
  trait-select-and-alias-clause
  trait-select-and-alias-clauses   trait-select-and-alias-clause

trait-select-and-alias-clause:
  trait-select-insteadof-clause
  trait-alias-as-clause

trait-select-insteadof-clause:
  name   insteadof   name

trait-alias-as-clause:
  name   as   visibility-modifier_opt   name
  name   as   visibility-modifier   name_opt
```

name is defined in §§; *visibility-modifier* is defined in §§; and *trait-member-declarations* is defined in §§.

Constraints

The names in *trait-name-list* must designate trait names, excluding the name of the trait being declared.

The left-hand name in *trait-select-insteadof-clause* must unambiguously designate a member of a trait made available by *trait-use-clauses*. The right-hand name in *trait-select-insteadof-clause* must unambiguously designate a trait made available by *trait-use-clauses*.

The left-hand name in *trait-alias-as-clause* must unambiguously designate a member of a trait made available by *trait-use-clauses*. The right-hand name in *trait-alias-as-clause* must be a new, unqualified name.

Semantics

A *trait-declaration* defines a named set of members, which are made available to any class that uses that trait.

Trait names are case-insensitive.

A *trait-declaration* may also use other traits. This is done via one or more *trait-use-clauses*, each of which contains a comma-separated list of trait names. A *trait-use-clause* ends in a semicolon or a brace-delimited set of *trait-select-insteadof-clauses* and *trait-alias-as-clauses*.

A *trait-select-insteadof-clause* allows name clashes to be avoided. Specifically, the left-hand name designates which name to be used from of a pair of names. That is, `T1::compute insteadof T2`; indicates that calls to method compute, for example, should be satisfied by a method of that name in trait `T1` rather than `T2`.

A *trait-alias-as-clause* allows a (possibly qualified) name to be assigned a simple alias name. Specifically, the left-hand name in *trait-alias-as-clause* designates a name made available by *trait-use-clauses* ~~ that is to be aliased, and the right-hand name is the alias.

If *trait-alias-as-clause* contains a *visibility-modifier*, that controls the visibility of the alias, if a right-hand name is provided; otherwise, it controls the visibility of the left-hand name.

Examples

```
trait T1 { public function compute( ... ) { ... } }
trait T2 { public function compute( ... ) { ... } }
trait T1 { public function sort( ... ) { ... } }
trait T4
{
  use T3;
  use T1, T2
  {
    T1::compute insteadof T2; // disambiguate between two computes
    T3::sort as private sorter; // make alias with adjusted visibility
  }
}
```

Trait Members

Syntax

```
  trait-member-declarations:
    trait-member-declaration
    trait-member-declarations   trait-member-declaration

  trait-member-declaration:
    property-declaration
```

```
    method-declaration
    constructor-declaration
    destructor-declaration
```

property-declaration is defined in §§; *method-declaration* is defined in §§; *constructor-declaration* is defined in §§; and *destructor-declaration* is defined in §§.

Semantics

The members of a trait are those specified by its *trait-member-declaration*s, and the members from any other traits it uses.

A trait may contain the following members:

- Properties – the variables made available to the class in which the trait is used (§§).
- Methods – the computations and actions that can be performed by the class in which the trait is used (§§, §§).
- Constructor – the actions required to initialize an instance of the class in which the trait is used (§§)
- Destructor – the actions to be performed when an instance of the class in which the trait is used is no longer needed (§§).

If a member has no explicit visibility, `public` is assumed.

Examples

```
trait T
{
  private $prop1 = 1000;
  protected static $prop2;
  var $prop3;
  public function compute( ... ) { ... }
  public static function getData( ... ) { ... }
}
```

Exception Handling

General

An *exception* is some unusual condition in that it is outside the ordinary expected behavior. (Examples include dealing with situations in which a critical resource is needed, but is unavailable, and detecting an out-of-range value for some computation). As such, exceptions require special handling. This clause describes how exceptions can be created and handled.

Whenever some exceptional condition is detected at runtime, an exception is *thrown*. A designated exception handler can *catch* the thrown exception and service it. Among other things, the handler might recover from the situation completely (allowing the script to continue execution), it might perform some recovery and then throw an exception to get further help, or it might perform some cleanup action and terminate the script. Exceptions may be thrown on behalf of the Engine or by explicit code source code in the script.

Exception handling involves the use of the following keywords:

- `try` (§§), which allows a *try-block* of code containing one or more possible exception generations, to be tried
- `catch` (§§), which defines a handler for a specific type of exception thrown from the corresponding try-block or from some function it calls
- `finally` (§§), which allows the *finally-block* of a try-block to be executed (to perform some cleanup, for example), whether or not an exception occurred within that try-block
- `throw` (§§), which generates an exception of a given type, from a place called a *throw point*

When an exception is thrown, an exception object of type `Exception` (§§), or of a subclass of that type, is created and made available to the first catch-handler that can catch it. Among other things, the exception object contains an exception message and an exception code, both of which can be used by a handler to determine how to handle the situation.

Prior to the addition of exception handling to PHP, exception-like conditions were handled using Error Reporting (§xx). Now, errors can be translated to exceptions via the class `ErrorException` (which is not part of this specification).

Class `Exception`

Class `Exception` is the base class of all exception types. This class is defined, as follows:

```
Class Exception
{
  private   $string;
  private   $trace;
  private   $previous;

  protected $message = 'Unknown exception';
  protected $code = 0;
  protected $file;
  protected $line;

  public function __construct($message = "", $code = 0,
               Exception $previous = NULL);

  final private function __clone();

  final public  function getMessage();
  final public  function getCode();
  final public  function getFile();
  final public  function getLine();
  final public  function getTrace();
  final public  function getPrevious();
  final public  function getTraceAsString();
  public function __toString();
}
```

For information about exception trace-back, see §§. For information about nested exceptions, see §§.

The class members are defined below:

Name	Purpose
`$code`	`int` ; the exception code (as provided by the constructor)
`$file`	`string` ; the name of the script where the exception was generated
`$line`	`int` ; the source line number in the script where the exception was generated
`$message`	`string` ; the exception message (as provided by the constructor)
`$previous`	The previous exception in the chain, if this is a nested exception; otherwise, `NULL`
`$string`	Work area for `__toString`
`$trace`	Work area for function-call tracing
`__construct`	Takes three (optional) arguments – `string` : the exception message (defaults to ""), `int` : the exception code (defaults to 0), and `Exception` : the previous exception in the chain (defaults to `NULL`)
`__clone`	Present to inhibit the cloning of exception objects

`__toString`	`string`; retrieves a string representation of the exception in some unspecified format
`getCode`	`mixed`; retrieves the exception code (as set by the constructor). For an exception of type Exception, the returned value has type int; for subclasses of `Exception`, it may have some other type.
`getFile`	`string`; retrieves the name of the script where the exception was generated
`getLine`	`int`; retrieves the source line number in the script where the exception was generated
`getMessage`	`string`; retrieves the exception message
`getPrevious`	`Exception`; retrieves the previous exception (as set by the constructor), if one exists; otherwise, `NULL`
`getTrace`	`array`; retrieves the function stack trace information as an array (see §§)
`getTraceAsString`	`string`; retrieves the function stack trace information formatted as a single string in some unspecified format

Tracing Exceptions

When an exception is caught, the `get*` functions in class `Exception` provide useful information. If one or more nested function calls were involved to get to the place where the exception was generated, a record of those calls is also retained, and made available by getTrace, through what is referred to as the function stack trace, or simply, `*trace*`.

Let's refer to the top level of a script as function-level 0. Function-level 1 is inside any function called from function-level 0. Function-level 2 is inside any function called from function-level 1, and so on. The library function `getTrace` returns an array. Exceptions generated at function-level 0 involve no function call, in which case, the array returned by `getTrace` has zero elements.

Each element of the array returned by `getTrace` provides information about a given function level. Let us call this array trace-array and the number of elements in this array call-level. The key for each of trace-array's elements has type int, and ranges from 0 to call-level - 1. For example, when a top-level script calls function `f1`, which calls function `f2`, which calls function `f3`, which then generates an exception, there are four function levels, 0–3, and there are three lots of trace information, one per call level. That is, trace-array contains three elements, and they each correspond to the reverse order of the function calls. For example, trace-array[0] is for the call to function `f3`, trace-array[1] is for the call to function `f2`, and trace-array[2] is for the call to function `f1`.

Each element in trace-array is itself an array that contains elements with the following key/value pairs:

Key	Value Type	Value
"args"	`array`	The set of arguments passed to the function
"class"	`string`	The name of the function's parent class
"file"	`string`	The name of the script where the function was called
"function"	`string`	The name of the function or class method
"line"	`int`	The line number in the source where the function was called
"object"	`object`	The current object
"type"	`string`	Type of call; `->` for an instance method call, `::` for a static method call, ordinary function call, "" is returned.

As to whether extra elements with other keys are provided is unspecified.

The key `args` has a value that is yet another array, which we shall call argument-array. That array contains a set of values that corresponds directly to the set of values passed as arguments to the corresponding function. Regarding element order, argument-array[0] corresponds to the left-most argument, argument-array[1] corresponds to the next argument to the right, and so on.

Consider the case in which a function has a default argument value defined for a parameter. If that function is called without an argument for the parameter having the default value, no corresponding argument exists in array-argument. Only arguments present at the function-call site have their values recorded in array-argument.

See also, library functions `debug_backtrace` (§xx) and `debug_print_backtrace` (§xx).

User-Defined Exception Classes

An exception class is defined simply by having it extend class `Exception` (§§). However, as that class's `__clone` method is declared `final` (§§), exception objects cannot be cloned.

When an exception class is defined, typically, its constructors call the parent class' constructor as their first operation to ensure the base-class part of the new object is initialized appropriately. They often also provide an augmented implementation of `__toString()` (§§).

Namespaces

General

A problem encountered when managing large projects is that of avoiding the use of the same name in the same scope for different purposes. This is especially problematic in a language that supports modular design and component libraries.

A namespace is a container for a set of (typically related) definitions of classes, interfaces, traits, functions, and constants. Namespaces serve two purposes:

- *They help avoid name collisions.*
- *They allow certain long names to be accessed via shorter, more convenient and readable, names.*

A namespace may have sub-namespaces, where a sub-namespace name shares a common prefix with another namespace. For example, the namespace `Graphics` might have sub-namespaces `Graphics\D2` and `Graphics\D3`, for two- and three-dimensional facilities, respectively. Apart from their common prefix, a namespace and its sub-namespaces have no special relationship. The namespace whose prefix is part of a sub-namespace need not actually exist for the sub-namespace to exist. That is, `NS1\Sub` can exist without `NS1`.

In the absence of any namespace definition, the names of subsequent classes, interfaces, traits, functions, and constants are in the default namespace, which has no name, per se.

The namespaces PHP, php, and sub-namespaces beginning with those prefixes are reserved for use by PHP.

Name Lookup

When an existing name is used in source code, the Engine must determine how that name is found with respect to namespace lookup. For this purpose, names can have one of the three following forms:

- *Unqualified name: Such names are just simple names without any prefix, as in the class name `Point` in the following expression: `$p = new Point(3,5)`. If the current namespace is `NS1`, the name `Point` resolves to `NS1\Point`. If the* current namespace is the default namespace (§§), the name `Point` resolves to `Point`. In the case of an unqualified function or constant name, if that name does not exist in the current namespace, a global function or constant by that name is used.

- Qualified name: Such names have a prefix consisting of a namespace name and/or one or more levels of sub-namespace names, and, possibly, a class, interface, trait, function, or constant name. Such names are relative. For example, `D2\Point` could be used to refer to the class Point in the sub-namespace `D2` of the current namespace. One special case of this is when the first component of the name is the keyword `namespace`. This means "the current namespace".
- Fully qualified name: Such names begin with a backslash (`\`) and are followed optionally by a namespace name and one or more levels of sub-namespace names, and, finally, a class, interface, trait, function, or constant name. Such names are absolute. For example, `\Graphics\D2\Point` could be used to refer unambiguously to the class `Point` in namespace `Graphics`, sub-namespace `D2`.

The names of the standard types (such as `Exception`), constants (such as `PHP_INT_MAX`), and library functions (such as `is_null`) are defined outside any namespace. To refer unambiguously to such names, one can prefix them with a backslash (`\`), as in `\Exception`, `\PHP_INT_MAX`, and `\is_null`.

Defining Namespaces

Syntax

```
namespace-definition:
  namespace  namespace-name  ;
  namespace  namespace-name_opt   compound-statement
```

namespace-name is defined in §§, and compound-statement is defined in §§.

Constraints

Except for white space and an optional declare-statement (§§), the first occurrence of a namespace-definition in a script must be the first thing in that script.

All occurrence of a namespace-definition in a script must have the compound-statement form or must not have that form; the two forms cannot be mixed.

When a script contains source code that is not inside a namespace, and source code that is inside one or namespaces, the namespaced code must use the compound-statement form of namespace-definition.

compound-statement must not contain a namespace-definition.

Semantics

Although a namespace may contain any PHP source code, the fact that that code is contained in a namespace affects only the declaration and name resolution of classes, interfaces, traits, functions, and constants.

Namespace and sub-namespace names are case-insensitive.

The pre-defined constant `__NAMESPACE__` (§§) contains the name of the current namespace.

When the same namespace is defined in multiple scripts, and those scripts are combined into the same program, the namespace is considered the merger of its individual contributions.

The scope of the non-compound-statement form of namespace-definition runs until the end of the script, or until the lexically next namespace-definition, whichever comes first. The scope of the compound-statement form is the compound-statement.

Examples

Script1.php:

```
namespace NS1;
...       // __NAMESPACE__ is "NS1"
namespace NS3\Sub1;
...       // __NAMESPACE__ is "NS3\Sub1"
```

Script2.php:

```
namespace NS1
{
...       // __NAMESPACE__ is "NS1"
}
namespace
{
...       // __NAMESPACE__ is ""
}
namespace NS3\Sub1;
{
...       // __NAMESPACE__ is "NS3\Sub1"
}
```

*Namespace Use Declarations***

Syntax

```
  namespace-use-declaration:
    use  namespace-use-clauses  ;

  namespace-use-clauses:
    namespace-use-clause
    namespace-use-clauses  ,  namespace-use-clause

  namespace-use-clause:
    qualified-name   namespace-aliasing-clause_opt

  namespace-aliasing-clause:
    as  name
```

qualified-name and name are defined in §§.

Constraints

A namespace-use-declaration must not occur except at the pseudomain level or directly in the context of a namespace-definition (18.3).

If the same qualified-name is imported multiple times in the same scope, each occurrence must have a different alias.

Semantics

qualified-name is always interpreted as referring to a class, interface, or trait by that name. namespace-use-clauses can only create aliases for classes, interfaces, or traits; it is not possible to use them to create aliases to functions or constants.

A namespace-use-declaration imports—that is, makes available—one or more names into a scope, optionally giving them each an alias. Each of those names may designate a namespace, a sub-namespace, a class, an interface, or a trait. If a namespace-alias-clause is present, its name is the alias for qualified-name. Otherwise, the right-most name in qualified-name is the implied alias for qualified-name.

Examples

```
namespace NS1
{
  const CON1 = 100;
  function f() { ... }
  class C { ... }
  interface I { ... }
  trait T { ... }
}

namespace NS2
{
  use \NS1\C, \NS1\I, \NS1\T;
  class D extends C implements I
  {
    use T;
  }
  $v = \NS1\CON1; // explicit namespace still needed for constants
  \NS1\f();   // explicit namespace still needed for functions

  use \NS1\C as C2; // C2 is an alias for the class name \NS1\C
  $c2 = new C2;
}
```

Grammar

General

The grammar notation is described in §§.

Lexical Grammar

General

```
  input-file::
    input-element
    input-file   input-element
  input-element::
    comment
    white-space
    token
```

Comments

```
  comment::
    single-line-comment
    delimited-comment

  single-line-comment::
    //   input-characters_opt
      #    input-characters_opt

  input-characters::
    input-character
```

```
    input-characters   input-character

  input-character::
    Any source character except new-line

  new-line::
    Carriage-return character (U+000D)
    Line-feed character (U+000A)
    Carriage-return character (U+000D) followed by line-feed character (U+000A)

  delimited-comment::
    /*   No characters or any source character sequence except /*   */
```

White Space

```
  white-space::
    white-space-character
    white-space   white-space-character

  white-space-character::
    new-line
    Space character (U+0020)
    Horizontal-tab character (U+0009)
```

Tokens

General

```
  token::
    variable-name
    name
    keyword
    literal
    operator-or-punctuator
```

Names

```
  variable-name::
    $   name

  namespace-name::
    name
    namespace-name   \   name

  namespace-name-as-a-prefix::
    \
    \opt   namespace-name   \
    namespace   \
    namespace   \   namespace-name   \

  qualified-name::
    namespace-name-as-a-prefixopt   name

  name::
    name-nondigit
    name   name-nondigit
    name   digit

  name-nondigit::
```

```
  nondigit
  one of the characters U+007f–U+00ff

nondigit:: one of
  _
  a   b   c   d   e   f   g   h   i   j   k   l   m
  n   o   p   q   r   s   t   u   v   w   x   y   z
  A   B   C   D   E   F   G   H   I   J   K   L   M
  N   O   P   Q   R   S   T   U   V   W   X   Y   Z
```

Keywords

```
keyword:: one of
  abstract   and   as   break   callable   case   catch   class   clone
  const   continue   declare   default   do   echo   else   elseif
  enddeclare   endfor   endforeach   endif   endswitch   endwhile
  extends   final   finally   for   foreach   function   global
  goto   if   implements   include   include_once   instanceof
  insteadof   interface   namespace   new or   print   private
  protected   public   require   require_once   return static   switch
  throw   trait   try   use   var   while   xor   yield
```

Literals

General

```
literal::
  boolean-literal
  integer-literal
  floating-literal
  string-literal
  null-literal
```

Boolean Literals

```
boolean-literal::
  TRUE (written in any case combination)
  FALSE (written in any case combination)
```

Integer Literals

```
integer-literal::
  decimal-literal
  octal-literal
  hexadecimal-literal
  binary-literal

  decimal-literal::
    nonzero-digit
    decimal-literal   digit

  octal-literal::
    0
    octal-literal   octal-digit

  hexadecimal-literal::
    hexadecimal-prefix   hexadecimal-digit
```

```
    hexadecimal-literal   hexadecimal-digit

  hexadecimal-prefix:: one of
    0x  0X

  binary-literal::
    binary-prefix   binary-digit
    binary-literal   binary-digit

  binary-prefix:: one of
    0b  0B

  digit:: one of
    0  1  2  3  4  5  6  7  8  9

  nonzero-digit:: one of
    1  2  3  4  5  6  7  8  9

  octal-digit:: one of
    0  1  2  3  4  5  6  7

  hexadecimal-digit:: one of
    0  1  2  3  4  5  6  7  8  9
          a  b  c  d  e  f
          A  B  C  D  E  F

  binary-digit:: one of
      0  1
```

Floating-Point Literals

```
floating-literal::
  fractional-literal   exponent-part_opt
  digit-sequence   exponent-part

fractional-literal::
  digit-sequence_opt . digit-sequence
  digit-sequence .

exponent-part::
  e  sign_opt   digit-sequence
  E  sign_opt   digit-sequence

sign:: one of
  +  -

digit-sequence::
  digit
  digit-sequence   digit
```

String Literals

```
string-literal::
  single-quoted-string-literal
  double-quoted-string-literal
  heredoc-string-literal
  nowdoc-string-literal

single-quoted-string-literal::
  b_opt  ' sq-char-sequence_opt  '

sq-char-sequence::
```

```
    sq-char
    sq-char-sequence   sq-char

  sq-char::
    sq-escape-sequence
    \opt   any member of the source character set except single-quote (') or backslash (\)

  sq-escape-sequence:: one of
    \'  \\

  double-quoted-string-literal::
    bopt  " dq-char-sequenceopt  "

  dq-char-sequence::
    dq-char
    dq-char-sequence   dq-char

  dq-char::
    dq-escape-sequence
    any member of the source character set except double-quote (") or backslash (\)
    \  any member of the source character set except "\$efnrtvxX or
octal-digit

  dq-escape-sequence::
    dq-simple-escape-sequence
    dq-octal-escape-sequence
    dq-hexadecimal-escape-sequence

  dq-simple-escape-sequence:: one of
    \"   \\   \$   \e   \f   \n   \r   \t   \v

  dq-octal-escape-sequence::
    \   octal-digit
    \   octal-digit   octal-digit
    \   octal-digit   octal-digit   octal-digit

  dq-hexadecimal-escape-sequence::
    \x  hexadecimal-digit   hexadecimal-digitopt
    \X  hexadecimal-digit   hexadecimal-digitopt

  heredoc-string-literal::
    <<<  hd-start-identifier   new-line   hd-char-sequenceopt  new-line hd-end-identifier  ;opt   new-line

  hd-start-identifier::
    name

  hd-end-identifier::
    name

  hd-char-sequence::
    hd-char
    hd-char-sequence   hd-char

  hd-char::
    hd-escape-sequence
    any member of the source character set except backslash (\)
    \  any member of the source character set except \$efnrtvxX or
octal-digit

  hd-escape-sequence::
    hd-simple-escape-sequence
    dq-octal-escape-sequence
    dq-hexadecimal-escape-sequence

  hd-simple-escape-sequence:: one of
    \\   \$   \e   \f   \n   \r   \t   \v
```

```
nowdoc-string-literal::
  <<<  '  hd-start-identifier  '  new-line  hd-char-sequence_opt   new-line hd-end-identifier  ;_opt   new-line
```

The Null Literal

```
null-literal::
  NULL (written in any case combination)
```

Operators and Punctuators

```
operator-or-punctuator:: one of
  [   ]    (   )   {    }   .   ->   ++   --   **   *   +   -   ~   !
  $   /   % <<   >>   <   >   <=   >=   ==   ===   !=   !==   ^   |
  &   &&   ||   ?   :   ; =   **=   *=   /=   %=   +=   -=   .=   <<=
  >>=   &=   ^=   |=   ,
```

Syntactic Grammar

Program Structure

```
script:
 script-section
 script   script-section

script-section:
   text_opt <?php statement-list_opt ?>_opt text_opt

text:
  arbitrary text not containing the sequence <?php
```

Variables

```
function-static-declaration:
  static name   function-static-initializer_opt ;

function-static-initializer:
  = const-expression

global-declaration:
  global variable-name-list ;

variable-name-list:
  expression
  variable-name-list  , expression
```

Expressions

Primary Expressions

```
primary-expression:
  variable-name
```

```
  qualified-name
  literal
  const-expression
  intrinsic
  anonymous-function-creation-expression
  (  expression  )
  $this

intrinsic:
  array-intrinsic
  echo-intrinsic
  empty-intrinsic
  eval-intrinsic
  exit-intrinsic
  isset-intrinsic
  list-intrinsic
  print-intrinsic
  unset-intrinsic

array-intrinsic:
  array ( array-initializer_opt  )

echo-intrinsic:
  echo  expression
  echo  (  expression  )
  echo  expression-list-two-or-more

expression-list-two-or-more:
  expression  ,  expression
  expression-list-two-or-more  ,  expression

empty-intrinsic:
  empty ( expression  )

eval-intrinsic:
  eval (  expression  )

exit-intrinsic:
  exit  expression_opt
  exit  (  expression_opt  )
  die   expression_opt
  die   (   expression_opt )

isset-intrinsic:
  isset  (  expression-list-one-or-more  )

expression-list-one-or-more:
  expression
  expression-list-one-or-mor  ,  expression

list-intrinsic:
  list  (  list-expression-list_opt  )

list-expression-list:
list-or-variable
,
list-expression-list  ,  list-or-variable_opt

list-or-variable:
  list-intrinsic
  expression

print-intrinsic:
  print  expression
  print  (  expression  )
```

```
unset-intrinsic:
  unset  (  expression-list-one-or-more  )

anonymous-function-creation-expression:
function  &opt (  parameter-declaration-listopt  )  anonymous-function-use-clauseopt
    compound-statement

anonymous-function-use-clause:
  use  (  use-variable-name-list  )

use-variable-name-list:
  &opt   variable-name
  use-variable-name-list  ,  &opt  variable-name
```

Postfix Operators

```
postfix-expression:
  primary-expression
  clone-expression
  object-creation-expression
  array-creation-expression
  subscript-expression
  function-call-expression
  member-selection-expression
  postfix-increment-expression
  postfix-decrement-expression
  scope-resolution-expression
  exponentiation-expression


clone-expression:
  clone  expression

object-creation-expression:
  new  class-type-designator  (  argument-expression-listopt  )
  new  class-type-designator

class-type-designator:
  static
  qualified-name
  expression

array-creation-expression:
  array  (  array-initializeropt  )
  [ array-initializeropt ]

array-initializer:
  array-initializer-list  ,opt

array-initializer-list:
  array-element-initializer
  array-element-initializer  ,  array-initializer-list

array-element-initializer:
  &opt   element-value
  element-key  =>  &opt   element-value

element-key:
  expression

element-value
  expression
```

```
subscript-expression:
  postfix-expression  [  expression_opt  ]
  postfix-expression  {  expression_opt  }   [Deprecated form]

function-call-expression:
  postfix-expression  (  argument-expression-list_opt  )

argument-expression-list:
  assignment-expression
  argument-expression-list  ,  assignment-expression

member-selection-expression:
  postfix-expression  ->  member-selection-designator

member-selection-designator:
  name
  expression

postfix-increment-expression:
  unary-expression  ++

postfix-decrement-expression:
  unary-expression  --

scope-resolution-expression:
  scope-resolution-qualifier  ::  member-selection-designator
  scope-resolution-qualifier  ::  class

scope-resolution-qualifier:
  qualified-name
  expression
  self
  parent
  static

exponentiation-expression:
  expression  **  expression
```

Unary Operators

```
unary-expression:
  postfix-expression
  prefix-increment-expression
  prefix-decrement-expression
  unary-op-expression
  error-control-expression
  shell-command-expression
  cast-expression
  variable-name-creation-expression

prefix-increment-expression:
  ++ unary-expression

prefix-decrement-expression:
  -- unary-expression

unary-op-expression:
  unary-operator cast-expression

unary-operator: one of
  +  -  !  \

error-control-expression:
  @   expression
```

```
shell-command-expression:
  `  dq-char-sequence_opt  `

cast-expression:
  unary-expression
  (  cast-type  ) cast-expression

cast-type: one of
  array  binary  bool  boolean  double  int  integer  float  object
  real  string  unset

variable-name-creation-expression:
  $   expression
  $  {  expression  }
```

instanceof Operator

```
instanceof-expression:
  unary-expression
  instanceof-subject  instanceof   instanceof-type-designator

instanceof-subject:
  expression

instanceof-type-designator:
  qualified-name
  expression
```

Multiplicative Operators

```
multiplicative-expression:
  instanceof-expression
  multiplicative-expression  *  multiplicative-expression
  multiplicative-expression  /  multiplicative-expression
  multiplicative-expression  %  multiplicative-expression
```

Additive Operators

```
additive-expression:
  multiplicative-expression
  additive-expression  +  multiplicative-expression
  additive-expression  -  multiplicative-expression
  additive-expression  .  multiplicative-expression
```

Bitwise Shift Operators

```
shift-expression:
  additive-expression
  shift-expression  <<  additive-expression
  shift-expression  >>  additive-expression
```

Relational Operators

```
relational-expression:
  shift-expression
```

```
  relational-expression  <   shift-expression
  relational-expression  >   shift-expression
  relational-expression  <=  shift-expression
  relational-expression  >=  shift-expression
```

Equality Operators

```
equality-expression:
  relational-expression
  equality-expression  ==  relational-expression
  equality-expression  !=  relational-expression
  equality-expression  <>  relational-expression
  equality-expression  ===  relational-expression
  equality-expression  !==  relational-expression
```

Bitwise Logical Operators

```
bitwise-AND-expression:
  equality-expression
  bit-wise-AND-expression  &  equality-expression

bitwise-exc-OR-expression:
  bitwise-AND-expression
  bitwise-exc-OR-expression  ^   bitwise-AND-expression

bitwise-inc-OR-expression:
  bitwise-exc-OR-expression
  bitwise-inc-OR-expression  |  bitwise-exc-OR-expression
```

Logical Operators (form 1)

```
logical-AND-expression-1:
  bitwise-incl-OR-expression
  logical-AND-expression-1  &&  bitwise-inc-OR-expression

logical-inc-OR-expression-1:
  logical-AND-expression-1
  logical-inc-OR-expression-1  ||  logical-AND-expression-1
```

Conditional Operator

```
conditional-expression:
  logical-inc-OR-expression-1
  logical-inc-OR-expression-1  ?  expression_opt  :  conditional-expression
```

Assignment Operators

```
assignment-expression:
  conditional-expression
  simple-assignment-expression
  byref-assignment-expression
  compound-assignment-expression

simple-assignment-expression:
  unary-expression  =  assignment-expression

byref-assignment-expression:
```

```
  unary-expression  =  &  assignment-expression

compound-assignment-expression:
  unary-expression   compound-assignment-operator   assignment-expression

compound-assignment-operator: one of
  **=  *=  /=  %=  +=  -=  .=  <<=  >>=  &=  ^=  |=
```

Logical Operators (form 2)

```
logical-AND-expression-2:
  assignment-expression
  logical-AND-expression-2  and  assignment-expression

logical-exc-OR-expression:
  logical-AND-expression-2
  logical-exc-OR-expression  xor  logical-AND-expression-2

logical-inc-OR-expression-2:
  logical-exc-OR-expression
  logical-inc-OR-expression-2  or  logical-exc-OR-expression
```

yield Operator

```
yield-expression:
  logical-inc-OR-expression-2
  yield  array-element-initializer
```

Script Inclusion Operators

```
expression:
  yield-expression
  include-expression
  include-once-expression
  require-expression
  require-once-expression

include-expression:
  include  (  include-filename  )
  include  include-filename

include-filename:
  expression

include-once-expression:
  include_once  (  include-filename  )
  include_once  include-filename

require-expression:
  require  (  include-filename  )
  require  include-filename

require-once-expression:
  require_once  (  include-filename  )
  require_once  include-filename
```

Constant Expressions

```
constant-expression:
  array-creation-expression
  const-expression

const-expression:
  expression
```

Statements

General

```
statement:
  compound-statement
  labeled-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  declare-statement
  const-declaration
  function-definition
  class-declaration
  interface-declaration
  trait-declaration
  namespace-definition
  namespace-use-declaration
  global-declaration
  function-static-declaration
```

Compound Statements

```
compound-statement:
  {   statement-list_opt  }

statement-list:
  statement
  statement-list   statement
```

Labeled Statements

```
labeled-statement:
  named-label
  case-label
  default-label

named-label:
  name  :  statement

case-label:
  case   expression   case-default-label-terminator   statement

default-label:
  default  case-default-label-terminator   statement

case-default-label-terminator:
  :
  ;
```

Expression Statements

```
 expression-statement:
   expression_opt  ;

selection-statement:
  if-statement
  switch-statement

if-statement:
  if   (   expression   )   statement   elseif-clauses-1opt   else-clause-1opt
  if   (   expression   )   :   statement-list   elseif-clauses-2opt   else-clause-2opt   endif   ;

elseif-clauses-1:
  elseif-clause-1
  elseif-clauses-1   elseif-clause-1

elseif-clause-1:
  elseif   (   expression   )   statement

else-clause-1:
  else   statement

elseif-clauses-2:
  elseif-clause-2
  elseif-clauses-2   elseif-clause-2

elseif-clause-2:
  elseif   (   expression   )   :   statement-list

else-clause-2:
  else   :   statement-list

switch-statement:
  switch  (  expression  )  compound-statement
  switch  (  expression  )  :   statement-list  endswitch;
```

Iteration Statements

```
iteration-statement:
  while-statement
  do-statement
  for-statement
  foreach-statement

while-statement:
  while  (  expression  )  statement
  while  (  expression  )  :   statement-list  endwhile ;

do-statement:
  do  statement  while  (  expression  )  ;


for-statement:
  for   (   for-initializeropt   ;   for-controlopt   ;   for-end-of-loopopt   )   statement
  for   (   for-initializeropt   ;   for-controlopt   ;   for-end-of-loopopt   )   :   statement-list   endfor   ;

for-initializer:
  for-expression-group

for-control:
  for-expression-group
```

```
for-end-of-loop:
  for-expression-group

for-expression-group:
  expression
  for-expression-group   ,   expression

foreach-statement:
  foreach  (  foreach-collection-name  as  foreach-key_opt  foreach-value  )   statement
  foreach  (  foreach-collection-name  as  foreach-key_opt   foreach-value  )  :   statement-list  endforeach  ;

foreach-collection-name:
  expression

foreach-key:
  expression  =>

foreach-value:
  &_opt   expression
  list-intrinsic
```

Jump Statements

```
jump-statement:
  goto-statement
  continue-statement
  break-statement
  return-statement
  throw-statement

goto-statement:
  goto  name  ;

continue-statement:
  continue   breakout-level_opt  ;

breakout-level:
  integer-literal

break-statement:
  break  breakout-level_opt  ;

return-statement:
  return  expression_opt  ;

throw-statement:
  throw  expression  ;
```

The try Statement

```
try-statement:
  try  compound-statement   catch-clauses
  try  compound-statement   finally-clause
  try  compound-statement   catch-clauses   finally-clause

catch-clauses:
  catch-clause
  catch-clauses   catch-clause
```

```
catch-clause:
  catch  (  parameter-declaration-list  )  compound-statement

finally-clause:
  finally   compound-statement
```

The declare Statement

```
declare-statement:
  declare  (  declare-directive  )  statement
  declare  (  declare-directive  )  :  statement-list  enddeclare  ;
  declare  (  declare-directive  )  ;

declare-directive:
  ticks  =  declare-tick-count
  encoding  =  declare-character-encoding

declare-tick-count
  expression

declare-character-encoding:
  expression
```

Functions

```
function-definition:
  function-definition-header   compound-statement

function-definition-header:
  function  &opt   name  (  parameter-declaration-listopt  )

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list  ,  parameter-declaration

parameter-declaration:
  type-hintopt  &opt   variable-name   default-argument-specifieropt

type-hint:
  array
  callable
  qualified-name

default-argument-specifier:
  =  const-expression
```

Classes

```
class-declaration:
  class-modifieropt  class  name   class-base   clauseopt  class-interface-clauseopt   {   trait-use-clausesopt   clas

class-modifier:
  abstract
  final

class-base-clause:
  extends  qualified-name

class-interface-clause:
```

```
    implements  qualified-name
    class-interface-clause  ,  qualified-name

  class-member-declarations:
    class-member-declaration
    class-member-declarations   class-member-declaration

   class-member-declaration:
     const-declaration
     property-declaration
     method-declaration
     constructor-declaration
     destructor-declaration

  const-declaration:
    const  name  =  const-expression   ;

  property-declaration:
    property-modifier   name   property-initializer_opt  ;

  property-modifier:
    var
    visibility-modifier   static-modifier_opt
    static-modifier   visibility-modifier_opt

  visibility-modifier:
    public
    protected
    private

  static-modifier:
    static

  property-initializer:
    =  constant-expression

  method-declaration:
    method-modifiers_opt   function-definition
    method-modifiers   function-definition-header  ;

  method-modifiers:
    method-modifier
    method-modifiers   method-modifier

  method-modifier:
    visibility-modifier
    static-modifier
    abstract
    final

  constructor-definition:
    visibility-modifier  function &_opt   __construct  (  parameter-declaration-list_opt  )  compound-statement
    visibility-modifier  function &_opt    name  (  parameter-declaration-list_opt  )  compound-statement     [Deprecate

  destructor-definition:
    visibility-modifier  function  &_opt  __destruct  ( ) compound-statement
```

Interfaces

```
  interface-declaration:
    interface   name   interface-base-clause_opt {  interface-member-declarations_opt  }
```

```
interface-base-clause:
  extends   qualified-name
  interface-base-clause  ,  qualified-name

interface-member-declarations:
  interface-member-declaration
  interface-member-declarations   interface-member-declaration

interface-member-declaration:
  const-declaration
  method-declaration
```

Traits

```
trait-declaration:
  trait   name   {   trait-use-clauses_opt   trait-member-declarations_opt   }

trait-use-clauses:
  trait-use-clause
  trait-use-clauses   trait-use-clause

trait-use-clause:
  use   trait-name-list   trait-use-terminator

trait-name-list:
  qualified-name
  trait-name-list   ,   qualified-name

trait-use-terminator:
  ;
  {   trait-select-and-alias-clauses_opt   }

trait-select-and-alias-clauses:
  trait-select-and-alias-clause
  trait-select-and-alias-clauses   trait-select-and-alias-clause

trait-select-and-alias-clause:
  trait-select-insteadof-clause
  trait-alias-as-clause

trait-select-insteadof-clause:
  name   insteadof   name

trait-alias-as-clause:
  name   as   visibility-modifier_opt   name
  name   as   visibility-modifier   name_opt

trait-member-declarations:
  trait-member-declaration
  trait-member-declarations   trait-member-declaration

trait-member-declaration:
  property-declaration
  method-declaration
  constructor-declaration
  destructor-declaration
```

Namespaces

```
namespace-definition:
```

```
    namespace  namespace-name  ;
    namespace  namespace-name_opt   compound-statement

  namespace-use-declaration:
    use  namespace-use-clauses  ;

  namespace-use-clauses:
    namespace-use-clause
    namespace-use-clauses  ,  namespace-use-clause

  namespace-use-clause:
    qualified-name   namespace-aliasing-clause_opt

  namespace-aliasing-clause:
    as  name
```

Bibliography

The following documents are useful references for implementers and users of this specification:

IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems (previously designated IEC 559:1989). (This standard is widely known by its U.S. national designation, ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic).

The Unicode Consortium. The Unicode Standard, Version 5.0, www.Unicode.org).