TECHNICAL GUIDE

Writing PHP Extensions

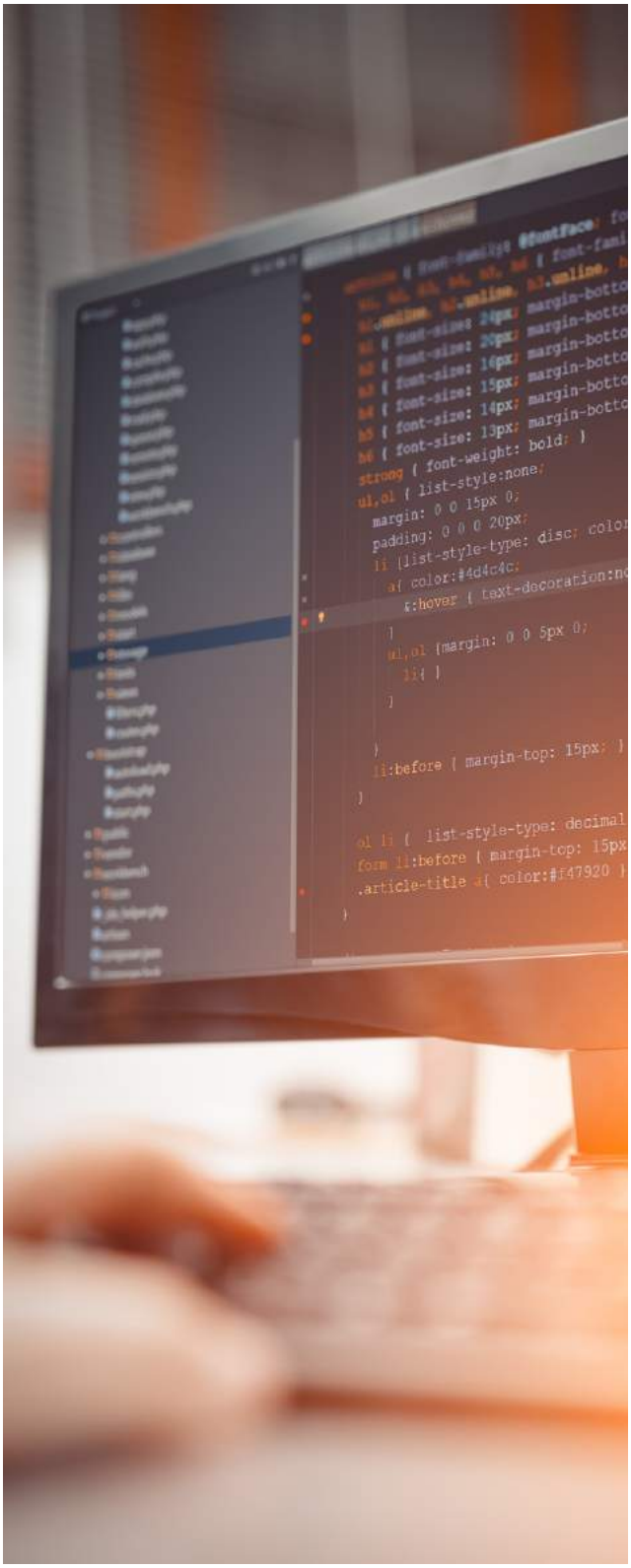
Introduction

Knowing how to use and write PHP extensions is a critical PHP development skill that can save significant time and enable you to quickly add new features to your apps. For example, today, there are more than 150 extensions from the PHP community that provide ready-to-go compiled libraries that enable functions. By using them in your apps, you can avoid developing them yourself.

Despite the large number of existing PHP extensions, you may need to write your own. To help you do that, this document describes how to:

- Setup a Linux PHP build environment.
- Generate an extension skeleton.
- Build and install a PHP extension.
- Rebuild extensions for production.
- Understand extension skeleton file content.
- Run extension tests.
- Add new functionality (functions, callbacks, constants, global variables, and configuration directives).
- Use basic PHP structures, including the API.
- Use PHP arrays.
- Catch memory leaks.
- Manage memory.
- Use PHP references.
- Use copy on write.
- Use PHP classes and objects.
- Use object-oriented programming (OOP) in an extension.
- Embed C data in PHP objects.
- Override object handlers.
- Avoid common issues with external library linking, naming conventions, and PHP resource type.

To help you learn from all the coding examples in this document, please visit the GIT repository, https://github.com/dstogov/php-extension. It includes a copy of all the files generated when creating the sample extension described in this book. The extension modifications are reflected by separate GIT commits.



Setting up Your PHP Build Environment on Linux

PHP extensions are written in C, and when we develop extensions, we need to care about memory management, array boundaries, and many other low-level problems. Consequently, it's almost impossible to develop extension from scratch without bugs, and therefore we'll have to debug them. This is the reason I highly recommend that you create a "DEBUG" PHP build when you setup your PHP build environment on Linux. It will help to detect common errors much earlier.

Building PHP from Linux-based sources is not too complicated. However, you first need to install the necessary development components, which include a C compiler, linker, libraries, and include files. Use your Linux package manager to do this.

For Ubuntu/Debian:

```
$ sudo apt-get install build-essential autoconf automake bison flex re2c gdb \
      libtool make pkgconf valgrind git libxml2-dev libsqlite3-dev
```

For RedHat/Fedora:

```
$ sudo dnf install gcc gcc-c++ binutils glibc-devel autoconf automake bison \
      flex re2c gdb libtool make pkgconf valgrind git \
      libxml2-devel libsqlite3x-devel
```

Now, you can clone the PHP GIT repository from github.com and switch to the sources of the necessary PHP version. (Without the last command, you are going to work with a "master" branch or ongoing new PHP 8.)

```
$ git clone https://github.com/php/php-src.git
$ cd php-src
$ git checkout php-7.4.1  (switch to tag/branch of necessary PHP version)
```

Next, configure PHP. We are going to build "DEBUG" PHP, install it inside our home directory, and use a custom php.ini file. The "./configure" command may be extended with additional options, depending on your PHP build requirement. You can specify:

- Which SAPI (CLI, FastCGI, FPM, Apache) you are going to use.
- Enable or disable embedded PHP extensions and their options.

The full list of possible configuration options is available at "./configure –help."

```
$ ./buildconf --force
$ ./configure --enable-debug \
      --prefix=$HOME/php-bin/DEBUG \
      --with-config-file-path=$HOME/php-bin/DEBUG/etc
```

Usually, you will need to build PHP in a way that's similar to your existing binary build. To save time, you can retrieve the configuration options you use for existing builds with the "php -i | grep 'Configure Command'" and adding it to our "./configure" command. Note that building some PHP extensions may require installation of additional libraries and headers. All the package dependencies are usually checked during this step.

Usually, you will need to build PHP in a way that's similar to your existing binary build. To save time, you can retrieve the configuration options you use for existing builds with the "php -i | grep 'Configure Command'" and adding it to our "./configure" command. Note that building some PHP extensions may require installation of additional libraries and headers. All the package dependencies are usually checked during this step.

Finally, when configure succeeds, we compile and install our PHP build:

```
$ make -j4
$ make install
$ cd ..
```

Now we need to create our custom php.ini:

```
$ mkdir ~/php-bin/DEBUG/etc
$ vi ~/php-bin/DEBUG/etc/php.ini
```

It should contain something like the following to enable error reporting and catch possible bugs early:

```
date.timezone=GMT
max_execution_time=30
memory_limit=128M

error_reporting=E_ALL | E_STRICT ; catch all error and warnings
display_errors=1
log_errors=1

zend_extension=opcache.so
opcache.enable=1
opcache.enable_cli=1
opcache.protect_memory=1         ; catch invalid updates of shared memory
```

It makes sense to include your PHP binaries into PATH to override the PHP system:

```
$ export PATH=~/php-bin/DEBUG/bin:$PATH
```

Now we can check that everything works fine:

```
$ php -v
```

You should get something like this:

```
PHP 7.4.1 (cli) (built: Jan 15 2020 12:52:43) ( NTS DEBUG )
Copyright (c) The PHP Group
Zend Engine v3.4.0, Copyright (c) Zend Technologies
      with Zend OPcache v7.4.1, Copyright (c), by Zend Technologies
```

Our "DEBUG" PHP build is ready to start development.

Generating a PHP Extension Skeleton

Writing a basic PHP extension is not too difficult. You only need to create a few files. You can do this manually, but you may also use the "ext_skel.php" script:

```
$ php php-src/ext/ext_skel.php --ext test --dir .
```

Unfortunately, this script is not distributed with binary PHP builds and is only available in source distribution. This will create directory "test" with extension skeleton files. Let's look inside:

```
$ cd test$ ls
config.m4  config.w32  php_test.h  test.c  tests
```

In the above code snippet:

- **config.m4** is an extension configuration script used by "phpize" or "buildconf" to add extension configuration options into the "configure" command.
- **config.w32** is a similar configuration file for the Windows build system, which is discussed later in this blog.
- **php_test.h** is a C header file that contains our common extension definitions. It's not necessary for simple extensions with a single-source C file, but it's useful in case the implementation is spread among few files.
- **test.c** is the main extension implementation source. It defines all the structures that allow to plug the extension into PHP and make all their internal functions, classes and constants to be available.
- **tests** refers to the directory with PHP tests. We will review them later.

Building and Installing a PHP Extension

This extension skeleton can be compiled without any changes. The first "phpize" command is a part of the PHP build we created in the first step. (It should still be in the PATH.)

```
$ phpize
$ ./configure
$ make
$ make install
```

These commands should build our shared extension "test.so" and copy it into appropriate directory of our PHP installation. To load it, we need to add a line into our custom php.ini

```
$ vi ~/php-bin/DEBUG/etc/php.ini
```

Add the following line:

```
extension=test.so
```

Check that extension is loaded and works. "php -m" command prints the list of loaded extensions:

```
$ php -m | grep test
test
```

We may also run the functions defined in our "test" extension:

```
$ php -r 'test_test1();'
The extension test is loaded and working!
$ php -r 'echo test_test2("world\n");'
Hello world
```

Now it makes sense to start tracking our source changes using version control system. (I prefer GIT.)

```
$ git init
$ git add config.m4 config.w32 test.c php_test.h tests
$ git commit -m "Initial Extension Skeleton"
```

Rebuilding Extensions for Production

While we are on the subject of build processes and before we go deeper into building a PHP extension skeleton, it makes sense to explain how to rebuild the extension for production when it's ready. Actually, you may try this right now. At first, you'll need PHP development tools especially for your PHP build. It may be a system package.

Installation for Ubuntu/Debian:

```
$ sudo apt-get install php-dev
```

Installation for RedHat/Fedora:

```
$ sudo dnf install php-devel
```

For Zend Server you should get "php-dev-zend-server" package installed (through Zend Server installer or system package manager) and use its components in the PATH.

```
$ export PATH=/usr/local/zend/bin:$PATH
```

The building is very similar to the "DEBUG" build. The difference is that now we use "phpize" from "production" build. (PATH shouldn't include our DEBUG PHP build directory.)

```
$ phpize
$ ./configure
$ make
$ sudo make install
```

Add extension into php.ini:

```
$ vi /etc/php.ini
```

Add the following line:

```
extension=test.so
```

Check that extension is loaded and works:

```
$ php -m | grep test
test
```

Now you can restart your web server or PHP-FPM and start using your extension in a web environment.

Extension Skeleton File Content

Let's review the contents of extension skeleton files.

"config.m4" is an extension configuration script, used during generation of "configure" script by "phpize" or "buildconf" commands. It's written in M4 macro-processing language. Very basic knowledge is enough for PHP extension configuration. You can copy-paste blocks from this tutorial or other extension configuration files.

```
PHP_ARG_ENABLE([test],
      [whether to enable test support],
      [AS_HELP_STRING([--enable-test],
          [Enable test support])],
      [no])

if test "$PHP_TEST" != "no"; then
      AC_DEFINE(HAVE_TEST, 1, [ Have test support ])
      PHP_NEW_EXTENSION(test, test.c, $ext_shared)
fi
```

PHP_ARG_ENABLE(...) – macro adds a configuration option "--enable-test". It may get three values "yes", "no", and "shared".

When you run "phpize", the default value is "shared" which means we are going to build a dynamically loadable PHP extension. However, it is possible to copy the "test" extension directory into the main PHP distribution ("ext/test") and re-run "./buildconf" and "./configure ... –enable-test" to re-build the whole PHP with extension "test", statically linked in.

It's possible to enable extension by default, replacing "no" to "yes" at line 5. In this case, it's possible to disable "test" extension by "./configure --disable-test".

Following "if" is just a regular UNIX shell code that tests the value defined by "--enable-test", "--disable-test", or "--enable-test=shared".

AC_DEFINE(HAVE_TEST) adds C macro HAVE_TEST into "config.h", so you can use conditional compilation directives (#ifdef, #ifndef) to skip useless code, if necessary.

Finally, the PHP_NEW_EXTENSION (test, test.c, $ext_shared) macro states that we are going to build extension "test" from "test.c" file. It's possible to specify few files. Depending on the value of $ext_shared variable, the extension could be built as shared object or linked statically. (It's taken from the same "--enable-test" option.)

This file might need to be extended in case you add new source files or need to link some external libraries. I'll show how to link libraries later. Just don't forget to rerun "phpize"/"buildconf" + "configure" after you make any changes in this file.

Windows PHP uses a different build system. For Windows, file "config.w32" is a replacement of "config.m4". The two are almost the same. They use similar macros, just a different language: on Windows PHP build system uses JavaScript instead of M4 and Shell. I won't repeat the explanation of the macros. You should be able to guess.

```
ARG_ENABLE('test', 'test support', 'no');

if (PHP_TEST != 'no') {
      AC_DEFINE('HAVE_TEST', 1, 'test support enabled');
      EXTENSION('test', 'test.c', null, '/DZEND_ENABLE_STATIC_TSRMLS_CACHE=1');
}
```

"php_test.h" is a C header file with common definitions. In our very basic case, it defines:

- test_module_entry — an extension description structure. (It's an entry point to the extension.)
- PHP_TEST_VERSION — a version of the extension.
- ZEND_TSRMLS_CACHE_EXTERN — a thread-local storage cache entry, if the extension was built for a thread-safe build (ZTS) and compiled as shared object (COMPILE_DL_TEST).

```
/* test extension for PHP */

#ifndef PHP_TEST_H
# define PHP_TEST_H

extern zend_module_entry test_module_entry;
# define phpext_test_ptr &test_module_entry

# define PHP_TEST_VERSION "0.1.0"

# if defined(ZTS) && defined(COMPILE_DL_TEST)
ZEND_TSRMLS_CACHE_EXTERN()
# endif

#endif    /* PHP_TEST_H */
```

"test.c" is the main (and in our case, single) extension source file. It's too big to fit into one page/screen, so I'll split it into small parts and explain each part separately.

```
/* test extension for PHP */

#ifdef HAVE_CONFIG_H
# include “config.h”
#endif

#include “php.h”
#include “ext/standard/info.h”
#include “php_test.h”
```

Include necessary C header files. You may add additional “#include” directives if necessary.

```
/* For compatibility with older PHP versions */
#ifndef ZEND_PARSE_PARAMETERS_NONE
#define ZEND_PARSE_PARAMETERS_NONE() \
      ZEND_PARSCE_PARAMETERS_START(0, 0) \
      ZEND_PARSE_PARAMETERS_END()
#endif
```

Some forward compatibility macro, to make it possible to compile the extension for older PHP-7 versions.

```
/* {{{ void test_test1()
 */
PHP_FUNCTION(test_test1)
{
      ZEND_PARSE_PARAMETERS_NONE();

      php_printf(“The extension %s is loaded and working!\r\n”, “test”);
}
/* }}} */
```

A C code for function test_test1() provided by our PHP extension. The argument of PHP_FUNCTION() macro is the function name. ZEND_PARSE_PARAMETERS_NONE() tells that this function doesn’t require any arguments. php_printf(…) is just a C function call that prints the string into the output stream, similar to PHP printf() function.

```
/* {{{ string test_test2( [ string $var ] )
 */
PHP_FUNCTION(test_test2)
{
      char *var = "World";
      size_t var_len = sizeof("World") - 1;
      zend_string *retval;

      ZEND_PARSE_PARAMETERS_START(0, 1)
          Z_PARAM_OPTIONAL
          Z_PARAM_STRING(var, var_len)
      ZEND_PARSE_PARAMETERS_END();

      retval = strpprintf(0, "Hello %s", var);

      RETURN_STR(retval);
}
/* }}}*/
```

Another, more complex function uses "Fast Parameter Parsing API" to describe its arguments.

ZEND_PARSE_PARAMETERS_START(0, 1) starts the parameter description section. Its first argument (0) defines the number of required arguments.

The second argument (1) defines the maximum number of arguments. So, our function may be called without arguments, or with a single argument.

Inside this section, we should define all parameters, their types, and where they will be copied. For our case:

- Z_PARAM_OPTIONAL separates required parameters from optional ones.
- Z_PARAM_STRING() defines a string parameter that value is going to be copied to variable "var" and the length into variable "var_len."

Note that our argument is optional and therefore may be omitted. In this case a default value "World" is used. See initializers for variables "var" and "var_len" above ZEND_PARSE_PARAMETERS_START.

The code creates a "zend_string" value and returns it though macro RETURN_STR() similar to PHP sprintf() function:

```
/* {{{ PHP_RINIT_FUNCTION
 */
PHP_RINIT_FUNCTION(test)
{
#if defined(ZTS) && defined(COMPILE_DL_TEST)
      ZEND_TSRMLS_CACHE_UPDATE();
#endif

      return SUCCESS;
}
/* }}} */
```

PHP_RINIT_FUNCTION() defines a callback function that is going to be called at each request start-up. In our case, it only initializes thread-local storage cache. It would be much better to do this early (in MINIT or GINIT callbacks). I predict this will be fixed in the PHP 8 extension skeleton.

```
/* {{{ PHP_MINFO_FUNCTION
 */
PHP_MINFO_FUNCTION(test)
{
      php_info_print_table_start();
      php_info_print_table_header(2, “test support”, “enabled”);
      php_info_print_table_end();
}
/* }}} */
```

PHP_MINFO_FUNCTION() defines a callback function that is going to be called from PHP phpinfo() function, to print information about the extension.

```
/* {{{ arginfo
 */
ZEND_BEGIN_ARG_INFO(arginfo_test_test1, 0)
ZEND_END_ARG_INFO()
```

Information about arguments of the first function. There are no arguments.

```
ZEND_BEGIN_ARG_INFO(arginfo_test_test2, 0)
      ZEND_ARG_INFO(0, str)
ZEND_END_ARG_INFO()
/* }}} */
```

Information about arguments of the second function. The single optional argument with name “str” is passed by value.

```
/* {{{ test_functions[]
 */
static const zend_function_entry test_functions[] = {
      PHP_FE(test_test1,            arginfo_test_test1)
      PHP_FE(test_test2,            arginfo_test_test2)
      PHP_FE_END
};
/* }}} */
```

"test_functions" is a list of all extension functions with information about their arguments. The list is terminated by PHP_FE_END macro./*

```
/* {{{ test_module_entry
*/
zend_module_entry test_module_entry = {
      STANDARD_MODULE_HEADER,
      "test",                       /* Extension name */
      test_functions,               /* zend_function_entry */
      NULL,                         /* PHP_MINIT - Module initialization */
      NULL,                         /* PHP_MSHUTDOWN - Module shutdown */
      PHP_RINIT(test),              /* PHP_RINIT - Request initialization */
      NULL,                         /* PHP_RSHUTDOWN - Request shutdown */
      PHP_MINFO(test),              /* PHP_MINFO - Module info */
      PHP_TEST_VERSION,             /* Version */
      STANDARD_MODULE_PROPERTIES
};
/* }}} */
```

test_module_entry is the main extension entry structure. PHP core takes all information about extensions from such structures. It defines:

- Extension name ("test").
- A list of declared PHP functions ("test_functions").
- A few callback functions and extension version (PHP_TEST_VERSION - defined in the header file).

The callbacks occur when PHP started (MINIT), on PHP termination (MSHUTDOWN), at start of each request processing (RINIT), at the end of each request processing (RSHUTDOWN) and from phpinfo() (MINFO).

```
#ifdef COMPILE_DL_TEST
# ifdef ZTS
ZEND_TSRMLS_CACHE_DEFINE()
# endif
ZEND_GET_MODULE(test)
#endif
```

Finally, a couple of definitions for dynamic linking.

Running PHP Extension Tests

In addition to these four files, "ext_skel" script created a "tests" directory, with several *.phpt files inside it. These are automated tests that may be executed all together, by running the "make test" command:

```
$ make test
…
=====================================================================
TEST RESULT SUMMARY
---------------------------------------------------------------------
Exts skipped    :    0
Exts tested     :   26
---------------------------------------------------------------------

Number of tests :    3                 3
Tests skipped   :    0 (  0.0%) --------
Tests warned    :    0 (  0.0%) (  0.0%)
Tests failed    :    0 (  0.0%) (  0.0%)
Tests passed    :    3 (100.0%) (100.0%)
---------------------------------------------------------------------
Time taken      :    1 seconds
=====================================================================
```

The command will print the test execution progress and result summary. It's a good practice to cover most extension logic with corresponding tests and always run tests after changes.

Let's investigate the test file "tests/003.phpt."

```
--TEST--
test_test2() Basic test
--SKIPIF--
<?php
if (!extension_loaded('test')) {
      echo 'skip';
}
?>
--FILE--
<?php
var_dump(test_test2());
var_dump(test_test2('PHP'));
?>
--EXPECT--
string(11) "Hello World"
string(9) "Hello PHP"
```

The file combines a few sections:

- "--TEST--" defines the test name.
- "--SKIPIF--" (optional) contains a PHP code to check skip conditions. If it prints some string, started from word "skip", the whole test is going to be skipped. Our section prints "skip" if extension "test" is not loaded because it doesn't make sense to check functions that are not loaded (the test would fail).
- "--FILE--" section contains the main test PHP code.
- "--EXPECT--" contains the expected output of the test script.

Adding New Functionality

At this point you should already know the basic structure of a PHP extension and the building process. Now, we are going to learn how to implement new basic PHP extension features. Starting from this section, I'll write new code in red and keep existing code black.

FUNCTIONS

Functions are the simplest primitives to add new functionality. To implement a new function, we first have to write the function code itself. This is regular C code that starts with the PHP_FUNCTION() macro and name of the function. This code requires a single double number argument and it returns an argument that is scaled by factor 2. I will describe parameter parsing API and most macros for manipulating values later.

```
PHP_FUNCTION(test_scale)
{
      double x;

      ZEND_PARSE_PARAMETERS_START(1, 1)
          Z_PARAM_DOUBLE(x)
      ZEND_PARSE_PARAMETERS_END();

      RETURN_DOUBLE(x * 2);
}
```

We also have to define the arguments' description block. This block is started from ZEND_BEGIN_ARG_INFO() macro (or its variant) and terminated by ZEND_END_ARG_INFO() macro.

The first argument of ZEND_BEGIN_ARG_INFO() is the name of the arg_info structure. The same name should be reused in PHP_FE() macro.

The second argument is ignored. (In PHP 5, it meant pass rest of arguments by reference.) Each argument is defined by the ZEND_ARG_INFO() macro that takes the "pass by reference" value and the argument name.

```
ZEND_BEGIN_ARG_INFO(arginfo_test_scale, 0)
      ZEND_ARG_INFO(0, x)
ZEND_END_ARG_INFO()
```

It's possible to use extended variants of ARG_INFO macros to specify additional arguments and return type hints, null-ability, returning by reference, number of required arguments, functions with variable number of arguments, etc.

Finally, we have to add our new function into the list of extension functions:

```
static const zend_function_entry test_functions[] = {
      PHP_FE(test_test1,            arginfo_test_test1)
      PHP_FE(test_test2,            arginfo_test_test2)
      PHP_FE(test_scale,            arginfo_test_scale)
      PHP_FE_END
};
```

After extension rebuild and installation, the new function should start working.

```
$ php -r 'var_dump(test_scale(5));'
float(10)
```

To declare functions inside a namespace, it is possible to use ZEND_NS_FUNCTION(ns, name) instead of PHP_FUNCTION(name) — and ZEND_NS_FE(ns, name, arg_info) instead of PHP_FE(name, arg_info).

It's also possible to add some function flags (e.g. deprecate function adding ZEND_ACC_DEPRECATED flag), using ZEND_FENTRY() instead if PHP_FE(). See Zend/zend_API.h for the main extension API.

EXTENSION CALLBACKS

Only PHP extension functions may be implemented in a pure declarative manner. All other extension features may be implemented calling special API functions during PHP start-up. To do this, the extension should implement MINIT() callback. This is, again, a regular C function, starting with PHP_MINIT_FUNCTION() macro and the extension name as an argument. The function should return SUCCESS to link the PHP extension into the core and enable all its functions and other features. Our MINIT function just initializes thread-local storage cache. Previously, this code was called from RINIT, but in case you execute something in MINIT, and directly or indirectly access module global variables or common global variables, it's better to move this code into the beginning of MINIT.

```
PHP_MINIT_FUNCTION(test)
{
#if defined(ZTS) && defined(COMPILE_DL_TEST)
      ZEND_TSRMLS_CACHE_UPDATE();
#endif

      return SUCCESS;
}
```

The MINIT callback address should be added into the module entry structure. You can also remove RINIT callback if it was used only for thread-local storage and is now empty.

```
zend_module_entry test_module_entry = {
      STANDARD_MODULE_HEADER,
      "test",                       /* Extension name */
      test_functions,               /* zend_function_entry */
      PHP_MINIT(test),              /* PHP_MINIT - Module initialization */
      NULL,                         /* PHP_MSHUTDOWN - Module shutdown */
      NULL,                         /* PHP_RINIT - Request initialization */
      NULL,                         /* PHP_RSHUTDOWN - Request shutdown */
      PHP_MINFO(test),              /* PHP_MINFO - Module info */
      PHP_TEST_VERSION,             /* Version */
      STANDARD_MODULE_PROPERTIES
};
```

Similar to MINIT, you may implement other callbacks like MSHUTDOWN, when you need to free up some resources before PHP termination.

CONSTANTS

MINIT callback is suitable to add various extension entities like a new internal constant. This is done using REGISTER_LONG_CONSTANT() macro, where the first argument is the constant name, the second is the constant value, and the third is the constant flags:

- CONST_CS refers to the case sensitive constant name.
- CONST_PERSISTENT refers to the persistent constant. (All internal extension constants should be persistent.)

```
PHP_MINIT_FUNCTION(test)
{
#if defined(ZTS) && defined(COMPILE_DL_TEST)
      ZEND_TSRMLS_CACHE_UPDATE();
#endif

      REGISTER_LONG_CONSTANT("TEST_SCALE_FACTOR", 2,
          CONST_CS | CONST_PERSISTENT);
      return SUCCESS;
}
```

You can access a new constant after the extension rebuild and reinstallation.

```
$ php -r 'var_dump(TEST_SCALE_FACTOR);'
int(2)
```

Of course, there are various other API macros to declare constants of different value types:

- REGISTER_NULL_CONSTANT(name, flags): Constant with value NULL.
- REGISTER_BOOL_CONSTANT(name, bval, flags): FALSE or TRUE.
- REGISTER_LONG_CONSTANT(name, lval, flags): Any long number.
- REGISTER_DOUBLE_CONSTANT(name, dval, flags): A double number.
- REGISTER_STRING_CONSTANT(name, str, flags): A zero terminated string.
- REGISTER_STRINGL_CONSTANT(name, str, len, flags): A string (with length).

It's also possible to use the similar REGISER_NS_...() group of macros to declare constants in some namespaces. See Zend/zend_constants.h for complete PHP constants API.

MODULE GLOBAL VARIABLES

For now, our new scale() function is purely functional. It doesn't depend on any internal state and it always returns a value that's only based on input arguments. However, some real-world functions access or update global state. In C, it's usually stored in global variables, but multi-threaded software has to use some tricks to make a distinction between states of different threads.

PHP is usually built to work in context of single thread, but it also may be configured for multi-threading. In any case, to be portable across different configurations, PHP recommends declaring global variables as a specially declared "module globals" structure.

The following code should be added at the end of "php_test.h" file:

```
ZEND_BEGIN_MODULE_GLOBALS(test)
      zend_long scale;
ZEND_END_MODULE_GLOBALS(test)

ZEND_EXTERN_MODULE_GLOBALS(test)

#define TEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(test, v)
```

The module global variables are declared between ZEND_BEGIN_MODULE_GLOBALS and ZEND_END_MODULE_GLOBALS macros. These are just plain C declarations. Actually, the C preprocessor converts them into "zend_tests_globlas" C structure definition. ZEND_EXTERN_MODULE_GLOBLAS() defines an external C name to access the structure and TEST_G() macro provides a way to access our module global variables. So instead of global "scale", we will use TEST_G(scale).

In the "test.c" file we should declare the real module global variable:

```
ZEND_DECLARE_MODULE_GLOBALS(test)
```

We will also define a GINIT callback to initialize this structure. This callback is called before MINIT, so we have to move the thread-local storage cache initialization code here:

```
static PHP_GINIT_FUNCTION(test)
{
#if defined(COMPILE_DL_BCMATH) && defined(ZTS)
      ZEND_TSRMLS_CACHE_UPDATE();
#endif
      test_globals->scale= 1;
}
```

We should also add information about module global variables and their initialization callback into the extension entry structure:

```
zend_module_entry test_module_entry = {
      STANDARD_MODULE_HEADER,
      "test",                                 /* Extension name */
      test_functions,                         /* zend_function_entry */
      PHP_MINIT(test),                        /* PHP_MINIT - Module initialization */
      NULL,                                   /* PHP_MSHUTDOWN - Module shutdown */
      NULL,                                   /* PHP_RINIT - Request initialization */
      NULL,                                   /* PHP_RSHUTDOWN - Request shutdown */
      PHP_MINFO(test),                        /* PHP_MINFO - Module info */
      PHP_TEST_VERSION,                       /* Version /
      PHP_MODULE_GLOBALS(test),               /* Module globals */
      PHP_GINIT(test),                        /* PHP_GINIT — Globals initialization */
      NULL,                                   /* PHP_GSHUTDOWN — Globals shutdown */
      NULL,
      STANDARD_MODULE_PROPERTIES_EX
};
```

And now we can update our "scale" function to use the module global variable:

```
PHP_FUNCTION(test_scale)
{
      double x;

      ZEND_PARSE_PARAMETERS_START(1, 1)
          Z_PARAM_DOUBLE(x)
      ZEND_PARSE_PARAMETERS_END();

      RETURN_DOUBLE(x * TEST_G(scale));
}
```

It works after extension recompilation and reinstallation:

```
$ php -r 'var_dump(test_scale(5));'
float(5)
```

CONFIGURATION DIRECTIVES

What else can we do? We may implement a way to define the value of our "scale" factor through configuration directive in php.ini. This is done by two additional pieces of code in "test.c."

The first defines configuration directives, their names, default values, types, and storage locations:

```
PHP_INI_BEGIN()
      STD_PHP_INI_ENTRY("test.scale", "1", PHP_INI_ALL, OnUpdateLong, scale,
          zend_test_globals, test_globals)
PHP_INI_END()
```

STD_PHP_INI_ENTRY() declares a configuration directive named "test.scale," with default value "1." PHP_INI_ALL indicates that it may be modified at any time (in php.ini, in per-directory configuration files and by ini_set() function during script excution). PHP_INI_SYSTEM (instead) would allow modification only during PHP startup (in php.ini). PHP_INI_PERDIR would allow modification only in php.ini and per-directory configuration files.

OnUpdateLong is a common callback that sets the integer value of the directive. (There are few other common callbacks like OnUpdateString.) "scale" is the module global variable name. "zend_test_globals" is the name of the structure (C type name) that keeps module global variables. "test_globals" is the global variable that keeps module global variables for a non-thread-safe build.

The complete PHP.ini API is defined at Zend/zend_ini.h and main/php_ini.h.

The second piece calls an API function that registers the directives declared in the previous block:

```
PHP_MINIT_FUNCTION(test)
{
      REGISTER_INI_ENTRIES();

      return SUCCESS;
}
```

PHP configuration directives may be set through a "-d" command line argument:

```
$ php -d test.scale=4 -r 'var_dump(test_scale(5));'
float(20)
```

COMMON PHP GLOBALS

Except for our own global variables, we may also need to get some values from common PHP global variables that are wrapped into similar module global structures and may be accessed through similar macros:

- CG(name): Compiler global variables.
- EG(name): Executor global variables.

All declarations can be viewed at Zend/zend_globals.h.

Basic PHP Structures

In this section, we will take a deeper look into the most important internal PHP data structures, which include:

- Values
- Strings
- Parameter parsing API
- Return Values

And you can see an example of using the basic PHP structures in the sample extension we have been building in this book.

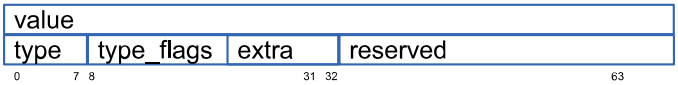
PHP VALUES (ZVAL)

Zval is the key PHP structure. It represents any PHP value (like a number, string, or array), following its simplified C definition.

```
typedef struct _zval_struct {
      union {
          zend_long          lval;
          double             dval;
          zend_refcounted    *counted;
          zend_string        *str;
          zend_array         *arr;
          zend_object        *obj;
          zend_resource      *res;
          zend_reference     *ref;
          …
      } value;
      zend_uchar        type;
      zend_uchar        type_flags;
      uint16_t          extra;
      uint32_t          reserved;
} zval;
```

PHP is a dynamically-typed language, and the same zval may keep values of different types. The first field of zval is a union of all possible value types. It may keep integer, double number, or a pointer to some dependent structure. Zval also keeps "type," "type_flags," and two reserved fields. This space would be used for proper structure alignment anyway, but by reserving it, we may store some dependent data.

In memory, zval is represented as two 64-bit words. The first word keeps the value — and the second word keeps the type, type_flags, extra, and reserved fields.



Zvals are usually allocated in the PHP stack or inside other data structures. They are almost never allocated on heap.

The single important zval flag is IS_TYPE_REFCOUNTED, which defines how to handle zval during copying and destruction.

If it's not set, zval is scalar.

To copy zval, we copy the first word (with its value) and a half of the second word (with type, type_flags and extra fields). We don't need any special actions to destroy them.

There are few pure scalar PHP types that are completely represented by this zval structure:

- IS_UNDEF: Uninitialized PHP local variable. (You usually won't get with this type in PHP extensions. PHP interpreter take cares about initialization, warnings and conversion to NULL.)
- IS_NULL: Null constant. (Value is not used.)
- IS_FALSE: False constant of boolean type. (Value is not used.)
- IS_TRUE: True constant of boolean type. (Value is not used.)
- IS_LONG: Long integer number.
- IS_DOUBLE: Long floating-point number.

There is a special C macro API to retrieve fields of zvals. All the macros are defined in two forms: plain, for zvals and with "_P" suffix, for pointers to zvals (e.g. Z_TYPE(zval) and Z_TYPE_P(zval_ptr)). The following are the most important:

- Z_TYPE_FLAGS(zv): Returns type flags (set of bits: IS_TYPE_REFCOUNTED and few others).
- Z_REFCOUNTED(zv): Returns true if the IS_TYPE_REFCOUNTED is set.
- Z_TYPE(zv): Returns type of the zval (IS_NULL, IS_LONG, etc).
- Z_LVAL(zv): Returns long integer value of the zval (the type must be IS_LONG).
- Z_DVAL(zv): Returns double value of the zval (the type must be IS_DOUBLE).

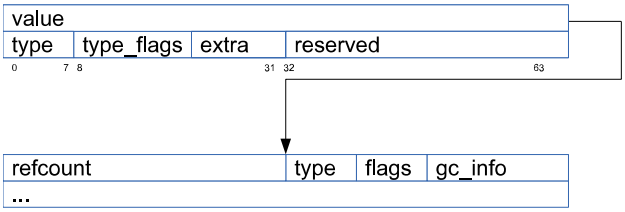
Another family of macros is used for zval initialization:

- ZVAL_UNDEF(zv): Initializes undefined zval
- ZVAL_NULL(zv): Initializes zval by null constant
- ZVAL_FALSE(zv): Initializes zval by false constant
- ZVAL_TRUE(zv): Initializes zval by true constant
- ZVAL_BOOL(zv, bval): Initializes zval by true constant if "bval" is true or by false otherwise
- ZVAL_LONG(zv, lval): Initializes a long integer number zval
- ZVAL_DOUBLE(zv, dval): Initializes a long floating point number zval

The most zval-related declarations are done at Zend/zend_types.h.

All non-scalar values — like strings, arrays, objects, resources, and references — are represented by structures specific to a certain type. Zval keeps just a pointer to this structure. In terms of object-oriented programming, all these specific structures have a common abstract parent class: zend_refcounted. It defines the format of the first 64-bit word of the structure. It contains the reference-counter, type, flags, and information used by a garbage collector.

Specific structures for concrete types are built on top of this one and add some additional data after this first word.



The following ref-counted types are possible:

- IS_STRING: PHP string.
- IS_ARRAY: PHP array.
- IS_REFERENCE: PHP reference.
- IS_OBJECT: PHP object.
- IS_RESOURCE: PHP resource.

Looking ahead, PHP strings and arrays may be non-reference-counted (or immutable) and behave similar to scalar values.

The following API macros are intended to work with reference-counted zvals (and there are also variants with "_P" suffix):

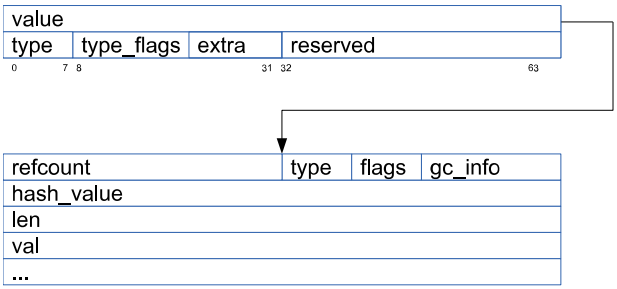
- Z_COUNTED(zv): Returns pointer to the dependent zend_refcounted structure.
- Z_REFCOUNT(zv): Returns reference-counter of the dependent zend_refcounted structure.
- Z_SET_REFCOUNT(zv, rc): Sets reference-counter of the dependent zend_refcounted structure.
- Z_ADDREF(zv): Increments reference-counter of the dependent zend_refcounted structure.
- Z_DELREF(zv): Decrements reference-counter of the dependent zend_refcounted structure.

The following universal macros may be used with both reference-counted and non-reference-counted zvals:

- Z_TRY_ADDREF(zv): Checks IS_TYPE_REFCOUNTED flag and increment reference-counter of the dependent zend_refcounted structure, if it's set.
- Z_TRY_DELREF(zv): Checks IS_TYPE_REFCOUNTED flag and decrements reference-counter of the dependent zend_refcounted structure, if it's set.
- zval_ptr_dtor(zv): Release the zval value: checks IS_TYPE_REFCOUNTED flag; decrements reference-counter of the dependent zend_refcounted structure, if it's set; call specific to zval type destruction function, if reference-counter became zero.
- ZVAL_COPY_VALUE(dst, src): Copies zval (value, type and type_flags) from "src" do "dst."
- ZVAL_COPY(dst, src): Copies zval (value, type and type_flags) from "src" do "dst" and increments reference-counter of the dependent zend_refcounted structure, if IS_TYPE_REFCOUNTED flag is set.

PHP STRINGS

Strings are represented by the dependent "zend_string" structure. Its first word repeats the word defined by "zend_refcounted" structure. "zend_string" also keeps the pre-calculated value of a hash function, string length, and the actual embedded characters. Hash value doesn't have to be pre-calculated. It's initialized by zero, lazily calculated on demand, and then reused. PHP string copying doesn't require duplication of the actual characters. Few zval structures may point to the same "zend_string" with the corresponding reference-counter value.



Strings may be immutable (or interned) without IS_TYPE_REFCOUNTED flag set, and in this case, they behave similar to scalars. The PHP engine doesn't need to perform any reference counting at all. Such strings are used only for reading and may be destroyed only at the end of request processing. They are never destroyed in the middle of request.

The same "zend_string" representation is not only used for PHP values, but also for all other character data in the PHP engine, such as names of functions, classes, and methods.

The different fields of zend_string may be accessed through the following API macros (and there are also variants with "_P" suffix for pointers to zvals):

- Z_STR(zv): Returns a pointer to corresponding zend_string structure.
- Z_STRVAL(zv): Returns a pointer to corresponding C string (char*).
- Z_STRLEN(zv): Returns length of the corresponding string.
- Z_STRHASH(zv): Returns hash value of the corresponding string. "zend_string.hash_value" is used as a cache to eliminate repeatable hash function calculation.

PHP strings values may be constructed through the following macros:

- ZVAL_STRING(zv, cstr): Allocates zend_string structure, initializes it with the given C zero-terminated string, and initializes PHP string zval.
- ZVAL_STRINGL(zv, cstr, len): Allocates zend_string structure, initializes it with the given C string and length, and initializes PHP string zval.
- ZVAL_EMPTY_STRING(zv): Initializes empty PHP string zval.
- ZVAL_STR(zv, zstr): Initializes PHP string zval using given zend_string.
- ZVAL_STR_COPY(zv, zstr): Initializes PHP string zval using given zend_string. Reference-counter of "zend_string" is incremented, if necessary.

Of course, it's also possible to work with "zend_string" structures directly, without zval. The following are the most important and useful API macros and functions:

- ZSTR_VAL(zstr): Returns a pointer to corresponding C string (char*).
- ZSTR_LEN(zstr): Returns length of the string.
- ZSTR_IS_INTERNED(zstr): Checks if the string is interned (or immutable).
- ZSTR_HASH(zstr): Returns hash_value of the string using hash_value as cache.
- ZSTR_H(zstr): Returns value of hash_value field (it may be zero which means it has not been calculated yet).
- zend_string_hash_func(zstr): Calculates and returns hash value of the string.
- zend_hash_func(cstr, len): Calculates and returns hash value of the given C string (char*) and specified length.
- ZSTR_EMPTY_ALLOC(): Returns an empty "zend_string." This macro doesn't actually allocate anything, but rather returns a pointer to a single interned "zend_string" structure.
- zend_string_alloc(len, persistent): Allocates memory for the "zend_string" structure of the given string length. The "persistent" argument tells whether the created string should relive request boundary. (Usually it shouldn't, and therefore "persistent" should be zero.)
- zend_string_safe_alloc(len, number, addition, persistent): Similar to zend_string_alloc(), but the final string size is calculated as (len * number + addition) and checked for possible overflow.zend_string_init(cstr, len, persistent): Allocates memory for the "zend_string" structure (similar to zend_alloc) and initializes it with the given C string (char*) and length.
- zend_string_copy(zstr): Creates a copy of the given "zend_string" and returns a pointer to the same string and increments reference-counter, if necessary.
- zend_string_release(zstr): Releases a pointer to the given "zend_string" and checks if the given string is reference-counted (not interned), decrements reference-counter, and frees memory, if it reached zero.
- zend_string_equals(zstr1, zstr2): Checks equality of two "zend_string" structures.
- zend_string_equals_literal(zstr, cstr): Checks equality of "zend_string" with a given C string literal.
- zend_string_equals_literal_ci(zstr, cstr): The case insensitive variant of zend_string_equals_literal().

The compete zend_string API is defined in Zend/zend_string.h.

PARAMETER PARSING API

The parameter parsing API is a way to get the values of an actual PHP parameter in an internal PHP function. We already used some elements of this API in our "test" extension. These were the blocks between ZEND_PARSE_PARAMETERS_START and ZEND_PARSE_PARAMETERS_END. Let's review this API in more detail.

First, there are two different parameters parsing APIs: the one we already used — the Fast Parameter Parsing API introduced in PHP 7 — and the old API that is compatible with PHP 5. There is no a single answer for which API is better for a particular function. The old API is slower, but its usage requires less machine code. If the function body is small and needs to be fast, it should use the Fast Parameter Parsing API, because the overhead of the old API may be bigger than the semantic part of the function itself. On the other hand, if the function is going to be slow, it doesn't make sense to care about parameter parsing overhead and increase code size.

FAST PARAMETER PARSING API

This new API is implemented using C pre-processor macros that are converted to almost optimal C code to fetch values of actual parameters into C variables, especially for this function.

- ZEND_PARSE_PARAMETERS_NONE(): This macro should be used for functions that don't expect any parameters. In case something is passed, the function will produce warning "expects exactly 0 parameters, %d given" and return NULL.
- ZEND_PARSE_PARAMETERS_START(min_num_args, max_num_args): This macro opens a block of parameter fetching code. The first argument is the minimal number of arguments that should be zero or more. The second argument is the maximum number of arguments. For functions with a variable number of arguments, its value should be -1. In case the number of passed parameters exceed the defined argument boundaries, the function will produce a warning about invalid number of arguments and return NULL.
- ZEND_PARSE_PARAMETERS_END(): This macro terminates the block of parameter fetching code.

There are a few ZPP macros inside the block between START and END macros. There is one macro for each argument, except for functions with variable number of arguments, where the last argument may receive many values. Required arguments should be separated from optional using the Z_PARAM_OPTIONAL macro.

The following are the most common macros for parameter parsing:

- Z_PARAM_BOOL(dest): Receives a boolean argument and stores the value of the actual parameter in a C variable of type zend_bool. Here and below, "receive" means checking the type of actual parameter and its conversion to required type, if possible, or producing a corresponding type incompatibility warning and returning NULL.
- Z_PARAM_LONG(dest): Receives the integer number argument and stores the value of the actual parameter in a C variable of type zend_long.
- Z_PARAM_DOUBLE(dest): Receives a floating point number argument and stores the value of the actual parameter in a C variable of type double.
- Z_PARAM_STR(dest): Receives a string argument and stores the value of the actual parameter in a C variable of type zend_string*.
- Z_PARAM_STRING(dest, dest_len): Receives a string argument and stores a pointer to the C string and the length of passed string in the given C variables "dest" of type char* and "dest_len" of type size_t.
- Z_PARAM_ARRAY_HT(dest): Receives a PHP array argument and stores the value of the actual parameter in a C variable of type HashTable. (PHP arrays and HashTables are described in the next chapter.)
- Z_PARAM_ARRAY(dest): Receives a PHP array argument and stores the value of the actual parameter in a C variable of type zval*.
- Z_PARAM_OBJECT(dest): Receives a PHP object argument and stores the value of the actual parameter in a C variable of type zval*.
- Z_PARAM_RESOURCE(dest): Receives a PHP resource argument and stores the value of the actual parameter in a C variable of type zval*.
- Z_PARAM_ZVAL(dest): Receives any PHP zval as passed without any conversions, and stores its value in a C variable of type zval*.
- Z_PARAM_ZVAL_DEREF(dest): Receives any PHP zval and de-reference, and stores the referenced value in a C variable of type zval*. This macro is useful for receiving parameters passed by reference. (We speak about them in the next chapter.)

- Z_PARAM_VARIADIC(spec, dest, dest_num): Receives the rest arguments as an array of zvals. This macro must be the last one in the parameter passing block. The "spec" argument may be "*" (zero or more parameters) or "+" (one or more parameters). The address of the parameters array is stored in the C variable "dest" of type zval* and the number of parameters in the variable "dest_num" of type "int."

Most of the above macros are available in extended variations with "_EX" and "_EX2" suffixes. Additional arguments of these macros allow control of nullability check, de-referencing, and separation.

OLD PARAMETER PARSING API

The old parameter parsing API was implemented as a C scanf() like function with a format string and following variable number of arguments, passed by address.

```
zend_parse_parameters(int num_args, const char *type_spec, …);
```

This function checks each letter of the "type_spec" string and performs parameter receiving and storing into the following variables accordingly. For example, we could use the following code in our test_scale() function.

```
PHP_FUNCTION(test_scale)
{
      double x;

      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "d", &x) == FAILURE) {
          return;
      }

      RETURN_DOUBLE(x * TEST_G(scale));
}
```

The "d" letter in "type_spec" assumes receiving of double argument and storing it in C variable of type double. Let's review most "type_spec" letters and their correlation with Fast Parameter Parsing API.

- '|' - Z_PARAM_OPTIONAL
- 'a' - Z_PARAM_ARRAY(dest)
- 'b' - Z_PARAM_BOOL(dest)
- 'd' - Z_PARAM_DOBLE(dest)
- 'h' - Z_PARAM_ARRAY_HT(dest)
- 'l' - Z_PARAM_LONG(dest)
- 'o' - Z_PARAM_OBJECT(dest)
- 'O' - Z_PARAM_OBJECT_OF_CLASS(dest, ce)
- 'r' - Z_PARAM_RESOURCE(dest)

- 's' - Z_PARAM_STRING(dest, dest_len)
- 'S' - Z_PARAM_STR(dest)
- 'z' - Z_PARAM_ZVAL(dest)
- '*' - Z_PARAM_VARIADIC('*', dest, dest_num)
- '+' - Z_PARAM_VARIADIC('+', dest, dest_num)

Each of the type specifiers may be followed by a modifier character:

- '/' - separate zval, if necessary. This is useful when function receives value by reference and is going to be modified. Otherwise, if the value is referenced from several places (reference-counter is more than 1), then all the values are going to be incorrectly modified at once.
- '!' - check if the actual parameter is null and set the corresponding pointer to NULL. For 'b', 'l' and 'd', an extra argument of type zend_bool* must be passed after the corresponding bool*, zend_long*, or double* argument.

RETURN VALUE

Each internal PHP function takes a "return_value" argument of type zval*. We may write into it using a family of ZVAL_...() macros described above, or use a special RETVAL_...() family of similar macros:

```
#define RETVAL_NULL()                   ZVAL_NULL(return_value)
#define RETVAL_BOOL(b)                  ZVAL_BOOL(return_value, b)
#define RETVAL_FALSE                    ZVAL_FALSE(return_value)
#define RETVAL_TRUE                     ZVAL_TRUE(return_value)
#define RETVAL_LONG(l)                  ZVAL_LONG(return_value, l)
#define RETVAL_DOUBLE(d)                ZVAL_DOUBLE(return_value, d)
#define RETVAL_STR(s)                   ZVAL_STR(return_value, s)
#define RETVAL_STR_COPY(s)              ZVAL_STR_COPY(return_value, s)
#define RETVAL_STRING(s)                ZVAL_STRING(return_value, s)
#define RETVAL_STRINGL(s, l)            ZVAL_STRINGL(return_value, s, l)
#define RETVAL_EMPTY_STRING()           ZVAL_EMPTY_STRING(return_value)
```

It's also possible to write value into "return_value" and perform actual return using RETURN_...() family of similar macros:

```
#define RETURN_NULL()                   {RETVAL_NULL(); return;}
#define RETURN_BOOL(b)                  {RETVAL_BOOL(b) return;}
#define RETURN_FALSE                    {RETVAL_FALSE; return;}
#define RETURN_TRUE                     {RETVAL_TRUE; return;
#define RETURN_LONG(l)                  {RETVAL_LONG(l); return}
#define RETURN_DOUBLE(d)                {RETVAL_DOUBLE(d); return;}
#define RETURN_STR(s)                   {RETVAL_STR(s); return;}
#define RETURN_STR_COPY(s)              {RETVAL_STR_COPY(s); return;}
#define RETURN_STRING(s)                {RETVAL_STRING(s); return;}
#define RETURN_STRINGL(s, l)            {RETVAL_STRINGL(s, l); return;}
#define RETURN_EMPTY_STRING()           {RETVAL_EMPTY_STRING(); return;}
```

USING BASIC PHP INTERNALS IN OUR EXAMPLE EXTENSION

Let's extend our test_scale() example to allow passing of the scale factor as the optional second argument — and make it behave differently, depending on type of the first argument. Perform multiplication when the first parameter is number, but keep the type of result to be the same as type of the first argument. In case the first argument is string, it should be repeated few times.

```
PHP_FUNCTION(test_scale)
{
      zval *x;
      zend_long factor = TEST_G(scale); // default value

      ZEND_PARSE_PARAMETERS_START(1, 2)
          Z_PARAM_ZVAL(x)
          Z_PARAM_OPTIONAL
          Z_PARAM_LONG(factor)
      ZEND_PARSE_PARAMETERS_END();

      if (Z_TYPE_P(x) == IS_LONG) {
          RETURN_LONG(Z_LVAL_P(x) * factor);
      } else if (Z_TYPE_P(x) == IS_DOUBLE) {
          RETURN_DOUBLE(Z_DVAL_P(x) * factor);
      } else if (Z_TYPE_P(x) == IS_STRING) {
          zend_string *ret = zend_string_safe_alloc(Z_STRLEN_P(x), factor, 0, 0);
          char *p = ZSTR_VAL(ret);
          while (factor-- > 0) {
              memcpy(p, Z_STRVAL_P(x), Z_STRLEN_P(x));
              p += Z_STRLEN_P(x);
          }
          *p = '\000';
          RETURN_STR(ret);
      } else {
          php_error_docref(NULL, E_WARNING, "unexpected argument type");
          return;
      }
}
ZEND_BEGIN_ARG_INFO(arginfo_test_scale, 0)
      ZEND_ARG_INFO(0, x)
      ZEND_ARG_INFO(0, factor)
ZEND_END_ARG_INFO()
```

At this point you should know all the PHP internals and understand the details of this function implementation.

Now it's time to test our new implementation.

```
$ php -r 'var_dump(test_scale(2));'
int(2)
$ php -r 'var_dump(test_scale(2,3));'
int(6)
$ php -r 'var_dump(test_scale(2.0, 3));'
float(6)
$ php -r 'var_dump(test_scale("2", 3));'
string(3) "222"
```

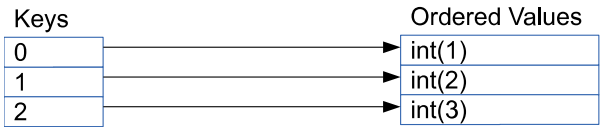
PHP Arrays

PHP arrays are complex data structures. They may represent an ordered map with integer and string keys to any PHP values (zval).

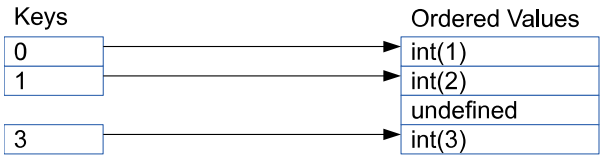
Internally, a PHP array is implemented as an adoptive data structure that may change its internal representation and behavior at run-time, depending on stored data. For example, if a script stores elements in an array with sorted and close numeric indexes (e.g. [0=>1, 1=>2, 3=>3]), it is going to be represented as a plain array. We will name such arrays – packed. Elements of packed arrays are accessed by offset, with near the same speed as C array. Once a PHP array gets a new element with a string (or "bad" numeric) key (e.g. [0=>1, 1=>3, 3=>3, "ops"=>4]), it's automatically converted to a real hash table with conflicts resolution.

The following examples explain how keys are logically organized in PHP:

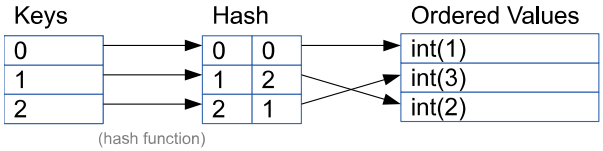
- $a = [1, 2, 3]; // packed array



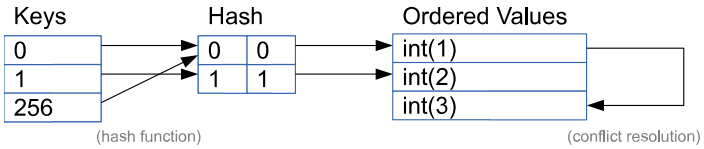
- $a = [0=>1, 1=>2, 3=>3]; //packed array with a "hole"



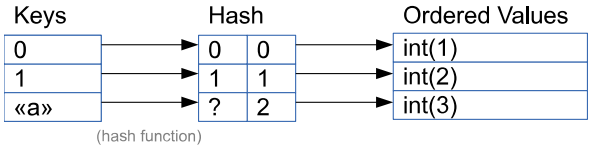
- $a = [0=>1, 2=>3, 1=>2]; // hash table (because of ordering) without conflicts



- $a = [0=>1, 1=>2, 256 =>3]; // hash table (because of density) with conflicts



- $a = [0=>1, 1=>2, "x"=>3]; // hash table (because of string keys)



Values are always stored as an ordered plain array. They may be simple iterated top-down or in reverse direction. Actually, this is an array of Buckets with embedded zvals and some additional information.

In packed arrays, value index is the same as numeric key. Offset is calculated as key * sizeof(Bucket).

HashTables uses additional arrays of indexes (Hash). It remaps value of hash function, calculated for numeric or string key value, to value index. Few array keys may make a collision, when they have the same value of hash function. They are resolved through linked lists of elements with the same hash value.

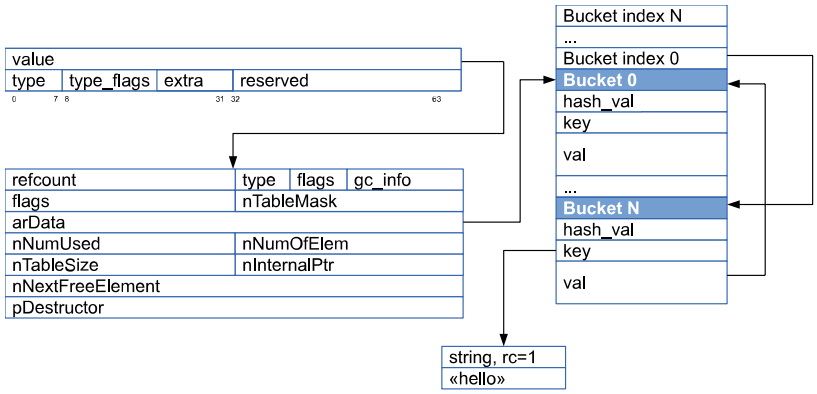
INTERNAL PHP ARRAY REPRESENTATION

Now, let's look into the internal PHP array representation. The value field of "zval" with IS_ARRAY type keeps a pointer to "zend_array" structure. It's "inherited" from zend_refcounted", that defines the format of the first 64-bit word with reference-counter.

Other fields are specific for zend_array or HashTable. The most important one is "arData", which is a pointer to a dependent data structure. Actually, they are two data structures allocated as a single memory block.

Above the address, pointed by "arData", is the "Hash" part (described above). Below, the same address is, the "Ordered Values" part. The "Hash" part is a turned-down array of 32-bit Bucket offsets, indexed by hash value. This part may be missed for packed arrays, and in this case, the Buckets are accessed directly by numeric indexes.

The "Ordered Values" part is an array of Buckets. Each Bucket contains embedded zval, string key represented by a pointer to zend_string (it's NULL for numeric key), and numeric key (or string hash_value for string key). Reserved space in zvlas is used to organize linked-list of colliding elements. It contains an index of the next element with the same hash_value.



Historically, PHP 5 made clear distinctions between arrays and HashTable structures. (HashTable didn't contain reference-counter.) However, in PHP 7, these structures were merged and became aliases.

PHP arrays may be immutable. This is very similar to interned strings. Such arrays don't require reference counting and behave in the same way as scalars values.

PHP ARRAY APIS

Use the following macros to retrieve zend_array from zval (also available with "_P" suffix for pointers to zval):

- Z_ARR(zv) – returns zend_array value of the zval (the type must be IS_ARRAY).
- Z_ARRVAL(zv) – historical alias of Z_ARR(zv).

Use the following macros and functions to work with arrays represented by zval:

- ZVAL_ARR(zv, arr) – initializes PHP array zval using given zend_array.
- array_init(zv) – creates a new empty PHP array.
- array_init_size(zv, count) – creates a new empty PHP array and reserves memory for "count" elements.
- add_next_index_null(zval *arr) – inserts new NULL element with the next index.
- add_next_index_bool(zval *ar, int b) – inserts new IS_BOOL element with value "b" and the next index.
- add_next_index_long(zval *arr, zend_long val) – inserts new IS_LONG element with value "val" and the next index.
- add_next_index_double(zval *arr, double val) – inserts new IS_DOUBLE element with value "val" and the next index.
- add_next_index_str(zval *arr, zend_string *zstr) – inserts new IS_DOUBLE element with value "zstr" and the next index.
- add_next_index_string(zval *arr, char *cstr) – creates PHP string from zero-terminated C string "cstr", and inserts it with the next index.
- add_next_index_stringl(zval *arr, char *cstr, size_t len)– creates PHP string from C string "cstr" with length "len", and inserts it with the next index.
- add_next_index_zval(zval *arr, zval *val) – inserts the given zval into array with the next index. Note that reference-counter of the inserted value is not changed. You should care about reference-counting yourself (e.g. calling Z_TRY_ADDREF_P(val)). All other add_next_index_...() functions are implemented through this function.

- add_index_...(zval *arr, zend_ulong idx, ...) – another family of functions to insert a value with the given numeric "idx". Variants with similar suffixes and arguments as the add_next_index_...() family above are available.
- add_assoc_...(zval *arr, char *key, ...) – another family of functions to insert a value with a given string "key", defined by zero-terminated C string.
- add_assoc_..._ex(zval *arr, char *key, size_t key_len, ...) – another family of functions to insert a value with a given string "key", defined by C string and its length.

Here are a few functions that can work directly with zend_array:

- zend_new_arra(count) – creates and returns new array (it reserves memory for "count" elements).
- zend_array_destroy(zend_array *arr) – frees memory allocated by array and all its elements.
- zend_array_count(zend_array *arr) – returns number of elements in array.
- zend_array_dup(zend_array *arr) – creates another array identical to the given one.

zend_array and HashTable are represented identically. And each zend_array is also a HashTable, but not vice-verse. zend_arrays may keep only zvals as elements. Generalized HashTables may keep pointers to any data structures. Technically this is represented by a special zval type IS_PTR. The HashTable API is quite extensive, so here we give just a quick overview:

- zend_hash_init() – initializes a hash table. The HashTable itself may be embedded into another structure, allocated on stack or through malloc()/emalloc(). One of the arguments of this function is a destructor callback, that is going to be executed for each element removed from HashTable. For zend_arrays this is zval_ptr_dtor().
- zend_hash_clean() – removes all elements of HashTable.
- zend_hash_destroy() – frees memory allocated by HashTable and all its elements.
- zend_hash_copy() – copies all elements of the given HashTable into another one.
- zend_hash_num_elements() – returns number of elements in HashTable.
- zend_hash_[str_|index_]find[_ptr|_deref|_ind]() – finds and returns element of HashTable with a given string or numeric key. Returns NULL, if key doesn't exist.
- zend_hash_[str_|index_]exists[_ind]() – checks if an element with the given string or numeric key exists in the HashTable.
- zend_hash_[str_|index_](add|update)[_ptr|_ind]() – adds new or updates existing elements of HashTable with given string or numeric key. "zend_hash...add" functions return NULL, if the element with the same key already exists. "zend_hash...update" functions insert new element, if it didn't exist before.
- zend_hash_[str_|index_]del[_ind]() – removes element with the given string or numeric key from HashTable.
- zend_symtable_[str_]find[_ptr|_deref|_ind]() – is similar to zend_hash_find...(), but the given key is always represented as string. It may contain a numeric string. In this case, it's converted to number and zend_hash_index_find...() is called.
- zend_symtable_[str_|]exists[_ind]() – is similar to zend_hash_exists...(), but the given key is always represented as string. It may contain a numeric string. In this case, it's converted to number and zend_hash_index_exists...() is called.

- zend_symtable_[str_](add|update)[_ptr|_ind]() – is similar to zend_hash_add/update…(), but the given key is always represented as string. It may contain a numeric string. In this case, it's converted to number and zend_hash_index_add/update…() is called.
- zend_symtable_[str_|]del[_ind]() – is similar to zend_hash_del…(), but the given key is always represented as string. It may contain a numeric string. In this case, it's converted to number and zend_hash_index_del…() is called.

There are also a number of ways to iterate over HashTable:

- ZEND_HASH_FOREACH_KEY_VAL(ht, num_key, str_key, zv) – a macro that starts an iteration loop over all elements of the HashTable "ht". The nested C code block is going to be called for each element. C variables "num_key", "str_key" and "zv" are going to be initialized with numeric key, string key and pointer to element zval. For elements with numeric keys, "str_key" is going to be NULL. There are more similar macros to work only with value, keys, etc. There are also similar macros to iterate in the reverse order. The usage of this macro is going to be demonstrated in the next example.
- ZEND_HASH_FOREACH_END() – a macro that ends an iteration loop.
- zend_hash_[_reverse]apply() – calls a given callback function for each element of the HashTable.
- zend_hash_apply_with_argument[s]() – calls a given callback function for each element of the given HashTable, with additional argument(s).

See more information in Zend/zend_hash.h.

USING PHP ARRAYS IN OUR EXAMPLE EXTENSION

Let's extend our test_scale() function to support arrays. Let it return another array with preserved keys and scaled values.

Because the element of an array may be another array (and recursively deeper), we have to separate the scaling logic into a separate recursive function do_scale(). The logic for IS_LONG, IS_DOUBLE and IS_STRING is kept the same, except, that our function now reports SUCCESS or FAILURE to the caller and therefore we have to replace our RETURN_…() macros with RETVAL_…() and "return SUCCESS".

```
static int do_scale(zval *return_value, zval *x, zend_long factor)
{
      if (Z_TYPE_P(x) == IS_LONG) {
          RETVAL_LONG(Z_LVAL_P(x) * factor);
      } else if (Z_TYPE_P(x) == IS_DOUBLE) {
          RETVAL_DOUBLE(Z_DVAL_P(x) * factor);
      } else if (Z_TYPE_P(x) == IS_STRING) {
          zend_string *ret = zend_string_safe_alloc(Z_STRLEN_P(x), factor, 0, 0);
          char *p = ZSTR_VAL(ret);

          while (factor-- > 0) {
              memcpy(p, Z_STRVAL_P(x), Z_STRLEN_P(x));
              p += Z_STRLEN_P(x);
          }
          *p = '\000';
          RETVAL_STR(ret);
      } else if (Z_TYPE_P(x) == IS_ARRAY) {
          zend_array *ret = zend_new_array(zend_array_count(Z_ARR_P(x)));
          zend_ulong idx;
          zend_string *key;
          zval *val, tmp;

          ZEND_HASH_FOREACH_KEY_VAL(Z_ARR_P(x), idx, key, val) {
              if (do_scale(&tmp, val, factor) != SUCCESS) {
                      return FAILURE;
              }
              if (key) {
                      zend_hash_add(ret, key, &tmp);
              } else {
                      zend_hash_index_add(ret, idx, &tmp);
              }
          } ZEND_HASH_FOREACH_END();
          RETVAL_ARR(ret);
      } else {
          php_error_docref(NULL, E_WARNING, "unexpected argument type");
          return FAILURE;
      }
      return SUCCESS;
}
PHP_FUNCTION(test_scale)
{
      zval *x;
      zend_long factor = TEST_G(scale); // default value
      ZEND_PARSE_PARAMETERS_START(1, 2)            Z_PARAM_ZVAL(x)
          Z_PARAM_OPTIONAL
          Z_PARAM_LONG(factor)
      ZEND_PARSE_PARAMETERS_END();
      do_scale(return_value, x, factor);
}
```

The new code for IS_ARRAY argument creates an empty resulting array (reserving the same number of elements, as in source array). It then iterates through source array element and calls the same do_scale() function for each element, storing the temporary result in tmp zval. Then it adds this temporary value into the resulting array under the same string key or numeric index.

Let's test new functionality...

```
$ php -r 'var_dump(test_scale([2, 2.0, "x" => ["2"]], 3));'
array(3) {
  [0]=>
  int(6)
  [1]=>
  float(6)
  ["x"]=>
  array(1) {
    [0]=>
    string(3) "222"
  }
}
```

Works fine, but, really, our function has a bug. It may leak memory on some edge conditions.

Catching Memory Leaks

Let's try to pass array with a value of some unexpected type:

```
$ php -r 'var_dump(test_scale([null]));'
Warning: test_scale(): unexpected argument type in Command line code on line 1
NULL
[Wed Jan 22 13:56:11 2020]  Script:  'Standard input code'
/home/dmitry/tmp/php-src/Zend/zend_hash.c(256) :  Freeing 0x00007f8189c57840 (56 bytes), script=Standard
input code
=== Total 1 memory leaks detected ===
```

We see our expected warning and NULL result, but then we see some debug info about leaked memory from internal PHP memory debugger. Note, that this information is only available in DEBUG PHP build, and this is one of the reasons, I recommend, to use DEBUG build during development. The information above says that 56-bytes of memory allocated on line 256 of Zend/zend_hash.c was leaked. This is the body of _zend_new_array() and we may already guess where it's called from, because we call it just once. However, in real-life we can't be sure about the call site, and it would be great to get a back-trace of the leaked allocation.

On Linux we may use valgrind. It's a great tool, that can catch memory-leaks and other incorrect memory access problems (e.g. use-after-free and out-of-boundary). Valgrind emulates the program with an overridden system memory manager (malloc, free and related functions) and catches inconsistencies.

Looking ahead, PHP uses its own memory manager and we should switch to system one, using USE_ZEND_ALLOC environment variable. It also makes sense to disable extension unloading.

```
$ USE_ZEND_ALLOC=0 ZEND_DONT_UNLOAD_MODULES=1 valgrind --leak-check=full \
     php -r 'var_dump(test_scale([null]));'
...
==19882== 56 bytes in 1 blocks are definitely lost in loss record 19 of 27
==19882==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==19882==    by 0x997CC5: __zend_malloc (zend_alloc.c:2975)
==19882==    by 0x996C30: _malloc_custom (zend_alloc.c:2416)
==19882==    by 0x996D6E: _emalloc (zend_alloc.c:2535)
==19882==    by 0x9E13BE: _zend_new_array (zend_hash.c:256)
==19882==    by 0x4849AE0: do_scale (test.c:66)
==19882==    by 0x4849F69: zif_test_scale (test.c:100)
==19882==    by 0xA3CE1B: ZEND_DO_ICALL_SPEC_RETVAL_USED_HANDLER (zend_vm_execute.h:1313)
==19882==    by 0xA9D0E8: execute_ex (zend_vm_execute.h:53564)
==19882==    by 0xAA11A0: zend_execute (zend_vm_execute.h:57664)
==19882==    by 0x9B7D0B: zend_eval_stringl (zend_execute_API.c:1082)
==19882==    by 0x9B7EBF: zend_eval_stringl_ex (zend_execute_API.c:1123)
...
```

Now we can be sure: the source of our memory-leak is a call of zend_new_array() function from our do_scale(). To fix it, we should destroy the array in case of FAILURE.

```
    } else if (Z_TYPE_P(x) == IS_ARRAY) {
        zend_array *ret = zend_new_       array(zend_array_count(Z_ARR_P(x)));
        zend_ulong idx;
        zend_string *key;
        zval *val, tmp;
        ZEND_HASH_FOREACH_KEY_VAL(Z_ARR_P(x), idx, key, val) {
            if (do_scale(&tmp, val, factor) != SUCCESS) {
                    zend_array_destroy(ret);
                    return FAILURE;
            }
            if (key) {
                    zend_hash_add(ret, key, &tmp);
            } else {
                    zend_hash_index_add(ret, idx, &tmp);
            }
        } ZEND_HASH_FOREACH_END();
        RETVAL_ARR(ret);
    } else {
```

Don't forget to test this.

Valgrind is much smarter then the internal PHP memory debugger and in case you cover your extension with *.phpt regression tests, you may run all of them under valgrind.

```
$ make test TESTS="-m"
```

PHP Memory Management

It's a good time to say few words about PHP Memory Manager.

The PHP Memory Manager API looks very much like classical libc malloc API, but it uses separate heap and it is especially optimized for PHP requirements. Usually, all the memory allocated during request processing should be freed at the end of a request at once. PHP allocator is especially optimized to do this extremely fast and without system memory fragmentation. It also avoids thread-safe checks, because even in multi-thread environment, each PHP thread is going to use a separate heap.

- emalloc(size_t size) – allocates the given amount of memory in the PHP request heap and returns pointer.
- safe_emalloc(size_t num, size_t size, size_t offset) – calculates the amount of required memory as (size * num + offset), checks for possible overflow, and allocates memory similar to emalloc().
- ecalloc(size_t num, size_t size, size_t offset) – allocates memory similar to safe_emalloc() and clears it (fill by zero byte).
- erealloc(void *ptr, size_t new_size) – reallocates a pointer, previously allocated in PHP request heap. This function may truncate or extend the allocated memory block, may move it into another memory location, or re-size in place. In case ptr is NULL, it's equivalent to emalloc().
- efree(void *ptr) – frees the memory block previously allocated in PHP request heap.

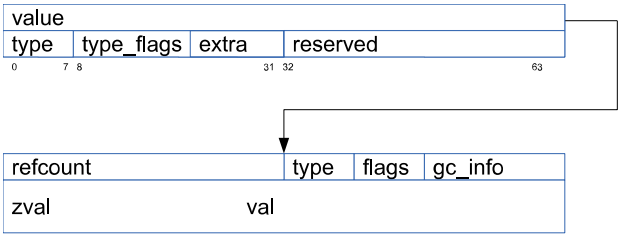
The compete Zend Memory Manager API is defined in Zend/zend_alloc.h.

PHP References

Usually, when you pass a parameter to function, you do it by value, and it means, the called function cannot modify it. In general, in a PHP extension you may modify a parameter passed by value, but most probably, this will lead to memory errors and crashes. In case you need to modify the argument of function (like sort() function does), it must be passed by reference.

Passing by reference is the main use-case for php references, but they also may be used everywhere inside other data structures (e.g. element of array).

Internally, they are represented by zval with IS_REFERENCE as type and pointer to zend_reference structure as value. As all reference-counted types, it's inherited from the zend_refcounted structure, which defines the format of the first 64-bit word. The rest of the structure is another embedded zval.



There are few C macros to check or retrieve fields of reference zvals (and the same macros with "_P" suffix take pointers to zvals):

- Z_ISREF(zv) – checks if the value is a PHP reference (has type IS_REFERENCE).
- Z_REF(zv) – returns dependent zend_reference structure (type must be IS_REFERENCE).
- Z_REFVAL(zv) – returns a pointer to the referenced value (zval).

There are also few macros for constructing references and de-referencing:

- ZVAL_REF(zv, ref) – initializes zval by IS_REFERENCE type and give zend_reference pointer.
- ZVAL_NEW_EMPTY_REF(zv) – initializes zval by IS_REFERENCE type and a new zend_reference structure. Z_REFVAL_P(zv) needs to be initialized after this call.
- ZVAL_NEW_REF(zv, value) – initializes zval by IS_REFERENCE type and a new zend_reference structure with a given value.
- ZVAL_MAKE_REF_EX(zv, refcount) – converts "zv" to PHP reference with the given reference-counter.
- ZVAL_DEREF(zv) – if "zv" is a reference, it's de-referenced (a pointer to the referenced value is assigned to "zv").

USING PHP REFERENCES IN OUR EXAMPLE EXTENSIONS

Let's try to pass a reference to our test_scale() function.

```
$ php -r '$a = 5; var_dump(test_scale([&$a], 2));'
Warning: test_scale(): unexpected argument type in Command line code on line 1
NULL
```

References are not supported.

To fix this, we should just add de-referencing.

```
static int do_scale(zval *return_value, zval *x, zend_long factor)
{
      ZVAL_DEREF(x);
      if (Z_TYPE_P(x) == IS_LONG) {
          RETVAL_LONG(Z_LVAL_P(x) * factor);
```

Now everything is fine:

```
$ php -r '$a = 5; var_dump(test_scale([&$a], 2));'
array(1) {
  [0]=>
  int(10)
}
```

Let's also convert our test_scale() to a function test_scale_ref() that won't return any value, but will receive argument by reference and multiply the passed value in-place.

```
static int do_scale_ref(zval *x, zend_long factor)
{
      ZVAL_DEREF(x);
      if (Z_TYPE_P(x) == IS_LONG) {
          Z_LVAL_P(x) *= factor;
      } else if (Z_TYPE_P(x) == IS_DOUBLE) {
          Z_DVAL_P(x) *= factor;
      } else if (Z_TYPE_P(x) == IS_STRING) {
          size_t len = Z_STRLEN_P(x);
          char *p;

          ZVAL_STR(x, zend_string_safe_realloc(Z_STR_P(x), len, factor, 0, 0));
          p = Z_STRVAL_P(x) + len;
          while (--factor > 0) {
              memcpy(p, Z_STRVAL_P(x), len);
              p += len;
          }
          *p = '\000';
      } else if (Z_TYPE_P(x) == IS_ARRAY) {
          zval *val;
          ZEND_HASH_FOREACH_VAL(Z_ARR_P(x), val) {
              if (do_scale_ref(val, factor) != SUCCESS) {
                    return FAILURE;
              }
          } ZEND_HASH_FOREACH_END();
      } else {
          php_error_docref(NULL, E_WARNING, "unexpected argument type");
          return FAILURE;
      }
      return SUCCESS;
}

PHP_FUNCTION(test_scale_ref)
{
      zval *  x;
      zend_long factor = TEST_G(scale); // default value

      ZEND_PARSE_PARAMETERS_START(1, 2)
          Z_PARAM_ZVAL(x)
          Z_PARAM_OPTIONAL
          Z_PARAM_LONG(factor)
      ZEND_PARSE_PARAMETERS_END();

      do_scale_ref(x, factor);
}

ZEND_BEGIN_ARG_INFO(arginfo_test_scale_ref, 1)
      ZEND_ARG_INFO(1, x) // pass by reference
      ZEND_ARG_INFO(0, factor)
ZEND_END_ARG_INFO()

static const zend_function_entry test_functions[] = {
      PHP_FE(test_test1,            arginfo_test_test1)
      PHP_FE(test_test2,            arginfo_test_test2)
      PHP_FE(test_scale_ref,        arginfo_test_scale_ref)
      PHP_FE_END
};
```

Testing:

```
$ php -r '$x=5; test_scale_ref($x, 2); var_dump($x);'
int(10)
$ php -r '$x=5.0; test_scale_ref($x, 2);
var_dump($x);'
float(10)
$ php -r '$x="5"; test_scale_ref($x, 2);
var_dump($x);'
string(2) "55"
$ php -r '$x=[[5]]; test_scale_ref($x,
2); var_dump($x);'
array(1) {
[0]=>  array(1) {
      [0]=>
      int(10)
  }
}
```

Everything looks correct, but in some cases our function is going to behave incorrectly. See the next section for more details about how to avoid issues by using Copy on Write.

Copy on Write

The following code shouldn't modify variable $y but it does.

```
php -r '$x=$y=[5]; test_scale_ref($x, 2); var_dump($x, $y);'
array(1) {
      [0]=>
      int(10)
}
array(1) {
      [0]=>
      int(10)
}
```

This is a bug. And it occurs because we update some value in-place without taking into account its reference-counter. In case it's greater than 1, the same value is referenced from some other place. And we have to "separate" it (or perform a Copy-on-Write).

This may be done using few macros:

- SEPARATE_STRING(zv) – this will perform a copy-on-write for PHP string if its reference-counter is above 1.
- SEPARATE_ARRAY(zv) – this will perform a copy-on-write for PHP array if its reference-counter is above 1.

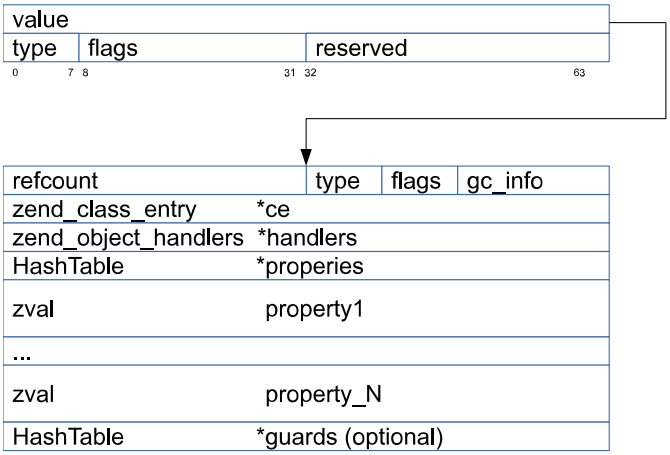
In our example, we need to "separate" only arrays, because zend_string_safe_realloc() takes care about reference-counting and performs copy-on-write itself. The fix is simple:

```
    } else if (Z_TYPE_P(x) == IS_ARRAY) {
        zval *val;

        SEPARATE_ARRAY(x);
        ZEND_HASH_FOREACH_VAL(Z_ARR_P(x), val) {
            if (do_scale_ref(val, factor) != SUCCESS) {
```

PHP Classes and Objects

Objects in PHP are represented by zend_object structure immediately followed by plain array of properties (zvals). The first word of object is defined by zend_refcounted structure and used for reference-counting, then there is a pointer to class entry structure, then pointer to object handlers table (similar to Virtual Methods Table) and HashTable of undeclared properties. All the declared properties are followed and may be accessed by offset.



There are two important dependent structures -zend_class_entry and zend_object_handlers. The first one keeps all the static information about the class, including its name, parent, methods, constants, properties — and their default values, values of static properties, etc. Here you can also see, a callback function “create_object”, that may be overridden to create something custom.

```
typedef struct zend_class_entry {
      char type;
      zend_string *name;
      zend_class_entry *parent;
      int refcount;
      uint32_t ce_flags;

      int default_properties_count;
      int default_static_members_count;
      zval *default_properties_table;
      zval *default_static_members_table;
      zval *static_members_table;
      HashTable function_table;
      HashTable properties_info;
      HashTable constants_table;
      zend_property_info **properties_info_table;
      ...
      zend_object* (*create_object)(zend_class_entry *class_type);
      ...
} zend_class_entry;
```

The second contains callback functions that determine the object behavior. We may create objects with custom behavior overriding this table and then changing some specific functions. This is advanced PHP internals knowledge. Now you can just read the list of handler names and guess what they are going to do.

```
typedef struct _zend_object_handlers {
      /* offset of real object header (usually zero) */
      int                                  offset;               /* object handlers */
      zend_object_free_obj_t               free_obj;             /* required */
      zend_object_dtor_obj_t               dtor_obj;             /* required */
      zend_object_clone_obj_t              clone_obj;            /* optional */
      zend_object_read_property_t          read_property;        /* required */
      zend_object_write_property_t         write_property;       /* required */
      zend_object_read_dimension_t         read_dimension;       /* required */
      zend_object_write_dimension_t        write_dimension;      /* required */
      zend_object_get_property_ptr_ptr_t   get_property_ptr_ptr; /* required */
      zend_object_get_t                    get;                  /* optional */
      zend_object_set_t                    set;                  /* optional */
      zend_object_has_property_t           has_property;         /* required */
      zend_object_unset_property_t         unset_property;       /* required */
      zend_object_has_dimension_t          has_dimension;        /* required */
      zend_object_unset_dimension_t        unset_dimension;      /* required */
      zend_object_get_properties_t         get_properties;       /* required */
      zend_object_get_method_t             get_method;           /* required */
      zend_object_call_method_t            call_method;          /* optional */
      zend_object_get_constructor_t        get_constructor;      /* required */
      zend_object_get_class_name_t         get_class_name;       /* required */
      zend_object_compare_t                compare_objects;      /* optional */
      zend_object_cast_t                   cast_object;          /* optional */
      zend_object_count_elements_t         count_elements;       /* optional */
      zend_object_get_debug_info_t         get_debug_info;       /* optional */
      zend_object_get_closure_t            get_closure;          /* optional */
      zend_object_get_gc_t                 get_gc;               /* required */
      zend_object_do_operation_t           do_operation;         /* optional */
      zend_object_compare_zvals_t          compare;              /* optional */
      zend_object_get_properties_for_t     get_properties_for;   /* optional */
} zend_object_handlers;
```

There are few APIs to work with PHP objects and classes. They are not compact, are not always consistent, and not well organized. I tried to collect the most important groups. The first one is the group of simple getter macros for basic internal object fields:

- Z_OBJ(zv) – returns pointer to zend_object structure from the zval (the type must be IS_OBJECT).
- Z_OBJCE(zv) – returns pointer to class entry of the given PHP object zval.
- Z_OBJ_HANDLE(zv) – returns object handle (unique number) of the given PHP object zval.
- Z_OBJ_HT(zv) – returns object handlers table of the given PHP object zval.
- Z_OBJ_HANDLER(zv, name) – returns named object handler from the table of the given PHP object zval.
- Z_OBJPROP(zv) – returns a HashTable with all properties (both declared and undeclared). Declared properties are stored as zvals of type IS_INDIRECT and the pointer to real value may be retrieved though Z_INDIRECT(zv) macro.

Few useful macros and functions to work with object or static class members:

- ZVAL_OBJ(zv, obj) – initializes zval with IS_OBJECT type and given zen_object.
- object_init(zv) – creates a new empty stdClass object and store it in zv.
- object_init_ex(zv, ce) – creates a new empty object of given class and store it in zv.
- zend_objects_new(ce) – creates and return object of the given class.
- zend_object_alloc(hdr_size, ce) – allocates a memory block for object with all the properties declared in given class. To create regular PHP objects, hdr_size must be equal to sizeof(zend_object). However, it's possible to request more memory and use it for keeping additional C data (this will be discussed later).
- zend_object_std_init(obj, ce) – initializes fields of the given zend_object as an object of given class.
- zend_object_release(obj) – releases object (decrements reference-counter and destroy object if it reaches zero).
- add_property_…(zv, name, value) – a family of functions to add named property of different values to the given object.
- zend_read_property(ce, zv, name, len, silent, ret) – reads the value of static property, and returns pointer to zval or NULL. "ret" is an address of zval where the property value should be stored (but not necessary).
- zend_read_static_property(ce, name, len, silent) – reads the value of static property, and returns pointer to zval or NULL.
- zend_update_property…(ce, zv, name, len, value) – a family of functions to assign values of different types to object property.
- zend_update_static_property…(ce, name, len, value) – a family of functions to assign values of different types to static property.

Here are a few functions to declare internal classes and their elements:

- INIT_CLASS_ENTRYce, name, functions) – initializes class entry structure with given name and list of methods.
- zend_register_internal_class(ce) – registers the given class entry in the global class table and returns the address of the real class entry.
- zend_register_internal_class_ex(ce, parent_ce) – similar to zend_register_internal_class(), but also perform inheritance from the given "parent_ce".
- zend_register_internal_interface(ce) – similar to zend_register_internal_class(ce), but registers interface instead.
- zend_class_implements(ce, num_interfaces, …) – makes class to implement given interfaces.
- zend_declare_class_constant_…(ce, name, len, value) – a family of function to declare class constants with different value types.
- zend_declare_property_…(ce, name, name, len, flags, value) – a family of functions to declare properties with different default value types. The "flags" argument may be used to declare public, protected, private and static properties.

Using OOP in our Example Extension

Let's try to extend our example extension, implementing the following PHP class in C.

```
<?php
class Scaller {
      const DEFAULT_FACTOR = 2;
      private $factor =  DEFAULT_FACTOR;
      function __construct($factor = self:: DEFAULT_FACTOR) {
          $this->factor = factor;
      }

      function scale(&$) {
          test_scale($x, $this[]factor);
      }
}
```

C-code versions of methods are written in a similar way to PHP internal functions. Just use PHP_METHOD() macro, with class and method names, instead of PHP_FUNCTION. In methods $this variable (zval) is available as ZEND_THIS macro. In method __construct(), we assign values to property $factor, using zend_update_property_long() function, and in method scale() read it, using zend_read_property().

```
static zend_class_entry *scaler_class_entry = NULL;

#define DEFAULT_SCALE_FACTOR 2

PHP_METHOD(Scaler, __construct)
{
      zend_long factor = DEFAULT_SCALE_FACTOR; // default value

      ZEND_PARSE_PARAMETERS_START(0, 1)
          Z_PARAM_OPTIONAL
          Z_PARAM_LONG(factor)
      ZEND_PARSE_PARAMETERS_END();

      if (ZEND_NUM_ARGS() > 0) {
          zend_update_property_long(Z_OBJCE_P(ZEND_THIS), ZEND_THIS,
              "factor", sizeof("factor")-1, factor);
      }
}

PHP_METHOD(Scaler, scale)
{
      zval *x, *zv, tmp;
      zend_long factor;

      ZEND_PARSE_PARAMETERS_START(1, 1)
          Z_PARAM_ZVAL(x)
      ZEND_PARSE_PARAMETERS_END();

      zv = zend_read_property(Z_OBJCE_P(ZEND_THIS), ZEND_THIS,
             "factor", sizeof("factor")-1, 0, &tmp);
      factor = zval_get_long(zv);

      do_scale_ref(x, factor);
}
```

Argument information descriptors are created and used in exactly the same way as for regular functions. Then, information about all the class methods must be collected into single array. This is similar to a list of extension functions, but using PHP_ME() macro instead of ZEND_FE(). PHP_ME(), which takes two additional arguments. The first one is the class name, and the fourth method flags (e.g. ZEND_ACC_PUBLIC, ZEND_ACC_PROTECTED, ZEND_ACC_PRIVATE, ZEND_ACC_STATIC).

```
ZEND_BEGIN_ARG_INFO(arginfo_scaler_construct, 0)
      ZEND_ARG_INFO(0, factor)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_scaler_scale, 0)
      ZEND_ARG_INFO(1, x) // pass by reference
ZEND_END_ARG_INFO()

static const zend_function_entry scaler_functions[] = {
      PHP_ME(Scaler, __construct, arginfo_scaler_construct, ZEND_ACC_PUBLIC)
      PHP_ME(Scaler, scale,       arginfo_scaler_scale,     ZEND_ACC_PUBLIC)
      PHP_FE_END
};
```

Finally, we have to register the class and its entities in MINIT.

INIT_CLASS_ENTRY() initializes temporary class entry structure, sets the name of the class, and adds the given class methods. zend_register_class_entry() registers class in the global class table and returns the resulting class entry. Then we add constant and property to the class.

```
PHP_MINIT_FUNCTION(test)
{
      zend_class_entry ce;

      REGISTER_INI_ENTRIES();

      INIT_CLASS_ENTRY(ce, "Scaler", scaler_functions);
      scaler_class_entry = zend_register_internal_class(&ce);

      zend_declare_class_constant_long(scaler_class_entry,
          "DEFAULT_FACTOR", sizeof("DEFAULT_FACTOR")-1, DEFAULT_SCALE_FACTOR);

      zend_declare_property_long(scaler_class_entry,
          "factor", sizeof("factor")-1, DEFAULT_SCALE_FACTOR, ZEND_ACC_PRIVATE);
      return SUCCESS;
}
```

And this works:

```
$ php -r '$o = new Scaler(5); $x = 5; $o->scale($x); var_dump($x, $o);'
int(25)
object(Scaler)#1 (1) {
      ["factor":"Scaler":private]=>
      int(5)
}
```

The implemented class doesn't make a lot of sense (except of educational), because it doesn't work better than the original class written in PHP. In practice, it makes sense to re-implement something in C, when it's more efficient, uses less memory, or just can't be written in PHP. For example, we may embed some C data into PHP object.

Embedding C Data into PHP Objects

Let's convert our regular PHP $factor property into embedded C data. To do this, zend_object must be allocated in a special way. C data and zend_object must be allocated together: C data above the pointed address and zend_object below. This way, such custom objects may be simply used by PHP core as regular objects and we may get C data using negative offsets. Anyway, PHP core should at least know the real address of the allocated block to free it. And we may inform it about this negative offset overriding the corresponding field in object_handlers.

So, we need to override object handlers table, and to do this we usea global variable of type "scaler_object_handlers". (We will initialize it later.) We declare new type "scaler_t" that unites our C data and zend_object. (It's possible to use many fields, of course.) zend_object must be the last element of the structure, because the PHP engine may also allocate memory for defined properties after it. (See the picture in the start of "PHP Classes and Objects" chapter.) We also define a macro Z_SCALER_P(), to perform pointer arithmetic and to get the address of our structure from the PHP object value, and a callback function scaler_new(), that creates objects of type Scale.

```
static zend_object_handlers scaler_object_handlers;

typedef struct scaler_t {
      zend_long   factor;
      zend_object std;
} scaler_t;

#define Z_SCALER_P(zv) \
      ((scaler_t*)((char*)(Z_OBJ_P(zv)) - XtOffsetOf(scaler_t, std)))

zend_object *scaler_new(zend_class_entry *ce)
{
      scaler_t *scaler = zend_object_alloc(sizeof(scaler_t), ce);

      zend_object_std_init(&scaler->std, ce);
      scaler->std.handlers = &scaler_object_handlers;
      return &scaler->std;
}
```

This function allocates the necessary amount of memory, initializes the standard PHP part of the object, overrides default object handlers, and returns the pointer to the standard part of the object.

The existing methods are converted to use new C field. They give the address of our custom object, and then use C data fields directly (no hash lookups, no type checks, conversions, etc).

```
PHP_METHOD(Scaler, __construct)
{
      scaler_t *scaler = Z_SCALER_P(ZEND_THIS);
      zend_long factor = DEFAULT_SCALE_FACTOR; // default value

      ZEND_PARSE_PARAMETERS_START(0, 1)
          Z_PARAM_OPTIONAL
          Z_PARAM_LONG(factor)
      ZEND_PARSE_PARAMETERS_END();

      scaler->factor = factor;
}

PHP_METHOD(Scaler, scale)
{
      zval *x;
      scaler_t *scaler = Z_SCALER_P(ZEND_THIS);

      ZEND_PARSE_PARAMETERS_START(1, 1)
          Z_PARAM_ZVAL(x)
      ZEND_PARSE_PARAMETERS_END();

      do_scale_ref(x, scaler->factor);
}
```

In MINIT, we override the "create_object" callback, copy the default object handlers table to our own, and override the "offset" field (to inform the engine about special object layout).

```
PHP_MINIT_FUNCTION(test)
{
      zend_class_entry ce;

      REGISTER_INI_ENTRIES();

      INIT_CLASS_ENTRY(ce, "Scaler", scaler_functions);
      scaler_class_entry = zend_register_internal_class(&ce);
      scaler_class_entry->create_object = scaler_new;

      memcpy(&scaler_object_handlers, &std_object_handlers,
          sizeof(zend_object_handlers));
      scaler_object_handlers.offset = XtOffsetOf(scaler_t, std);

      zend_declare_class_constant_long(scaler_class_entry,
          "DEFAULT_FACTOR", sizeof("DEFAULT_FACTOR")-1, DEFAULT_SCALE_FACTOR);

      return SUCCESS;
}
```

Testing:

```
$ php -r '$o = new Scaler(4); $x = 5; $o->scale($x); var_dump($x,$o);'
int(20)
object(Scaler)#1 (0) {
}
```

Everything is fine, except that we don't see the value of "factor" stored as C variable, but this is easy to fix. See the next section to learn how.

Overriding Object Handlers

If you remember, each object keeps an object handlers table. We started to use it in the previous chapter on embedding C data.

There are many different handlers, and all of them can be overridden to change object behavior. For example:

- ArrayObject overrides read/write/has/unset_dimension handlers to make an object behave as an array.
- ext/simplexml allows traversing XML tree through both property and dimension handlers.
- ext/ffi calls native functions through get_method handler etc.

We will override get_debug_info handler to make var_dump() print the value of our C factor field. This method takes the "object" argument and returns a HashTable with properties to zvals that are going to be displayed. The default implementation returns a real PHP properties table, but we don't have any PHP properties. Instead, we construct a new one and add the value of C factor, using the magic name "{factor}". We also specify that this is not the real properties table, but a temporary table, that should be freed after printing.

```
static HashTable* scaler_get_debug_info(zval *object, int *is_temp)
{
      scaler_t *scaler = OBJ_SCALER(object);
      HashTable *ret = zend_new_array(1);
      zval tmp;

      ZVAL_LONG(&tmp, scaler->factor);
      zend_hash_str_add(ret, "{factor}", sizeof("{factor}")-1, &tmp);
      *is_temp = 1;
      return ret;
}
```

Of course, we have to override the default value of get_debug_info handler, of Scaler class, by our own.

```
      memcpy(&scaler_object_handlers, &std_object_handlers,
          sizeof(zend_object_handlers));
      scaler_object_handlers.offset = XtOffsetOf(scaler_t, std);
      scaler_object_handlers.get_debug_info = scaler_get_debug_info;

      zend_declare_class_constant_long(scaler_class_entry,
          "DEFAULT_FACTOR", sizeof("DEFAULT_FACTOR")-1, DEFAULT_SCALE_FACTOR);
```

Works without problems:

```
$ php -r '$o = new Scaler(4); $x = 5; $o->scale($x); var_dump($x,$o);'
int(20)
object(Scaler)#1 (1) {
      ["{factor}"]=>
      int(4)
}
```

Answers to Common Extension Questions

To help you learn the basics about PHP, the tutorial we just completed described how to build a very simple practice extension. The following sections provide information that answer common questions and minimize issues relating to:

- Links to external libraries.
- Naming conventions.
- PHP resources

LINKING EXTERNAL LIBRARIES

Often, PHP extensions are created to provide binding to some third-party C library. In this case, your PHP extension must be linked with this library. This is done through extension configuration file "config.m4" (or "config.w32" on Windows). For example, the following "config,m4" would be used for zstd extension, that implements binding to libzstd:

```
PHP_ARG_WITH([zstd],
      [for zstd support],
      [AS_HELP_STRING([--with-zstd],
          [Include zstd support])])

if test "$PHP_ZSTD" != "no"; then
      PKG_CHECK_MODULES([LIBZSTD], [libzstd])
      PHP_EVAL_INCLINE($LIBZSTD_CFLAGS)
      PHP_EVAL_LIBLINE($LIBZSTD_LIBS, ZSTD_SHARED_LIBADD)

      PHP_SUBST(ZSTD_SHARED_LIBADD)

      AC_DEFINE(HAVE_ZSTD, 1, [ Have zstd support ])

      PHP_NEW_EXTENSION(zstd, zstd.c, $ext_shared)
fi
```

When an extension uses an external library, we use --with-<feature> option instead of --enabel-<feature>, and therefore PHP_ARG_WITH() macro instead of PHP_ARG_ENABLE(). Also, a few special macros were added to find the library and include paths, through pkg-config, and add special rules into the final Makefile.

NAMING CONVENTIONS

When you start writing a new extension, try to use a consistent naming convention. There are few alternatives:

- no prefix (scale).
- name prefix with underscore (test_scale).
- name prefix and mixed case (TestScale).
- namespace (Test\scale) – special variants if the macros (ZEND_NS_NAME, ZEND_NS_FENTRY, ZEND_NS_FE, INIT_NS_CLASS_ENTRY) should be used to construct namespaced names.
- use static class as a namespace (Test::scale) – it's possible to create a class and declare all functions as static methods of this class.

PHP RESOURCES

PHP has one more type: resource. This type was historically used when you had to keep some C data in PHP zval (e.g. file descriptors). It's represented as a child of zend_refcounted structure with a pointer to some C data structure.

PHP resources are still used in PHP extensions. However, it's not recommended to use them for new extensions, because you may implement smarter and more efficient solutions by embedding C data into PHP internal objects.