# How to contribute to PHP

Gina Peter Banyard (she/her)

February 15th 2024

The PHP Foundation

## Where to find these slides

https://gpb.moe/doc/slides/phpuk24.pdf

## Before we begin

- Create a new folder `phpuk24` and subfolder `custom-php`
- Clone the [`php/php-src`]() git repo into `phpuk24`
- `cd` into the cloned repo and run `./buildconf` followed by:

-
  ```
  ./configure -C CC=gcc CFLAGS="-DZEND_RC_DEBUG=1
  -DZEND_VERIFY_FUNC_INFO=1 -DZEND_TRACK_ARENA_ALLOC=1
  -ggdb3" --disable-all --enable-address-sanitizer \
  --enable-undefined-sanitizer --enable-debug \
  --enable-tokenizer --prefix /path/to/phpuk24/custom-php/
  ```
- Finally, `make -j$(getconf _NPROCESSORS_ONLN)`

## Who am I

The PHP Foundation

- Studied pure mathematics at ICL
- Working on PHP since 2018
  - Started with documentation
  - First RFC and php-src contributions in 2019
  - Paid by Foundation from April 2022
- No formal education in C

## Different way of contributing to PHP:

- The PHP documentation
- The php.net website
- Writing and improving tests for PHP
- PECL extensions
- Bundled extensions
- PHP itself

## The PHP documentation

Written in XML following the DocBook standard.

English language sources available on GitHub in the `php/doc-en` and `php/doc-base` repo.

A tutorial on how to contribute exists on `http://doc.php.net/tutorial/`

It is also possible to contribute to one of the existing translations (e.g. `php/doc-fr`)

## The PHP documentation

The `php/doc-base` repo contains the `configure.php` script that prepares the manual from the `manual.xml.in` skeleton.

Moreover, the docs must have the following directory structure:

```
php-doc/base/
       /en/
       /fr/
```

It also contains various QA scripts that could be enhanced!

## The PHP documentation: PhD

The manual is rendered via our custom render written in PHP named `PhD`.

It handles the indexing, search, and automatic linking for <`function`>, <`classname`>, and <`methodname`> tags. (Sadly no support for <`constant`> yet.) It is invoked with:

```
php -dmemory_limit="1024M" ./phd/render.php -P PHP -d ./
base/.manual.xml -f xhtml -f bigxhtml -f php -f tocfeed --
output ./output_bightml -r
```

## The php.net website

The website is also open source, and can be found in the `php/web-php` git repo on GitHub.

This also contains the CSS for the whole website and the rendered documentation.

## Tests

PHP and its extensions are tested via PHPT test files

How these files are formatted is described on the PHP QA website: https://qa.php.net/phpt_details.php

Code coverage for PHP and bundled extension can be found on Codecov: https://app.codecov.io/github/php/php-src/

Handy feature, failed tests generate a `.sh` file that can be executed with a `gdb` argument to launch the test with GDB

## PECL and Bundled extensions

- Mostly written in C and C++ (but Rust is also possible)
- Provides most of PHP's functionality (e.g. `ext/curl`, `ext/standard`, `ext/opcache`)
- Can do stuff not possible (yet?) in userland (e.g. operator overloading, overloading casting behaviour)

## PHP itself, a.k.a. Zend Engine

- Written in C
- Compiler/lexer/AST
- VM and opcodes
- Optimizer
- Engine APIs for extensions (e.g. ZPP)

## Resources to write C for PHP

**LXR** https://heap.space

**PHP Internals Book** www.phpinternalsbook.com

**Zend Tutorial** www.zend.com/resources/writing-php-extensions

**Outdated Engine API docs** https://phpinternals.net/

**Thomas Weinert's** Extension Samples GitHub repo

## For any and all repos

Helping with issue triage:

- Confirming the issue is valid
- Simplifying the script reproducing the issue

## A note on internals and RFCs

Internals is a mailing list that has an etiquette, such as no top posting.

Use `https://externals.io/` to search prior discussions

The PHP RFC Codex repo has a summary of discussions of various RFC ideas and how to behave on it to reduce friction.

Writing an RFC takes time and can be exhausting.

## What we will work on

## Creating an extension that provides the rational numbers $\mathbb{Q}$

## Creating our extension

1. `make install`
2. `php ./php-src/ext/ext_skel.php --dir ../ --ext rationals \`
   `--author "Name"`
3. `cd ../rationals`
4. `phpize`
5. `./configure --with-php-config=/path/to/phpuk24/custom-php/bin/php-config CFLAGS="-fsanitize=address -fsanitize=undefined"`
6. `make test TEST_PHP_ARGS="-j$(getconf _NPROCESSORS_ONLN) -q"`

## 🎉 It works! 🎉

Look around the files of the generated extension skeleton:

- `rationals.c`
- `rationals.stub.php`
- `rationals_arginfo.h`
- The `tests/` folder

## Let's define our class

- Add the `RationalNumber` class to our stub file
- Create a test which checks the class exist
- Execute `make test`

Make test which will do `make` which itself will run the `./build/gen_stub.php` script to generate the arginfo header.

## Let's define our class

The arginfo file now has a `register_class_RationalNumber` function

To register our class, we need to store the class entry somewhere:

```
static zend_class_entry *zend_ce_rational_number;
```

## Let's define our class

```
PHP_MINIT_FUNCTION(rationals)
{
  zend_ce_rational_number = register_class_RationalNumber();
  return SUCCESS;
}
```

And modify the `rationals_module_entry` struct to have the `PHP_MINIT(rationals)`

## Let's add a constructor!

```
PHP_METHOD(RationalNumber, __construct)
{
  zend_long numerator;
  zend_long denominator;

  ZEND_PARSE_PARAMETERS_START(2, 2)
    Z_PARAM_LONG(numerator)
    Z_PARAM_LONG(denominator)
  ZEND_PARSE_PARAMETERS_END();
}
```

## Let's set the values

```
zval *this_z = ZEND_THIS;
zend_object *this_obj = Z_OBJ_P(this_z);

zend_update_property_long(zend_ce_rational_number, this_obj,
"numerator", strlen("numerator"), numerator);
zend_update_property_long(zend_ce_rational_number, this_obj,
"denominator", strlen("denominator"), denominator);
```

## A 0 denominator makes no sense

```
if (denominator == 0) {
  zend_argument_value_error(2, "cannot be 0");
  RETURN_THROWS();
}
```

## Let's add useful methods

- Add an `add` method to the stubs
- `Z_PARAM_OBJ_OF_CLASS(zend_obj_ptr, zend_ce_rational_number)` or `Z_PARAM_OBJ_OF_CLASS_OR_LONG()` ZPP macro
- `zval *return_value` is defined and initialized to `IS_NULL` for every method/function.
- `object_init_ex(return_value, zend_ce_rational_number);`
- Use `zend_read_property()` function to read property
- `Z_LVAL_P(ptr)` access the value of a `zval *ptr` which is `IS_LONG`

## Extract common setting code

```
static void set_rational(zend_object *obj, zend_long num,
zend_long denominator) {
  // Common code
}
```

## Write more methods

- Subtraction
- Multiplication
- Division
- Modulo
- Exponentiation $\left(\frac{a}{b}\right)^x$ where $x$ integer
- Absolute value

## Adding operator overloading support

Adding custom object handlers:

```
static zend_object_handlers rational_object_handlers;
```

In MINIT:

```
zend_ce_rational_number->default_object_handlers =
&rational_object_handlers;
memcpy(&rational_object_handlers, &std_object_handlers,
sizeof(zend_object_handlers));
```

## Adding operator overloading support

```
rational_object_handlers.do_operation =
rational_do_operation;
```

## Writing the handler

```c
static zend_result rational_do_operation(uint8_t opcode, zval
*result, zval *op1, zval *op2)
```

The opcode is the VM opcode, if the opcode is supported return `SUCCESS` otherwise `FAILURE`.

Examples of opcode `ZEND_ADD`, `ZEND_BW_OR`, `ZEND_POW`

## Careful with op1 and op2

The op `zvals` might not be objects, nor objects of the correct type:

```
$object + $string;
$array + $object;
```

```
Z_TYPE_P(zval) == IS_OBJECT && instanceof_function(
  Z_OBJCE_P(zval), zend_ce_rational_number)
```

## Writing the handler

Binary assign op needs special treatment:

```
zval op1_copy;
if (result == op1) {
  ZVAL_COPY_VALUE(&op1_copy, op1);
  op1 = &op1_copy;
}
// CODE
if (op1 == &op1_copy) { zval_ptr_dtor(op1); }
```

## Adding a compare method

- Return a negative number if `self` is less than other, 0 if equal, and a positive one if greater than other (bonus normalise to $-1/0/1$)
- Note $\frac{1}{2} == \frac{2}{4}$

```
static zend_long gcd(zend_long op1, zend_long op2) {
  return (op1 % op2) ? gcd(op2, op1 % op2) : op2;
}
```

## Overloading the comparison operators

```
static int rational_compare(zval *op1, zval *op2) {}
```

And in MININT

```
rational_object_handlers.compare = rational_compare;
```

Note: Same problems with `zval` ops can happen as the do_operation handler

## Solutions

I have uploaded “solutions” to this workshop on GitHub’

## Extra challenges

While using LXR/heap.space, can you implement:

- Overloading the casting behaviour (`float`, `bool`, `string`)
- Add a function `rational_sum(array $rationals): RationalNumber`
- Parsing a string (e.g. `"5/48"`) into a RationalNumber

Thank you!