PHP FFI vs extension development

Which approach is better?

Bohuslav Šimek

Table of content

What is FFI?

A foreign function interface (FFI) is a mechanism by which a program written in one programming language can call routines or make use of services written in another.
[source: https://en.wikipedia.org/wiki/Foreign_function_interface]

What problems does FFI solve?

1. Reuse code which has already been written, eg.: database connectors.
2. Speed up some parts of the code, eg.: complex calculations.
3. Do something which is not supported in your language, eg.: low level access to HW.

Why use FFI?

Advantages of FFI, compared to extensions:

- Simpler usage - you do not leave PHP, no need to know other languages.
- Easier maintenance and deployment - no need for compilation or change the way how the application is deployed.
- More portability, more resistance to ZEND API changes - PHP internal API tend to change more often then the rest of the PHP.

Limitation of FFI

Limitations of FFI in comparison with extensions:

- No easy way to modify PHP internals.
- Extensions are usually faster.
- Supports only one target language (often, it’s C).

State of FFI in PHP 7.4

Brand new extension, which:

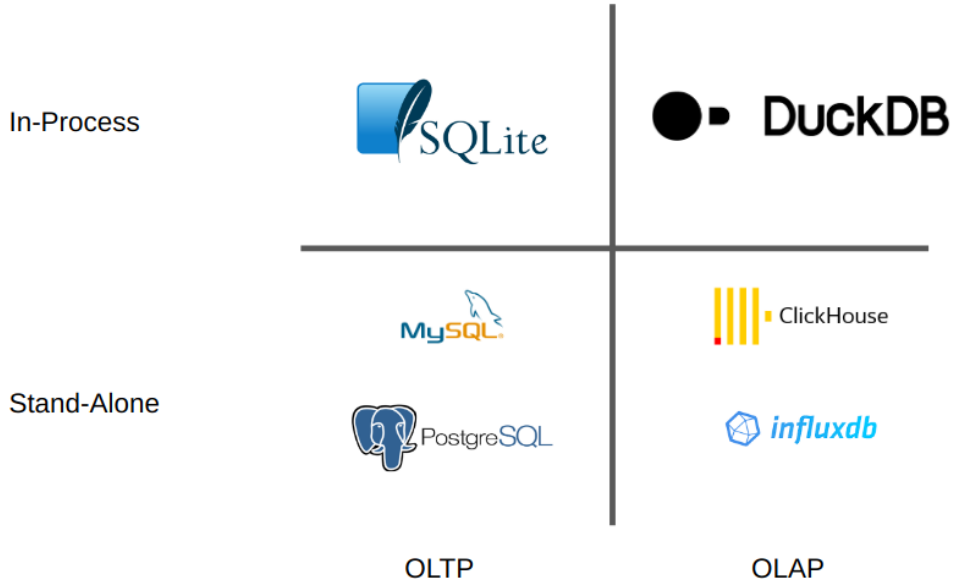
- is part of the core,
- is more mature than any existing solutions,
- is multiplatform,
- is maintained.

...but allows calling only C libraries (or exposing C ABI).

DuckDB - introduction

- In-process SQL OLAP column based database.
- C++11, no dependencies, single file build.
- Columnar-vectorized query execution engine.

In one sentence: **“The SQLite for data analytics”** with PostgreSQL flavour.



DuckDB - bindings

- APIs for Java, C, C++, and others
- No support for PHP! ⇒Ideal usecase for FFI and extension.

DuckDB - example

Implement basic example from DuckDB website[1].

Example in which we:

- create database in memory,
- insert some data and
- query stored data.

[1] https://github.com/duckdb/duckdb/blob/master/examples/embedded-c/main.c

How to use FFI?

Basic example with abs function. Abs function returns the absolute value of num. Let's rewrite PHP abs() function with help of FFI:

```
$ffi = FFI::cdef(
   'int abs(int j);',
   'libc.so.6'
);

var_dump($ffi->abs(-42)); // int(42)
```

PHP vs C - function signature

FFI provides an automatic conversion for simple data type, but this does not guarantee a "right" function call. For example function abs in PHP accept float and integer data type, but C counterpart does not.

```
$ffi = FFI::cdef(
   'int abs(int j);
   long int labs(long int j);',
   'libc.so.6'
);

var_dump($ffi->abs(-42));          // int(42)
var_dump($ffi->abs(-2147483649));  // int(2147483647)
var_dump($ffi->labs(-2147483649)); // int(2147483649)
```

Data types that go beyond the basic ones, such as strings, arrays, and structures, are more complicated. To use them properly, we must delve into the workings of C.

C for PHP developers

Basic introduction - PHP

PHP is a dynamic interpreted language.

- Nothing is AOT compiled to machine code.
- eval statement, reflection

PHP has dynamic/weak/gradual typing.

- Types are checked during runtime.
- Types are not mandatory.

Allow Multi-paradigm approach

- Mainly procedural and OOP.
- Provides some abstraction out of box.

PHP is a C like language and has some functions similar to the C standard library. But probably biggest difference is in the memory management, as PHP:

1. Automatic memory management - reference counting, garbage collector.
2. PHP guarantees memory safety => no direct access to memory.

Basic introduction - C

C can be considered the lowest-level of all general-purpose languages.

C is a static compiled language.

- Everything is compiled to machine code.
- No eval statement, no reflection

C is weakly typed:

- Types are checked during compilation.
- Types are mandatory, but can be bypassed.

Allow Multi-paradigm approach in theory:

- In reality it’s procedural, no OOP
- Virtually non synthetic sugar, few keywords.

Other features:

- Function and structure must be defined in advance.
- Same goes for variables.
- Strong preprocessor that allows a lot of magic.
- Undefined behaviour of some operations.

Undefined behaviour

- Some expressions yield undefined behaviour.
- Can act differently across compiler, platform and optimisation level.
- Eg.: use of an uninitialized variable.

Probably the biggest difference between PHP and C is memory management. In C:

- Manual memory management - if not done properly it can lead into memory leaks and seg faults.
- No memory safety => we can do anything.
- Direct access to memory.

PHP and C, common things

- Syntax
- Operators
- Control structures
- Function signature

PHP and C, function signature

In C:

```
int abs(int j);
```

In PHP:

```
function abs($j);
```

In PHP with types:

```
function abs(int $j) : int;
```

Signature differences:

- Mandatory types,
- Types position,
- Keyword “function” in PHP.

PHP and C, differences

- Different data types
- Memory management.
- Preprocessor and macros.

Data types C

- Basic data types (integer, float, etc.).
- Extended data types (arrays, string, etc.)
- User defined types (enums, structures)

Basic/primitive data types:

- char, int, float and double,
- modifiers signed, unsigned, short, and long,
- since C99 also bool,
- void - empty data type that has no value,
- pointers.

Extended data types:

- Arrays
- Strings (based on arrays)

Arrays in C

Size must be defined in advance usually during compilation[2].

- All elements must have a same type.
- Only number indexes.

Definition:

```
type arrayName[arraySize]
```

Example:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

- Their size is fixed, and cannot be changed after creation.
- There are no bounds checks.

Multidimensional arrays are also supported, but only of one type:

```
int threedim[5][10][4];
```

Pointers are used when we manipulate arrays.

---

[2] since C99 there are VLA, their usage is problematic.

Strings in C

- Array of char values with null terminating character (\0).
- Virtually the same as an array.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

OR

```
char greeting[] = "Hello";
```

Enums

- A data type that consists of integer constants.
- Numbers are not mandatory.

```
enum State {working = 1, failed = 0};
enum State {working, failed};
```

You can freely access them in file scope.

```
#include<stdio.h>

enum state {working = 1, failed = 0};

int main()
{
   enum state operation_result;
   operation_result = working;

   printf("%d",operation_result);
   return 0;
}
```

Enums - few interesting facts

- In the same enum individual items can have the same values.
- In same scope two enums cannot have same key:

```
enum State {working = 1, Failed = 0, guru_meditation = 0};
enum state {working, failed};
```

```
enum bg_job {working, failed};
```

Structures

- They do not have a direct counterpart in PHP.
- Allows to combine data items of different kinds.
- They have a fixed number of properties.
- They are like “objects”, but without methods (stdClass).
- Basic building block in C for most abstractions.

```
struct point {
   int x;
   int y;
   char *name;
};
```

Working with structure:

```
struct point p = { 1, 3 };
p.x = 48;

int x = p.x;
```

Unions

- Unions are special types of structures.
- Allows to store different types in the same memory block.
- In PHP all types are internally represented by union:

```
typedef union _zvalue_value {
   long lval;
   double dval;
   struct {
       char *val;
       int len;
   } str;
   HashTable *ht;
   zend_object_value obj;
} zvalue_value;
```

Pointers

Pointer - variable that stores the memory address of another variable located in computer memory. They are similar references in PHP, but **pointers are not references**. There is a whole page in documentation about this:
https://www.php.net/manual/en/language.references.arent.php

Biggest difference is that the pointer allows you to directly access memory and do a lot of operations on it.

Pointers are marked by * (asterisk) and their type must be provided, eg.:

```
type *var_name;
```

For integer:

```
int *ip;
```

Working with pointers:

- & operator - get an address
- * operator - get a value.

```
#include<stdio.h>

int main ( ){
   int meaning = 42;
   int * pInt;

   pInt = &meaning;

   printf (
      "Meaning address = %p\n",
      pInt
   );
   printf (
      "Meaning value = %d\n",
      *pInt
   );
}
```

Pointers and structures

- arrow operator
- st->sno is equivalent to (*st).sno

```
#include<stdio.h>

struct student{
   int sno;
};

int main ( ){
   struct student s;
   struct student *st;
   printf("enter sno:");
   scanf("%d", & s.sno);

   st = &s;
   printf (
      "Number = %d\n", st->sno
   );
}
```

Pointer arithmetics

- Pointer is an address
- We can do a mathematical operation on them!
- x[y] is identical to *(x + y)

```
#include <stdio.h>

const int MAX = 3;

int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   ptr = var;

   for ( i = 0; i < MAX; i++) {
       printf("Add %d = %p\n", i, ptr);
       printf("Val %d = %d\n", i, *ptr);

       /* move to the next location */
```

```
        ptr++;
    }

    return 0;
}
```

Void pointer

- Has no data type.
- Can be used for “everything”.
- During deference must be cast to some type.
- Void pointer definition:

```
void *p;
```

```
#include<stdio.h>

int main() {
    int a = 7;
    float b = 7.6;
    void *p;
    p = &a;
    printf(
        "Integer variable is = %d",
        *( (int*) p)
    );

    p = &b;
    printf(
        "\nFloat variable is = %f",
        *( (float*) p)
    );

    return 0;
}
```

Void pointer - universal function

```
void qsort (
        void *array, // void pointer to array
        size_t array_size,
```

```
       size_t data_type_size,
       int (*comp) (const void *a, const void *b)
);
```

Pointer to function

- Pointers can point to a function.
- Can be used as an argument in another function.
- Like a callback in PHP.

```
#include <stdio.h>

int sum(int x, int y)
{
   return x+y;
}


int main( )
{
   int (*fp)(int, int);
   fp = sum;
   int s = fp(10, 15);
   printf("Sum is %d", s);

   return 0;
}
```

Function qsort can be also used as an example for callback (pointer to function).

```
void qsort (
       void *array, // void pointer to array
       size_t array_size,
       size_t data_type_size,
       int (*comp) (const void *a, const void *b)
);

#include <stdio.h>
#include <stdlib.h>

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}
```

```
int main () {
   int n;
   int values[] = { 2, 3, 1};
   qsort(values, 3, sizeof(int), cmpfunc);

   printf("\nAfter sorting the list is: \n");
   for( n = 0 ; n < 3; n++ ) {
       printf("%d ", values[n]);
   }
}
```

Quicksort - peek to FFI

```
$ffi = FFI::cdef(
    "void qsort (void *array, size_t count, size_t size, int (*comp) (const
void *a, const void *b));",
   "libc.so.6"
);

$array = FFI::new("int[3]");

$array[0] = 2; $array[1] = 3;$array[2] = 1;

$cmp = function (FFI\CData $a, FFI\CData $b) {
   $aInt = FFI::cast("int", $a)->cdata;
   $bInt = FFI::cast("int", $b)->cdata;

   if ($aInt === $bInt) { return 0;}
   return ($aInt < $bInt) ? -1 : 1;
};

$ffi->qsort(
   FFI::addr($array),
   count($array),
   FFI::sizeof(FFI::type("int")),
   $cmp
);

var_dump($array);
```

Pointer to function in structure

- Structures can also contain pointer to some function
- It’s loosely similar to OOP.
- Often used as a building block for abstraction.

```
#include <stdio.h>

int sum(int x, int y)
{return x+y;}

typedef struct {
   int (*execute)(int, int);
} calculator;

int main()
{
   calculator Calc;

   Calc.execute=sum;
   int s = Calc.execute(34,80);

   printf("Sum is %d\n", s);

   return 0;
}
```

Memory management

- Biggest difference between PHP and C
- In PHP we don't care about memory but in C we have to.
- We have to decide where our variables will be stored.
- Simple example - returning array from function.

This won't work:

```
#include <stdio.h>

int[3] get_array()
{
   int arr[3] = {1,2,3};
   return arr;
}

int main() {
   int arr[3];
   arr=get_array();

   printf("First item %d", n[0]);

   return 0;
```

```
}
```

Compiler:

```
main.c:3:4: error: expected identifier or ‘(’ before ‘[’ token
    3 | int[3] get_array()
      |    ^
```

We cannot even return a pointer, so this is not proper solution:

```
#include <stdio.h>

int *get_array()
{
   int arr[3] = {1,2,3};
   return arr;
}

int main() {
   int *n;
   n=get_array();
   printf("First item %d", n[0]);
   return 0;
}
```

Compiler:

```
main.c:21:12: warning: function returns address of local variable
```

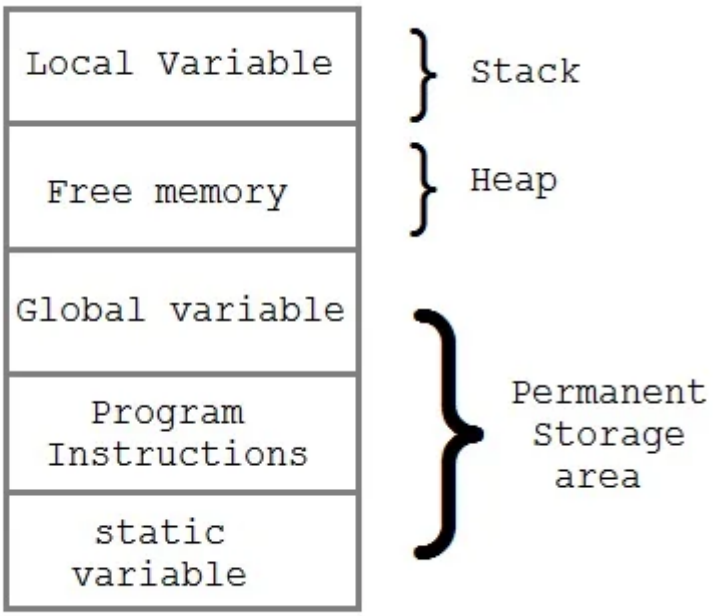
Program:

```
Process finished with exit code 139 (interrupted by signal 11: SIGSEGV)
```

Why does this not work?

- We are trying to return a local variable from function.
- Local variables must be copied => expansive operation for arrays.
- This is not supported in C.

How memory works in C



In C variables can be allocated in two places: heap and stack. In the case of function C allocate variables by default at stack. But stack is function exclusive and values will be removed, when the function ends its execution. On the other side a memory in heap can be accessed from everywhere.
In our previous case we actually try to return a memory address by using int *get_array() to no longer available memory address.

How to solve this?

- Using dynamic allocation.
- Using a static array.
- Using structure.

Dynamic allocation

- We will dynamically allocate memory on the heap.
- There is a special function for this malloc.

```
#include <stdio.h>
#include <malloc.h>

int* get_array() {
   int *arr= malloc(sizeof (int) * 5);
```

```
   arr[0] = 1;
   arr[1] = 2;
   arr[2] = 3;

   return arr;
}

int main() {
   int *n;
   n=get_array();

   printf("First item %d", n[0]);

   return 0;
}
```

But this is not everything - one thing is still missing:

- If we allocate memory we have to also release it.
- If we don't do this we will leak the memory.
- For this we have a function free.
- After that we should not access variables again.

```
#include <stdio.h>
#include <malloc.h>

int* get_array() {
   int *arr= malloc(sizeof (int) * 5);

   arr[0] = 1;
   arr[1] = 2;
   arr[2] = 3;

   return arr;
}

int main() {
   int *n;
   n=get_array();

   printf("First item %d", n[0]);
   free(n);

   return 0;
}
```

- This is a so-called dangling pointer.
- We don’t have any guarantee what is there.

```
int main() {
   int *n;
   n=get_array();

   printf("First item %d", n[0]);
   free(n);

   printf("First item %d", n[0]);

   return 0;
}
```

Header files

Header file contains C function declarations and macros.

Two usages:

- All called functions in C must be declared in advance.
- They describe functions and data structures which can be called from library.

Preprocessor

Preprocessor is a text substitution tool.

```
#include <stdio.h>

#if !defined (MESSAGE)
#define MESSAGE "You wish!"
#endif

int main(void) {
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

Preprocessor - it can be extremely powerful, we can for example create a foreach function in:

```
#include <stdio.h>
```

```
#define foreach(item, array) \
   for(int keep=1, \
           count=0,\
           size=sizeof (array)/sizeof *(array); \
       keep && count != size; \
       keep = !keep, count++) \
     for(item = (array)+count; keep; keep = !keep)

int main() {
   int a[] = { 1, 2, 3 };
   int sum = 0;
   foreach(int const* c, a)
           sum += *c;
   printf("sum = %d\n", sum);
}
```

Important tools/packages

All important tools are part of the provided docker image, their installation is described in Dockerfile.

Tools for compiling PHP

- C compiler, usually GCC on Linux, for debian based OS it can be installed as a part of **build-essential**
- Build tools:
  - autoconf
  - automake
- PHP build dependencies, this can depend on extension that we want to compile, common mandatory dependencies:
  - libzip-dev
  - libffi-dev
  - libssl-dev
  - libxml2-dev
  - libsqlite3-dev
  - libonig-dev
  - bison
  - re2c
- Debug tools:
  - gdb
  - valgrind

Build PHP in debug mode

```
git clone https://github.com/php/php-src.git
cd php-src
git checkout php-8.2.5
./buildconf --force
./configure --enable-debug --enable-mbstring --with-zip --with-zlib
--disable-phpdbg --disable-phpdbg-webhelper --enable-opcache --with-ffi
make -j4
make install
```

PHP FFI

Obtain an instance of FFI

- FFI::load creates a new instance of FFI object.
- The first argument points to a file with headers.
- Path to library must be provided inside the header file.

```
$ffi = FFI::load('duckdb-ffi.h');
```

```
#define FFI_LIB "./libduckdb.so"
```

Preprocess headers

- FFI does not support preprocessor directives.
- We let the preprocessor resolve variables for us.
- Before that we need to make a few changes.

inside duckdb.h remove or comment out these lines:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
```

- FFI does not support preprocessor directives.
- We let the preprocessor resolve variables for us.

```
echo '#define FFI_LIB "./libduckdb.so"' >> duckdb-ffi.h
cpp -P -C -D"__attribute__(ARGS)=" duckdb.h >> duckdb-ffi.h
```

Data types - C to PHP mapping

- Basic data types - automatic conversion.
- Extended data types: Can be created with new method on FFI instance
- User defined types:
    - Should be defined in advance (cdef method or in header file).

- Usually can be created by new method.

User defined types must be defined in advance:

```
$ffi = FFI::cdef(
   "typedef
        enum { DuckDBSuccess = 0, DuckDBError = 1 }
    duckdb_state;
    typedef void *duckdb_database;
   "
);
```

They can also be in the header file.

Creating complex data types

Complex data types can be created by the method new:

```
$database = $ffi->new("duckdb_database");
```

Enum values can be access directly on FFI instance:

```
if ($result === $ffi->DuckDBError) {
```

PHP vs C - pointers

- Arrays/strings in C are always passed as pointers.
- Sometimes structures are passed as pointers.

```
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
```

- In FFI pointer can be obtained only to C data structure.
- A pointer can be obtained by FFI::addr

```
$result = $duckDbFFI->duckdb_open(null, FFI::addr($database));
```

Complex data types - structures

Every structure must be defined by method FFI::cdef

```
$ffi = FFI::cdef(
   "typedef struct {
       idx_t __deprecated_column_count;
       idx_t __deprecated_row_count;
       idx_t __deprecated_rows_changed;
       duckdb_column *__deprecated_columns;
       char *__deprecated_error_message;
       void *internal_data;
   } duckdb_result;"
);
```

They can also be in the header file.

Structure variables can be accessed in the same way as an object property in regular PHP.

```
echo "Number of columns: ". $queryResult->__deprecated_column_count."\n";
```

Data type conversions - strings

- Data type conversion is partial.
- Parameters are converted, but **return values are not.**

```
$value = $duckDbFFI->duckdb_value_varchar(
   FFI::addr($queryResult), $column, $row
);

echo FFI::string($value);
```

Memory management

- FFI is partially managing the memory for us.

- Complex data types created by FFI new method are by default managed by PHP (they are owned).

But this is sometimes not enough.

Areas in which we have to be cautious:

- C pointers obtained by the method FFI::addr(...),
- individual elements of C arrays and structures
- and most data structures returned by C functions.

C pointers in FFI

- C pointers are non-owned => “dangling pointer”.
- Dangling pointer doesn’t point to a valid variable.
- This happens, when the referenced variable is destroyed, but the pointer still points to its memory location.

Extended data types returned by C functions

- When you deal with pointers you should ask yourself:

Who allocates and deallocates memory?

- Usually, it’s FFI, but sometimes it’s the called function.
- Sometimes we simply don’t know how much memory it needs in advance.

Function which allocates memory

- Clean up function must be called, we will find out this in documentation.
- After that, the variable is no longer usable!

```
$rawValue = $duckDbFFI->duckdb_value_varchar(
   FFI::addr($queryResult),
   $column,
   $row
);
$duckDbFFI->duckdb_free($rawValue);
```

Debug

Memory leaks

Valgrind can be used for leak detection.

```
valgrind --leak-check=full php test.php
```

This is the same as for any other C/C++ program.

Segfaults

```
gdb --args php test.php
source <php src folder>.gdbinit
```

- GDB can be used for code debugging.
- A usage of debug build of PHP is must.

GDB Commands

```
run
backtrace
dump_bt executor_globals.current_execute_data
quit
```

FFI in OOP fashion

Hide implementation in objects without exposing FFI.

```
<?php

$db         = new Kambo\DuckDB\Database();
$connection = new Kambo\DuckDB\Connection($db);
$connection->query("CREATE TABLE integers(i INTEGER, j INTEGER);");
$connection->query(
    "INSERT INTO integers VALUES (30, 40), (50, 60), (70, NULL);"
);
$result = $connection->query("SELECT * FROM integers");
```

```
var_dump($result->toArray());
```

Manage life cycle of FFI structure with destructors.

```
class Database
{
   private $database;

   final public function __construct(
       ?string $name=null
   ) {
       $FFI = DuckDBFFI::getInstance();
       $this->database = $FFI->new("duckdb_database");

       $result = $FFI->duckdb_open(
           $name,
           \FFI::addr($this->database)
       );
   }

   final public function __destruct() {
       DuckDBFFI::getInstance()->duckdb_close(
           \FFI::addr($this->database)
       );
   }
}
```

Further references for FFI - articles

PHPmagazin, 1/2021

- PHP FFI and What It Can Do for You
- https://devm.io/php/php-ffi-and-what-it-can-do-for-you
- https://entwickler.de/php/php-ffi-anwendung-und-funktionsweise

Php[architect], 3/2023

- A Guide to Practical Usage of PHP FFI.
- https://www.phparch.com/article/a-guide-to-practical-usage-of-php-ffi/

PHP extension

Introduction

Two types of extension:

- Zend extension (Xdebug, Taint)
- PHP extensions (MongoDB)

Compilation mode:

- statically compiled extensions
- dynamically loaded extensions.

Extension are usually written in C:

- There used to be an attempt to provide a CPP toolchain.
- Actually can be written in any language that can compile library with C ABI (eg.: rust)

Resources

- Writing PHP Extensions: https://www.zend.com/resources/writing-php-extensions
- Internals book is excellent resource: https://www.phpinternalsbook.com/index.html
- Existing extensions in php src from ext folder.

Prerequisites

- PHP source code.
- C compiler and all libraries need to compile PHP.
- For debugging - GDB and Valgrind.

It’s preferable and easier to do this under Linux.

Generate skeleton

Skeleton generator in PHP source code:
Generates simple fully functional extensions with tests and build scripts.

```
php /data/php-src/ext/ext_skel.php --ext testExt --dir .
```

Compilation

Compile and execute our first extension:

```
phpize
./configure
make
make test
```

Note - you can use: export NO_INTERACTION=1 if you want to turn off report to qa-php

```
export NO_INTERACTION=1
```

PHP lifecycle

- module startup step - MINIT
- the module shutdown step - MSHUTDOWN
- request startup step - RINIT
- request shutdown step - RSHUTDOWN

MINIT - module init

It’s used for permanent items such as:

- INI settings.
- Classes and exceptions
- Constants
- Global data (eg.: MYSQLi permanent connections)

MSHUTDOWN - module shutdown

- Opposite of MINIT
- No need to clean class, exception, constant definitions.
- But ini settings must be destroyed!
- Global data should be cleaned.

RINIT, RSHUTDOWN - request

INIT and SHUTDOWN for request

- Usually used for setting specific global states libraries.
- Also used for cleaning after these libraries.

Heart of extension - zend_module_entry

Heart of every PHP extension is C structure `zend_module_entry`

```
zend_module_entry duckdbext_module_entry = {
       STANDARD_MODULE_HEADER,     /* Hides internal information */
       "duckdbext",                /* Extension name */
       NULL,                       /* zend_function_entry */
       PHP_MINIT(duckdbext),       /* PHP_MINIT - Module initialization */
       NULL,                       /* PHP_MSHUTDOWN - Module shutdown */
       PHP_RINIT(duckdbext),       /* PHP_RINIT - Request init */
       NULL,                       /* PHP_RSHUTDOWN - Request shutdown */
       PHP_MINFO(duckdbext),       /* PHP_MINFO - Module info */
       PHP_DUCKDBEXT_VERSION,      /* Version */
       STANDARD_MODULE_PROPERTIES
};
```

Zend_module_entry uses pointers to functions a lot. All of this is done together with PHP specific macros.

Define API for our extension

```
<?php

$db         = new DuckDB\Database();
$connection = new DuckDB\Connection($db);
$connection->query("CREATE TABLE integers(i INTEGER, j INTEGER);");
$connection->query(
    "INSERT INTO integers VALUES (30, 40), (50, 60), (70, NULL);"
);
$result = $connection->query("SELECT * FROM integers");

var_dump($result->toArray());
```

Embedding C Data into PHP Objects

We have to merge together datža representing the class and duckdb connection. What are we trying to achieve in PHP code:

```
namespace DuckDB;
```

```
class Connection
{
   // FFI C data structure “duckdb_connection”
   private $duckdb_connection;
   // regular php properties and methods
    public function __construct(Database $database) {}
}
```

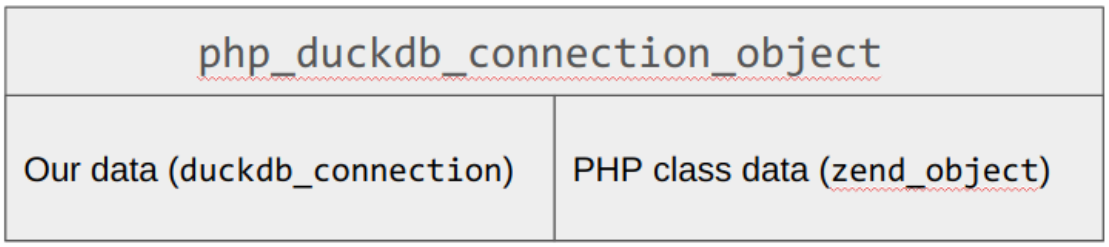
- zend_object represents class DuckDB\Connection
- C data represents duckdb_connection

We are forced to store C data inside PHP objects. This is not easy as:

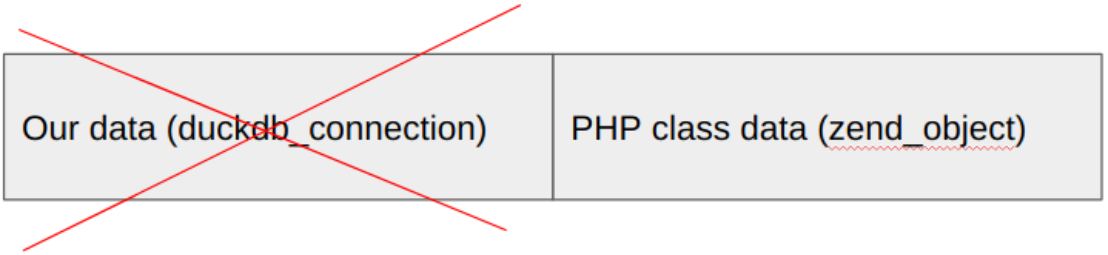
- zend_object needs special allocation.
- PHP knows only zend_object
- C data and zend_object must be allocated together:
- C data above the pointed address and zend_object below
- zend_object_alloc - allocates a memory block for object with all the properties
- It must also take into account the size of additional data.

```
typedef struct _php_duckdb_connection_object  {
   duckdb_connection connection;
   zend_object zo;
} php_duckdb_connection_object;
```

We are forced to store C data inside PHP objects.

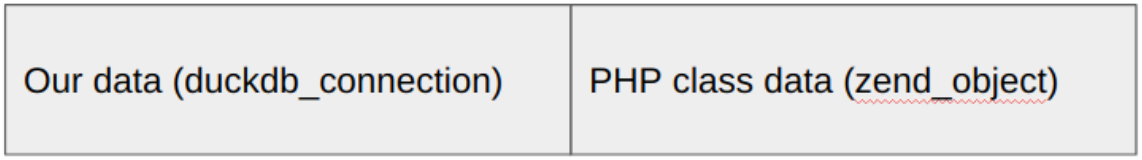


How does our class look to the PHP engine?

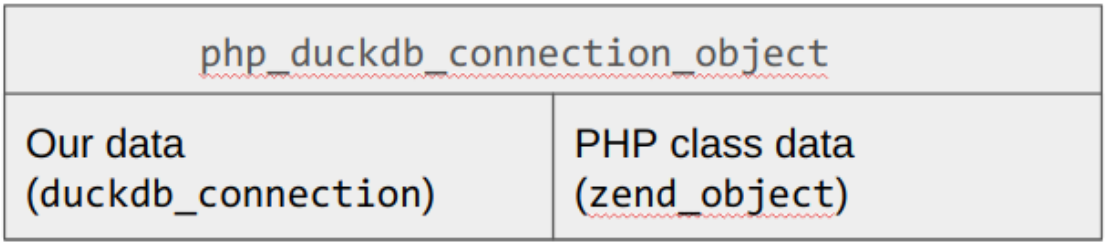


- PHP does not know about our data
- It recognize and can directly access only zend_object

We will use the memory address of zend_object and position of zend_object in php_duckdb_connection_object to calculate the proper address.



Thanks to XtOffsetOf we can know at which position zend_object starts in php_duckdb_connection_object.



- zend_object is in memory at address 100
- position of zend_object in php_duckdb_connection_object is 15
- Address of php_duckdb_connection_object is 100-15=85
- Pointer arithmetic allows us to access that structure.

- This will ensure object extraction if we want to use it latter:

```
connection_obj = Z_DUCK_CONNECTION_P(object);
#define Z_DUCK_CONNECTION_P(zv) (
     (php_duckdb_connection_object*)
     (
           (char*)(Z_OBJ_P(zv)) - XtOffsetOf(php_duckdb_connection_object,
zo)
     )
)
```

This is the reason why we have to override the object handlers:

```
duckdb_connection_handlers.offset =
XtOffsetOf(php_duckdb_db_object, zo);
duckdb_connection_handlers.clone_obj = NULL;
duckdb_connection_handlers.free_obj  = php_database_object_free;
```

Function definition

In extension functions are defined by PHP_METHOD. PHP_METHOD is macro not regular C function, it will expand before compilation:

```
PHP_METHOD(ClassName, methodName)

/* expands to */

void zim_ClassName_methodName(INTERNAL_FUNCTION_PARAMETERS)

/* expands to */

void    zim_ClassName_methodName(zend_execute_data    *execute_data,    zval
*return_value)
```

Zval a variable definition

zval is a basic data structure that allows PHP to hold multiple data types in one variable.

- It’s also used internally in extension development.
- We don’t need any special actions to destroy* them.

- Useful macros, eg.: ZVAL_STRING (allocates string), ZEND_THIS ($this)

```
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 - memory model

- Zend Memory Manager (ZendMM)
- Two allocation modes:
    - Request-bound dynamic allocations.
    - Permanent dynamic allocations.
- Extension almost exclusively use Request-bound
- Functions are same as in regular C, they are prefixed by e
  Eg.: emalloc(size_t) and efree(void *).
- They should be used only during request.
- ZendMM provide a basic leak detection.

Border cases

What if… …Somebody try to create an instance of a Result object without calling a constructor?

```
/**
 * Represents a query result
 * @not-serializable
```

```
 */
class Result {
   private function __construct() {}
   /**
    *
    * @return array
    */
   public function toArray() : array {}
}

$class = new ReflectionClass(DuckDB\Result::class);
$class->newInstanceWithoutConstructor();
```

We will check the newInstanceWithoutConstructor implementation in php-src!

```
/* {{{ Returns an instance of this class without invoking its constructor
*/
ZEND_METHOD(ReflectionClass, newInstanceWithoutConstructor)
{
...
```

Disable newInstanceWithoutConstructor
It’s possible to set the class to final.

```
duckdb_result_ce->ce_flags |= ZEND_ACC_FINAL|ZEND_ACC_NOT_SERIALIZABLE;
```

...or we set the initialised flag.

Debug

Valgrind
Tool for finding memory leaks

```
valgrind --leak-check=full php -d
extension_dir=/home/dev/DuckDBFFI/duckdbext/modules/ -d
extension=duckdbext.so tests/002.php
```

GDB debugger

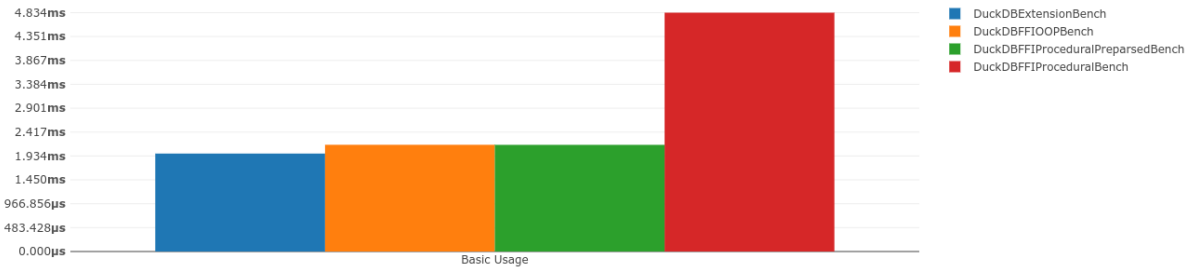
Useful when something goes wrong.

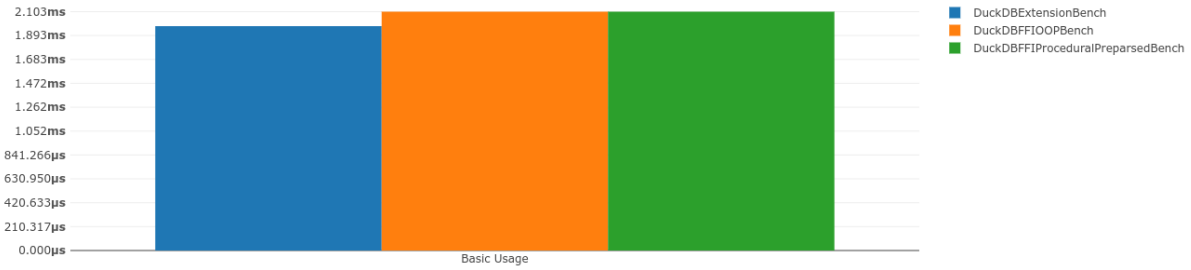
GDB commands:

backtrace

```
gdb --args php -d extension_dir=/home/dev/DuckDBFFI/duckdbext/modules/ -d extension=duckdbext.so tests/002.php
```

Performance





Performance - numbers

- FFI is roughly 5.87% slower then extension.
- FFI in OOP and procedural style are roughly similar.

- With definition parsing it’s 83.64% slower.
- Parsing of C definitions is fixed overhead.

Comparison

Comparison - common points

- We should have a decent knowledge of C.
- C is unforgiving, we can shoot ourselves in the foot.
- Cumbersome management of library dependency.
- For advance of debugging we need to use C tools (valgrind, GDB).
- Usage of FFI or extension can be insecure.

FFI

- No need to compile anything.
- Easier cross platform development.
- For some use cases performance is good enough.
- Easier to use => false sense of mastery.
- Not so powerful, some things are not fully functional.

Extension

- Extensions are faster.
- We have to set up a toolchain and compile PHP.
- Extra step of extension compilation.
- We should know how PHP works internally.
- A lot of boilerplate C code.

Which one is better?

- If performance is must have, nothing beat extensions.
- Extension can draw clear boundary.

X

- For one shot things choose FFI.
- FFI is also good for prototyping => define API, explore library, measure performance and then decide.

Code listings

1. Basic DuckDB example in C

```c
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include "duckdb.h"

int main() {
    // SLIDE explain data types
    duckdb_database db;
    duckdb_connection con;

    // SLIDE explain pointers + enums
    if (duckdb_open(NULL, &db) == DuckDBError) {
        duckdb_disconnect(&con);
        duckdb_close(&db);
        exit(1);
    }

    if (duckdb_connect(db, &con) == DuckDBError) {
        duckdb_disconnect(&con);
        duckdb_close(&db);
        exit(1);
    }

    // run queries...
    duckdb_state state;
    duckdb_result result;

    // create a table
    state = duckdb_query(
        Con,
        "CREATE TABLE integers(i INTEGER, j INTEGER);",
        NULL
    );
    if (state == DuckDBError) {
        duckdb_destroy_result(&result);
        duckdb_disconnect(&con);
        duckdb_close(&db);
        exit(1);
    }
```

```c
// insert three rows into the table
state = duckdb_query(
    Con,
    "INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL);",
    NULL
);
if (state == DuckDBError) {
    duckdb_destroy_result(&result);
    duckdb_disconnect(&con);
    duckdb_close(&db);
    exit(1);
}

// query rows again
state = duckdb_query(con, "SELECT * FROM integers", &result);
if (state == DuckDBError) {
    printf("%s", duckdb_result_error(&result));
    duckdb_destroy_result(&result);
    duckdb_disconnect(&con);
    duckdb_close(&db);
    exit(1);
}

// SLIDE explain structures
printf("Number of columns: %ld", result->__deprecated_column_count);

  idx_t row_count = duckdb_row_count(&result);
  idx_t column_count = duckdb_column_count(&result);

  // print the data of the result
  for (size_t row_idx = 0; row_idx < row_count; row_idx++) {
        for (size_t col_idx = 0; col_idx < column_count; col_idx++){
            char *val = duckdb_value_varchar(
                &result,
                Col_idx,
                Row_idx
            );
            printf("%s ", val);
            // SLIDE - memory management PHP vs C
            duckdb_free(val);
        }
        printf("\n");
  }

// destroy the result after we are done with it
duckdb_destroy_result(&result);
```

```
    // cleanup
    duckdb_disconnect(&con);
    duckdb_close(&db);
}
```

2. Basic DuckDB example in PHP

```php
<?php

$duckDbFFI = FFI::load('duckdb-ffi.h');

$database   = $duckDbFFI->new("duckdb_database");
$connection = $duckDbFFI->new("duckdb_connection");

$result = $duckDbFFI->duckdb_open(null, FFI::addr($database));

if ($result === $duckDbFFI->DuckDBError) {
    $duckDbFFI->duckdb_disconnect(FFI::addr($connection));
    $duckDbFFI->duckdb_close(FFI::addr($database));
    throw new Exception('Cannot open database');
}

$result = $duckDbFFI->duckdb_connect($database, FFI::addr($connection));
if ($result === $duckDbFFI->DuckDBError) {
    $duckDbFFI->duckdb_disconnect(FFI::addr($connection));
    $duckDbFFI->duckdb_close(FFI::addr($database));
    throw new Exception('Cannot connect to database');
}

$result = $duckDbFFI->duckdb_query(
    $connection,
    'CREATE TABLE integers(i INTEGER, j INTEGER);',
    null
);
if ($result === $duckDbFFI->DuckDBError) {
    $duckDbFFI->duckdb_disconnect(FFI::addr($connection));
    $duckDbFFI->duckdb_close(FFI::addr($database));
    throw new Exception('Cannot execute query');
}

$result = $duckDbFFI->duckdb_query(
    $connection,
    'INSERT INTO integers VALUES (3,4), (5,6), (7, NULL) ',
    null
);
if ($result === $duckDbFFI->DuckDBError) {
    $duckDbFFI->duckdb_disconnect(FFI::addr($connection));
    $duckDbFFI->duckdb_close(FFI::addr($database));
    throw new Exception('Cannot execute query');
}
```

```
$queryResult = $duckDbFFI->new('duckdb_result');

$result = $duckDbFFI->duckdb_query(
    $connection,
    'SELECT * FROM integers; ',
    FFI::addr($queryResult)
);

if ($result === $duckDbFFI->DuckDBError) {
    $error = "Error in query:
".$duckDbFFI->duckdb_result_error(FFI::addr($queryResult));

    $duckDbFFI->duckdb_destroy_result(FFI::addr($queryResult));
    $duckDbFFI->duckdb_disconnect(FFI::addr($connection));
    $duckDbFFI->duckdb_close(FFI::addr($database));
    throw new Exception($error);
}

echo "Number of columns: ".$queryResult->__deprecated_column_count."\n";

$rowCount = $duckDbFFI->duckdb_row_count(FFI::addr($queryResult));
$columnCount = $duckDbFFI->duckdb_column_count(FFI::addr($queryResult));

for ($row = 0; $row < $rowCount; $row++) {
    for ($column = 0; $column < $columnCount; $column++) {
        $value = $duckDbFFI->duckdb_value_varchar(
            FFI::addr($queryResult),
            $column,
            $row
        );
        echo ($value !== null ? FFI::string($value)  : '')." ";
        $duckDbFFI->duckdb_free($value);
    }

    echo "\n";
}

$duckDbFFI->duckdb_destroy_result(FFI::addr($queryResult));
$duckDbFFI->duckdb_disconnect(FFI::addr($connection));
$duckDbFFI->duckdb_close(FFI::addr($database));
```

3. DuckDB extension in C

```c
/* duckdb extension for PHP */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "ext/standard/info.h"
#include "zend_exceptions.h"
#include "php_duckdbext.h"
#include "duckdbext_arginfo.h"
#include "duckdb.h"
#include "php_duckdbext_structs.h"
#include "ext/spl/spl_exceptions.h"
#include "zend_interfaces.h"

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

static zend_class_entry *duckdb_database_ce = NULL;
static zend_class_entry *duckdb_connect_ce = NULL;
static zend_class_entry *duckdb_statementexception_ce = NULL;
static zend_class_entry *duckdb_result_ce = NULL;

static zend_object_handlers duckdb_database_handlers;
static zend_object_handlers duckdb_connection_object_handlers;
static zend_object_handlers duckdb_result_object_handlers;

static zend_object *php_database_object_new(zend_class_entry
*class_type)
{
    php_duckdb_db_object *intern;

    /* Allocate memory for it */
    intern = zend_object_alloc(sizeof(php_duckdb_db_object),
class_type);

    zend_object_std_init(&intern->zo, class_type);
    object_properties_init(&intern->zo, class_type);
```

```
    intern->zo.handlers = &duckdb_database_handlers;

    return &intern->zo;
}

static void php_database_object_free(zend_object *object)
{
    php_duckdb_db_object *intern = php_duckdb_database_from_obj(object);

    if (!intern) {
        return;
    }

    if (intern->db) {
        duckdb_close(&intern->db);
    }

    zend_object_std_dtor(&intern->zo);
}

PHP_METHOD(Database, __construct) {
    char *db_path = NULL;
    size_t db_path_len = 0;

    php_duckdb_db_object *db_obj;
    zval *object = ZEND_THIS;

    db_obj = Z_DUCKDATABASE_P(object);

    ZEND_PARSE_PARAMETERS_START(0, 1)
        Z_PARAM_OPTIONAL
        Z_PARAM_STRING(db_path, db_path_len)
    ZEND_PARSE_PARAMETERS_END();

    if (duckdb_open(db_path, &(db_obj->db)) == DuckDBError) {
        zend_throw_exception(zend_ce_exception, "Error during
initialization", 0);
        RETURN_THROWS();
    }
}

static zend_object *php_connection_object_new(zend_class_entry
*class_type)
{
    php_duckdb_connection_object *intern;
```

```
    /* Allocate memory for it */
    intern = zend_object_alloc(sizeof(php_duckdb_connection_object),
class_type);

    zend_object_std_init(&intern->zo, class_type);
    object_properties_init(&intern->zo, class_type);

    intern->zo.handlers = &duckdb_connection_object_handlers;

    return &intern->zo;
}

static void php_connection_object_free(zend_object *object)
{
    php_duckdb_connection_object *intern =
php_duckdb_connection_from_obj(object);

    if (!intern) {
        return;
    }

    if (intern->connection) {
        duckdb_disconnect(&intern->connection);
    }

    zend_object_std_dtor(&intern->zo);
}

PHP_METHOD(Connection, __construct) {
    php_duckdb_db_object *db_obj;
    zval *db_zval;
    zval *object = ZEND_THIS;
    php_duckdb_connection_object *connection_obj;

    connection_obj = Z_DUCK_CONNECTION_P(object);

    ZEND_PARSE_PARAMETERS_START(1, 1)
        Z_PARAM_OBJECT_OF_CLASS(db_zval, duckdb_database_ce)
    ZEND_PARSE_PARAMETERS_END();

    db_obj = Z_DUCKDATABASE_P(db_zval);

    if (duckdb_connect(db_obj->db, &(connection_obj->connection)) ==
DuckDBError) {
        zend_throw_exception(zend_ce_exception, "Error during
```

```
initialization", 0);
        RETURN_THROWS();
    }
}

static zend_object *php_result_object_new(zend_class_entry *class_type)
{
    php_duckdb_result_object *intern;

    /* Allocate memory for it */
    intern = zend_object_alloc(sizeof(php_duckdb_result_object),
class_type);

    zend_object_std_init(&intern->zo, class_type);
    object_properties_init(&intern->zo, class_type);

    intern->zo.handlers = &duckdb_result_object_handlers;

    return &intern->zo;
}

static void php_result_object_free(zend_object *object)
{
    php_duckdb_result_object *intern =
php_duckdb_result_from_obj(object);

    if (!intern) {
        return;
    }

    if (intern->result) {
        duckdb_destroy_result(intern->result);
    }

    efree(intern->result);

    zend_object_std_dtor(&intern->zo);
}

PHP_METHOD(Connection, query)
{
    char *query      = NULL;
    size_t query_len = 0;

    zval *object = ZEND_THIS;
    php_duckdb_connection_object *connection_obj;
```

```
    php_duckdb_result_object *result_obj;

    duckdb_result *inner_result;

    connection_obj = Z_DUCK_CONNECTION_P(object);

    ZEND_PARSE_PARAMETERS_START(1, 1)
        Z_PARAM_STRING(query, query_len)
    ZEND_PARSE_PARAMETERS_END();

    inner_result = emalloc(sizeof(duckdb_result));

    object_init_ex(return_value, duckdb_result_ce);
    result_obj = Z_DUCK_RESULT_P(return_value);
    result_obj->result = inner_result;

    if (duckdb_query(connection_obj->connection, query, inner_result) ==
DuckDBError) {
        zend_throw_exception_ex(duckdb_statementexception_ce, 0, "%s",
duckdb_result_error(inner_result));

        RETURN_THROWS();
    }
}


PHP_METHOD(Result, __construct)
{

}

void duckdb_value_to_zval(duckdb_result *result, idx_t row, idx_t
column, zval *data)
{
    char *val_s;
    val_s = duckdb_value_varchar(result, column, row);

    if (val_s == NULL) {
        ZVAL_NULL(data);
    } else {
        ZVAL_STRING(data, val_s);
        duckdb_free(val_s);
    }
}

PHP_METHOD(Result, toArray)
```

```
{
    zval *object = ZEND_THIS;
    php_duckdb_result_object *result_obj;
    result_obj = Z_DUCK_RESULT_P(object);
    array_init(return_value);

    idx_t row_count = duckdb_row_count(result_obj->result);
    idx_t column_count = duckdb_column_count(result_obj->result);

    for (idx_t row_idx = 0; row_idx < row_count; row_idx++) {
        zval row;
        array_init(&row);

        for (idx_t col_idx = 0; col_idx < column_count; col_idx++) {
            zval data;
            duckdb_value_to_zval(result_obj->result, row_idx, col_idx,
&data);
            add_index_zval(&row, col_idx, &data);
        }

        add_index_zval(return_value, row_idx, &row);
    }
}

/* {{{ PHP_RINIT_FUNCTION */
PHP_RINIT_FUNCTION(duckdbext)
{
    #if defined(ZTS) && defined(COMPILE_DL_DUCKDB)
        ZEND_TSRMLS_CACHE_UPDATE();
    #endif

    return SUCCESS;
}
/* }}} */

/* {{{ PHP_MINFO_FUNCTION */
PHP_MINFO_FUNCTION(duckdbext)
{
    php_info_print_table_start();
    php_info_print_table_header(2, "duckdb support", "enabled");
    php_info_print_table_end();
}
/* }}} */

/* {{{ PHP_MINIT_FUNCTION */
PHP_MINIT_FUNCTION(duckdbext)
```

```
{
    memcpy(&duckdb_database_handlers, &std_object_handlers,
sizeof(zend_object_handlers));
    memcpy(&duckdb_connection_object_handlers, &std_object_handlers,
sizeof(zend_object_handlers));
    memcpy(&duckdb_result_object_handlers, &std_object_handlers,
sizeof(zend_object_handlers));

    zend_class_entry duckdb_database_ce_local;
    INIT_NS_CLASS_ENTRY(duckdb_database_ce_local, "DuckDB","Database",
class_Database_methods);
    duckdb_database_ce_local.create_object = php_database_object_new;
    duckdb_database_handlers.offset = XtOffsetOf(php_duckdb_db_object,
zo);
    duckdb_database_handlers.clone_obj = NULL;
    duckdb_database_handlers.free_obj = php_database_object_free;
    duckdb_database_ce =
zend_register_internal_class(&duckdb_database_ce_local);
    duckdb_database_ce->ce_flags |= ZEND_ACC_NOT_SERIALIZABLE;

    zend_class_entry duckdb_connect_ce_local;
    INIT_NS_CLASS_ENTRY(duckdb_connect_ce_local, "DuckDB","Connection",
class_Connection_methods);
    duckdb_connect_ce_local.create_object = php_connection_object_new;
    duckdb_connection_object_handlers.offset =
XtOffsetOf(php_duckdb_connection_object, zo);
    duckdb_connection_object_handlers.clone_obj = NULL;
    duckdb_connection_object_handlers.free_obj =
php_connection_object_free;
    duckdb_connect_ce =
zend_register_internal_class(&duckdb_connect_ce_local);
    duckdb_connect_ce->ce_flags |= ZEND_ACC_NOT_SERIALIZABLE;

    zend_class_entry duckdb_statementexception_ce_local;
    INIT_NS_CLASS_ENTRY(duckdb_statementexception_ce_local,
"DuckDB","StatementException", NULL);
    duckdb_statementexception_ce =
zend_register_internal_class_ex(&duckdb_statementexception_ce_local,
spl_ce_RuntimeException);

    zend_class_entry duckdb_result_ce_local;
    INIT_NS_CLASS_ENTRY(duckdb_result_ce_local, "DuckDB","Result",
class_Result_methods);
    duckdb_result_ce_local.create_object = php_result_object_new;
    duckdb_result_object_handlers.offset =
XtOffsetOf(php_duckdb_result_object, zo);
```

```
    duckdb_result_object_handlers.clone_obj = NULL;
    duckdb_result_object_handlers.free_obj = php_result_object_free;
    duckdb_result_ce =
zend_register_internal_class(&duckdb_result_ce_local);
    duckdb_result_ce->ce_flags |=
ZEND_ACC_FINAL|ZEND_ACC_NOT_SERIALIZABLE;

    return SUCCESS;
}
/* }}} */

/* {{{ duckdbext_module_entry */
zend_module_entry duckdbext_module_entry = {
   STANDARD_MODULE_HEADER,
   "duckdbext",                  /* Extension name */
   NULL,                             /* zend_function_entry */
   PHP_MINIT(duckdbext),         /* PHP_MINIT - Module initialization */
   NULL,                        /* PHP_MSHUTDOWN - Module shutdown */
   PHP_RINIT(duckdbext),         /* PHP_RINIT - Request initialization */
   NULL,                        /* PHP_RSHUTDOWN - Request shutdown */
   PHP_MINFO(duckdbext),         /* PHP_MINFO - Module info */
   PHP_DUCKDBEXT_VERSION,        /* Version */
   STANDARD_MODULE_PROPERTIES
};
/* }}} */


#ifdef COMPILE_DL_DUCKDBEXT
# ifdef ZTS
ZEND_TSRMLS_CACHE_DEFINE()
# endif
ZEND_GET_MODULE(duckdbext)
#endif
```