PHP Extension Development with C++

Wrapping a C preprocessor API in C++

Florian Sowade

August 25, 2012

About Me

- Florian Sowade
- Head of embedded software development at crosscan GmbH
- Currently studying computer science in Dortmund
- C++ professional since about five years
- Member of PHP Usergroup Dortmund since about four years
- `@rioderelfte` on twitter

Overview

- PHP provides C (preprocessor) API to develop extensions
  - C API can be used directly from C++

Overview

- PHP provides C (preprocessor) API to develop extensions
  - C API can be used directly from C++

  $\Rightarrow$ We're done here

Overview

- PHP provides C (preprocessor) API to develop extensions
  - C API can be used directly from C++

  $\Rightarrow$ We're done here
- API can be wrapped in C++
  - Use C++ features (objects, exceptions, templates, ...)
  - Advanced techniques (RAII, TMP, ...)

Overview

- PHP provides C (preprocessor) API to develop extensions
  - C API can be used directly from C++

  $\Rightarrow$ We're done here
- API can be wrapped in C++
  - Use C++ features (objects, exceptions, templates, ...)
  - Advanced techniques (RAII, TMP, ...)
- Proof of concept wrapper library
  - C++11 (tested with gcc 4.7)
  - Incomplete (mainly defining functions)
  - Not tested
  - ...

Outline

C API

- Every PHP function has one corresponding C function
- Argument information to describe function parameters
  $\Rightarrow$ Reflection
- Function table containing information about every function
- Module entry containing information about the extension
- `get_module()` to retrieve module entry from extension

The test function: PHP

```php
<?php

function helloWorld($name, $flag) {
  if (!$flag) {
    return null;
  }

  return 'Hello World, ' . $name . '!!!';
}
```

The test function: C function

```
1  extern "C" ZEND_FUNCTION(helloWorld)
2  {
3    const char *nameData;
4    int nameLength;
5    long flag;
6    auto ret = zend_parse_parameters(
7      ZEND_NUM_ARGS() TSRMLS_CC,
8      "sb", &nameData, &nameLength, &flag
9    );
10   if (ret == FAILURE)
11     return;
12   if (!flag)
13     RETURN_NULL();
14   std::string name{nameData, nameLength};
15   std::string r{"Hello World, " + name + "!!!"};
16   RETURN_STRING(r.c_str(), 1);
17 }
```

The test function: C arg info

```
1  ZEND_BEGIN_ARG_INFO_EX(
2      arginfoHelloWorld,
3      0, 0,
4      2
5  )
6      ZEND_ARG_INFO(0, name)
7      ZEND_ARG_INFO(0, flag)
8  ZEND_END_ARG_INFO()
```

The test function: C function table

```
1  const zend_function_entry helloWorldFunctions[] =
2  {
3      PHP_FE(helloWorld, arginfoHelloWorld)
4      PHP_FE_END
5  };
```

The test function: C module entry

```
1  zend_module_entry helloWorldEntry =
2  {
3     STANDARD_MODULE_HEADER,
4     "helloWorld",
5     helloWorldFunctions,
6     0, 0, 0, 0, 0
7     "1.0",
8     STANDARD_MODULE_PROPERTIES
9  };
10
11 extern "C" const zend_module_entry *get_module()
12 {
13    return &helloWorldEntry;
14 }
```

Outline

Registering Functions

- Wraping arg info, function and module entry in classes
- Unify information at one place
- No macros

  $\Rightarrow$ Easyer to generate from higher level APIs
- Speaking API names (less positional arguments)
- Implementation has to rely on implementation details

  $\Rightarrow$ Can't be built using only the "public" macros

Registering Functions: example

```
1  ModuleEntry moduleEntry
2  {
3    "helloWorld", "1.0",
4    {
5      {
6        "helloWorld", helloWorld,
7        {
8          valueArgument("name"),
9          valueArgument("flag")
10       }
11     },
12     {
13       "foobar", foobar,
14       {
15         valueArgument("foo"),
16         optionalValueArgument("bar")
17       },
18       passRestByReference
19     }
20   }
21 };
```

Registering Functions: ArgInfo

```
1  struct ArgInfo
2  {
3    constexpr ArgInfo(bool optional /*, ...*/);
4    constexpr zend_arg_info argInfo();
5    constexpr bool isOptional();
6  };
7
8  struct ArgInfoSequence
9  {
10   template <class... Ts>
11     ArgInfoSequence(const Ts&... args);
12   const zend_arg_info *argInfo() const;
13   size_t size() const;
14   void passRestByReference(bool val);
15 };
16
17 template <size_t LEN>
18   constexpr ArgInfo valueArgument(
19     const char (&name)[LEN]
20   );
```

Registering Functions: Function Entry

```
1  struct FunctionEntry
2  {
3    template <class... FUNS>
4      FunctionEntry(
5        const char *name,
6        void (*handler)(/*...*/),
7        ArgInfoSequence argInfos,
8        const FUNS&... funs
9      );
10     zend_function_entry functionEntry() const;
11 };
12
13 struct FunctionEntrySequence
14 {
15   FunctionEntrySequence(
16     std::initializer_list<FunctionEntry> funs
17   );
18   // ...
19 };
20
21 void passRestByReference(FunctionEntry &entry);
```

Outline

Parsing Parameters

```
1  auto ret = zend_parse_parameters(
2    ZEND_NUM_ARGS() TSRMLS_CC,
3    "sb", &nameData, &nameLength, &flag
4  );
5  if (ret == FAILURE)
6    return;
```

Parsing Parameters

```
1  auto ret = zend_parse_parameters(
2    ZEND_NUM_ARGS() TSRMLS_CC,
3    "sb", &nameData, &nameLength, &flag
4  );
5  if (ret == FAILURE)
6    return;
```

- Duplication of types
- No type conversion (`bool`, `std::string`, ...)

Parsing Parameters

```
auto ret = zend_parse_parameters(
  ZEND_NUM_ARGS() TSRMLS_CC,
  "sb", &nameData, &nameLength, &flag
);
if (ret == FAILURE)
  return;
```

- Duplication of types
- No type conversion (`bool`, `std::string`, ...)

```
bool flag;
std::string name;
auto success = parseArguments(
  ZEND_NUM_ARGS() TSRMLS_CC,
  name, flag
)
if (!success)
  return;
```

Parsing Parameters: building the string

```
1  template <typename T>
2    struct Parameter;
3
4  template<>
5    struct Parameter<std::string>
6  {
7    static const char code = 's';
8    // ...
9  };
10
11 template <typename... Ts>
12   bool parseArguments(int ht TSRMLS_DC, Ts&... vs)
13 {
14   char paramCodes[] {Parameter<Ts>::code..., 0};
15   // ...
16 }
```

Parsing Parameters: parsing (1/2)

```
1  template <>
2    struct Parameter<std::string>
3  {
4    Parameter(std::string &str);
5    bool commit();
6    std::tuple<char**, int*> params();
7    // ...
8  };
9
10 template <typename... Ts>
11   bool doParse(int ht TSRMLS_DC, Ts&... vs)
12 {
13   auto params = std::tuple_cat(
14     std::make_tuple(ht TSRMLS_CC, paramCodes),
15     vs.params()...
16   );
17
18   return call(zend_parse_parameters, params) != FAILURE;
19 }
```

Parsing Parameters: parsing (2/2)

```
1  template <typename... Ts>
2    void commit(int ht TSRMLS_DC, Ts&... vs)
3  {
4    doNothing(vs.commit()...);
5  }
6
7  template <typename... Ts>
8    bool parseArguments(int ht TSRMLS_DC, Ts&... vs)
9  {
10   char paramCodes[] {Parameter<Ts>::code..., 0};
11
12   auto params = std::make_tuple(
13     ht TSRMLS_CC, paramCodes, Parameter<Ts>(vs)...
14   );
15
16   if (call(doParse<Parameter<Ts>...>, params)) {
17     call(commit<Parameter<Ts>...>, params);
18     return true;
19   }
20   return false;
21 }
```

Outline

Defining Functions

- For now I abstracted the PHP C API
- Now I will show something usefull based on that

Defining Functions: example

```
1  DEFINE_FUNCTION(
2      boost::optional<std::string>,
3      helloWorld,
4      ((std::string, name))((bool, flag))
5  ) {
6      if (!flag)
7          return {};
8
9      return "Hello World, " + name + "!!!";
10 }
11
12 ModuleEntry moduleEntry{
13     "cpptest", "0.1",
14     {
15         helloWorld
16     }
17 };
```

Defining Functions: `returnValue`

```
1  void returnValue(
2    INTERNAL_FUNCTION_PARAMETERS, const std::string &ret
3  ) {
4    RETVAL_STRINGL(ret.c_str(), ret.size(), 1);
5  }
6
7  template <typename T>
8    void returnValue(
9      INTERNAL_FUNCTION_PARAMETERS,
10     const boost::optional<T> ret
11   )
12 {
13   if (ret)
14     returnValue(INTERNAL_FUNCTION_PARAM_PASSTHRU, *ret);
15   else
16     RETVAL_NULL()
17 }
```

Defining Functions: expanded (1/2)

```
1  boost::optional<std::string> helloWorldImpl(
2    Parameter<std::string>::ParameterType name,
3    Parameter<bool>::ParameterType flag
4  );
5
6  extern "C"
7    void helloWorldCaller(INTERNAL_FUNCTION_PARAMETERS)
8  {
9    std::tuple<std::string, bool> params;
10
11   if (!parseArguments(ZEND_NUM_ARGS() TSRMLS_CC, params))
12     return;
13
14   returnValue(
15     INTERNAL_FUNCTION_PARAM_PASSTHRU,
16     call(helloWorldImpl, params)
17   );
18 }
```

Defining Functions: expanded (2/2)

```
1  FunctionEntry helloWorld{
2    "helloWorld",
3    helloWorldCaller,
4    {
5      valueArgument("name"),
6      valueArgument("flag")
7    }
8  };
9
10 boost::optional<std::string>
11   helloWorldImpl(
12     Parameter<std::string>::ParameterType name,
13     Parameter<bool>::ParameterType flag
14   )
15 {
16   if (!flag)
17     return {};
18
19   return "Hello World, " + name + "!!!";
20 }
```

Outline

ZValue

- C `struct` containing the data of a PHP variable
- Can reference eight data types: null, bool, long, double, string, resource, array and object
- Reference counted
  - $\Rightarrow$ RAII

RAII

- Resource Acquisition Is Initialization
- Manage resources by objects on the stack
- Free resources in the destructor

ZValue: overview

```
1  class ZValue {
2  public:
3    ZValue()
4    : zval_(0)
5    {
6      ALLOC_INIT_ZVAL(zval_);
7    }
8    ~ZValue() {
9      if (zval_) {
10       decRef();
11     }
12   }
13 private:
14   void addRef() {
15     Z_ADDREF_P(zval_);
16   }
17   void decRef() {
18     zval_ptr_dtor(&zval_);
19   }
20   _zval_struct *zval_;
21 };
```

ZValue: copy/move

```
1  ZValue(const ZValue &rhs)
2  : zval_(rhs.zval_)
3  {
4    addRef();
5  }
6
7  ZValue(ZValue &&rhs)
8  : zval_(rhs.zval_)
9  {
10   rhs.zval_ = 0;
11 }
12
13 ZValue &operator=(const ZValue &rhs) {
14   return *this = rhs.zval_;
15 }
16
17 ZValue &operator=(_zval_struct *rhs) {
18   ZVAL_ZVAL(zval_, rhs, true, false);
19
20   return *this;
21 }
```

ZValue: assign

```
1  template <typename T>
2    ZValue(T &&val)
3    : ZValue()
4  {
5    assign(std::forward<T>(val));
6  }
7  template <typename T>
8    ZValue &operator=(T &&val)
9  {
10   assign(std::forward<T>(val));
11   return *this;
12 }
13 private:
14 void assign(bool val) {
15   ZVAL_BOOL(zval_, val);
16 }
17
18 void assign(long val) {
19   ZVAL_LONG(zval_, val);
20 }
21 // ...
```