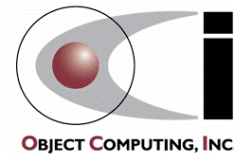


Ruby Plays Well With Others - Part 2

Ruby C Extensions

Mark Volkmann
mark@ociweb.com



Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Overview

- **Reasons to**
 - **invoke C from Ruby**
 - use C libraries from Ruby applications
 - performance
 - **invoke Ruby from C**
 - use Ruby libraries from C applications
- **Ruby can call C code in these ways**
 - **interpreter API**
 - since the Ruby interpreter is implemented in C, its API can be used
 - don't need a special API added for interacting with C like Java's JNI
 - **RubyInline**
 - supports mixing C code into Ruby code
 - **SWIG**
 - generates wrapper code for C functions in many languages including Ruby
 - **we'll focus on the interpreter API here**

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

2

mkmf (make makefile) Ruby Module

- Generates platform-specific Makefiles for compiling C extensions to Ruby
- Simple usage

- create a file containing the following, named `extconf.rb` by convention

```
require 'mkmf' ←  
extension_name = 'name'  
dir_config(extension_name) ←  
create_makefile(extension_name)
```

mkmf.rb is in
\$RUBY_HOME/lib

specifies nonstandard
directories where
include files and
libraries may be found

- use by running

```
ruby extconf.rb  
make
```

- generates

- `.so` under UNIX/Linux
- `.so` under Windows when building with Cygwin
- `.bundle` under Mac OS X

Can add conditional processing
using these Ruby functions:

- `check_sizeof`
- `disable_config`
- `enable_config`
- `find_executable`
- `find_header`
- `find_library`
- `have_func`
- `have_header`
- `have_library`
- `have_macro`
- `have_struct_member`
- `have_type`
- `have_var`
- `pkg_config`
- `with_config`

Ruby Constants and Types in C

- Boolean constants
 - `Qtrue` and `Qfalse`
- “No value” constant
 - `Qnil`
 - returned from C functions that are defined as Ruby functions that have no return value
- C struct types for some specific kinds of Ruby objects
 - `RBignum`, `RFloat`,
 - `RString`, `RRegexp`
 - `RStruct`, `RArray`, `RHash`
 - `RClass`, `RObject`
 - `RFile`
- C type for referring to arbitrary Ruby objects
 - `VALUE`
 - declared as an unsigned long in `ruby.h`
 - a pointer to one of the struct types listed above

declared
as structs
in `ruby.h`

Type Checks and Conversions

- Check data type
 - `TYPE(VALUE value)` - returns a constant value that identifies the type
 - `NIL_P(VALUE value)` - raises an exception if not nil
 - `FIXNUM_P(VALUE value)` - raises an exception if not a Fixnum
 - `Check_Type(VALUE value, int type)`
- raises an exception if not specified type
- Convert numeric type
 - `long FIX2INT(VALUE value)` - Ruby Fixnum to C long
 - `long NUM2INT(VALUE value)` - Ruby Numeric to C long
 - `double NUM2DBL(VALUE value)` - Ruby Numeric to C double
 - `VALUE INT2FIX(long i)` - C long to Ruby Fixnum
 - `VALUE INT2NUM(long i)` - C long to Ruby Fixnum or Bignum
- Macros that cast a `VALUE` to a pointer to a C struct that represents a Ruby object
 - `ROBJECT, RCLASS, RMODULE`
 - `RBIGNUM, RFLOAT, RSTRING, RREGEXP`
 - `RSTRUCT, RARRAY, RHASH`
 - `RDATA, RFILE`

Constants returned by `TYPE` are `T_NULL`, `T_OBJECT`, `T_CLASS`, `T_MODULE`, `T_FALSE`, `T_TRUE`, `T_FIXNUM`, `T_BIGNUM`, `T_FLOAT`, `T_SYMBOL`, `T_STRING`, `S_REGEXP`, `T_ARRAY`, `T_HASH`, `T_STRUCT`, `T_DATA`, `T_FILE`

Ruby `Fixnum` holds 4 byte integer values. Ruby `Numeric` holds any kind of numeric value including `Float`.

Extending Ruby from C

- Create new Ruby modules
 - `VALUE rb_define_module(const char* name);`
 - `VALUE rb_define_module_under(VALUE module, const char* name);`
- Create new Ruby classes
 - `VALUE rb_define_class(const char* name, VALUE super)`
 - `VALUE rb_define_class_under(VALUE module, const char* name, VALUE super)`
- Define functions / methods
 - `rb_define_global_function(const char* name, VALUE (*func)(), int argc)`
 - `rb_define_module_function(VALUE module, const char* name, VALUE (*func)(), int argc)`
 - `rb_define_method(VALUE class, const char* name, VALUE (*func)(), int argc)`

These functions are defined in `class.c` which defines many more than are covered here.

return value represents the created module or class

pointer to a C function that returns a VALUE

see `README.ext` for details on what happens when `argc` is -1 or -2

there are also functions to define private and singleton methods

Hello Example

- **Files**
 - `extconf.rb`
 - generates Makefile
 - `hello.c`
 - C code to be invoked from Ruby
 - `client.rb`
 - Ruby code that invokes C code
- **Steps to build and run**

```
ruby extconf.rb
make
ruby client.rb
```

Hello Example - hello.c

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

// These C functions will be associated with
// methods of a Ruby class on the next page.

static VALUE hello(VALUE self, VALUE arg) {
    char* name = RSTRING(arg)->ptr;
    printf("Hello %s!\n", name);
    return Qnil;
}

static VALUE goodbye(VALUE class) {
    printf("Later dude!\n");
    return Qnil;
}
```

returning Qnil since
there is no return value

static functions in C
are only visible to
other functions in
the same source file

RSTRING is a macro defined in
ruby.h that casts a VALUE to a
pointer to the underlying struct
that describes a Ruby String.
The ptr member points to its
char* value.

continued on next page

Hello Example - hello.c (Cont'd)

```
// This is called when the Ruby interpreter loads this C extension.
// The part after "Init_" is the name of the C extension specified
// in extconf.rb, not the name of the C source file.
void Init_hello() {
    // Create a Ruby module.
    VALUE myModule = rb_define_module("MyModule");

    // Create a Ruby class in this module.
    // rb_cObject is defined in ruby.h
    VALUE myClass =
        rb_define_class_under(myModule, "MyClass", rb_cObject);

    // Add an instance method to the Ruby class.
    int arg_count = 1;
    rb_define_method(myClass, "hello", hello, arg_count);

    // Add a class method to the Ruby class.
    arg_count = 0;
    rb_define_module_function(myClass, "goodbye", goodbye, arg_count);
}
```

superclass



Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

9

Hello Example - client.rb

```
require 'hello'
include MyModule # so MyClass doesn't need MyModule:: prefix

obj = MyClass.new # MyClass is defined in C
obj.hello('Mark') # calling an object method
MyClass.goodbye # calling a class method
```

Output

Hello Mark!
Later dude!

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

10

Ruby Strings in C

- Create

- `rb_str_new(const char* ptr, long len)`

- creates a Ruby `String` object, allocates `len` bytes for data, and if `ptr` isn't null, copies `len` bytes from `ptr` into it
 - example

```
VALUE ruby_string = rb_str_new2(NULL, 4);
ruby_string->ptr = "test";
```

- `rb_str_new2(const char* ptr)`

- creates a Ruby `String` object, allocates `strlen(ptr)` bytes for data, and copies the C string at `ptr` into it
 - `ptr` cannot be null
 - example

```
VALUE ruby_string = rb_str_new2("Hello World!");
```

These functions are defined in `string.c` which defines many more than are covered here.

Ruby Strings in C (Cont'd)

- Use

- example

```
int len = RSTRING(ruby_string)->len;
char* c_string = RSTRING(ruby_string)->ptr;
```

- Append

- `rb_str_cat(VALUE str, const char* ptr, long len)`

- concatenates `len` bytes from `ptr` onto `str`
 - example

```
char* c_string = "more";
rb_str_cat(ruby_string, c_string, strlen(c_string));
```

Gettin' Stringy With It!

```
#include <ruby.h>

static VALUE takeString(VALUE class, VALUE ruby_string) {
    // Create a new Ruby String object.
    VALUE s = rb_str_new2("Hello "); s is really a struct RString*

    // Concatenate C strings to the Ruby String.
    const char* c_string = RSTRING(ruby_string)->ptr;
    rb_str_cat(s, c_string, strlen(c_string));
    rb_str_cat(s, "!", 1);
    return s;
}

void Init_strings() {
    int arg_count = 1;
    rb_define_global_function("take_string", takeString, arg_count);
}
```

Ruby code

```
require 'strings'
puts take_string("Mark")
```

Output

```
Hello Mark!
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

13

Ruby Arrays in C

- **Create**
 - `rb_ary_new()`
 - creates an empty array (actually has a default size of 16)
 - `rb_ary_new2(long len)`
 - creates an array of a given size
 - `rb_ary_new3(long len, ...)`
 - creates an array of a given size and populates it
- **Set element values**
 - `rb_ary_store(VALUE ary, long index, VALUE value)`
 - grows array if necessary
 - `RARRAY(ary)->ptr[index] = value;`
 - can step off end of array
- **Get element values**
 - `rb_ary_entry(VALUE ary, long index)`
 - verifies that index is in bounds; negative indexes count from end
 - `VALUE value = RARRAY(ary)->ptr[index];`
 - can step off end of array

These functions are defined in array.c which defines many more than are covered here.

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

14

Ruby Arrays in C (Cont'd)

- Get length
 - `long len = RARRAY(ary)->len`
- Add an element
 - `rb_ary_push(VALUE ary, VALUE value)`
 - grows array if necessary and adds value to end
 - `rb_ary_shift(VALUE ary, VALUE value)`
 - shifts all elements forward and adds value to beginning
- Remove an element
 - `VALUE rb_ary_pop(VALUE ary)`
 - removes last element from array and returns it
 - `VALUE rb_ary_shift(VALUE ary)`
 - removes first element from array and returns it
- Find an element
 - `long rb_ary_index(VALUE ary, VALUE value)`
 - finds index of first occurrence; `Qnil` if not found
 - `long rb_ary_rindex(VALUE ary, VALUE value)`
 - finds index of last occurrence; `Qnil` if not found

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

15

Hooray for Arrays!

```
#include <ruby.h>
#include <stdbool.h>

static VALUE process(VALUE self, VALUE in_ary) {
    int len = RARRAY(in_ary)->len;
    VALUE* dataPtr = RARRAY(in_ary)->ptr;

    // Create new Ruby Array that is the same size as the one passed in.
    VALUE out_ary = rb_ary_new2(len);

    // Process each element in the input array
    // and place result in the corresponding element of the output array.
    int i, j, s_len;
    long v;
    for (i = 0; i < len; ++i) {
        VALUE value = dataPtr[i];
        int type = TYPE(value);
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

16

Hooray for Arrays! (Cont'd)

```
switch (type) {
  case T_STRING: // make uppercase
    s_len = RSTRING(value)->len;
    char* s = RSTRING(value)->ptr;
    for (j = 0; j < s_len; ++j) {
      s[j] = toupper(s[j]);
    }
    break;
  case T_FIXNUM: // square
    v = FIX2INT(value);
    value = INT2FIX(v * v);
    break;
  case T_TRUE:
  case T_FALSE: // flip
    b = FIX2INT(value);
    value = INT2FIX(!b);
    break;
} // of switch

rb_ary_store(out_ary, i, value);
}
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

17

Hooray for Arrays! (Cont'd)

```
return out_ary;
} // end of process function

// This is called when the Ruby interpreter loads this C extension.
void Init_arrays() {
  // Create a Ruby module.
  VALUE myModule = rb_define_module("MyModule");

  // Create a Ruby class.
  // rb_cObject is defined in ruby.h
  VALUE myClass =
    rb_define_class_under(myModule, "MyClass", rb_cObject);

  // Add a method to the Ruby class.
  int arg_count = 1;
  rb_define_method(myClass, "process", process, arg_count);
}
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

18

Hooray for Arrays! (Cont'd)

```
require 'arrays'

obj = MyModule::MyClass.new

array = ['Mark', 3, true]
puts obj.process(array)

array = ['Tami', 4, false]
puts obj.process(array)
```

```
Output
MARK
9
0
TAMI
16
1
```

Using Ruby From C

- Evaluate Ruby code

- VALUE rb_eval_string(const char* ruby_code)

- Create a Ruby object

- steps

```
ID class_id = rb_intern("class-name");
VALUE class = rb_const_get(rb_cObject, class_id);
VALUE obj = rb_class_new_instance(argc, argv, class);
```

For top-level classes,
use rb_cObject.
For classes in a module,
use a VALUE for the module.

- Invoke a method

- VALUE rb_funcall(
VALUE receiver, ID method_id, int argc, ...)
 - VALUE rb_funcall2(
VALUE receiver, ID method_id, int argc, VALUE* argv)

C array of VALUES to be
passed to initialize method

- example

```
VALUE ruby_string = rb_str_new2("some text");
ID method_id = rb_intern("upcase");
VALUE ruby_up_string = rb_funcall(ruby_string, method_id, 0);
```

passing parameters
in a single array

Errors and Exceptions

- **Raise**
 - `rb_raise(rb_eRuntimeError, const char* format_string, ...)`
 - raises a `RuntimeError` with the given message
 - `rb_raise(VALUE exception_object, const char* format_string, ...)`
 - raises the specified exception with the given message
- **Rescue**
 - `VALUE rb_rescue(...)`
 - invokes a specified function if any Ruby exception is raised in another function
- **Ensure**
 - `VALUE rb_ensure(...)`
 - ensures that a function is invoked regardless of whether another function raises a Ruby exception

Errors and Exceptions (Cont'd)

- **Warn**
 - `rb_warn(const char* format_string, ...)`
 - prints message
 - `rb_warning(const char* format_string, ...)`
 - only prints message if `$VERBOSE` is true
- **Terminate**
 - `rb_fatal(const char* format_string, ...)`
 - prints message, executes ensure blocks, skips exception handling, and raises a fatal error which terminates interpreter
 - `rb_bug(const char* format_string, ...)`
 - prints message, skips ensure blocks, skips exception handling, and raises a fatal error which terminates interpreter

Starting Ruby Interpreter From C

- Initialize interpreter
 - `ruby_init()`
- Allow interpreter to process command-line arguments
 - `ruby_options(int argc, char** argv)`
- Optionally specify a name for the “script” being run
 - `ruby_script(char* name)`
- Start execution
 - `ruby_run()`

Additional Features

- Get name of a Ruby class as a C string
 - `char* rb_class2name(VALUE class)`
- Determine if an object responds to a method
 - `int rb_respond_to(VALUE object, ID method_id)`
- Convert an ID to C string
 - `char* rb_id2name(ID id)`
- Define a constant
 - `rb_define_const(VALUE class, const char* name, VALUE value)`
 - `rb_define_global_const(const char* name, VALUE value)`
 - same as `rb_define_const(cKernel, name, value)`
- Share a global variable between Ruby and C
 - `rb_define_variable(const char* ruby_global_var_name, VALUE* c_variable)`
 - and related functions to define read-only, virtual and hooked variables

Additional Features (Cont'd)

- Get and set instance variables of a Ruby object
 - `VALUE rb_iv_get(VALUE object, const char* iv_name)` instance variable names must start with @
 - `rb_iv_set(VALUE object, const char* iv_name, VALUE value)` see example ahead
- Encapsulate a C struct as a Ruby object
 - `Data_Wrap_Struct(...)`
 - `Data_Make_Struct(...)`
 - `Data_Get_Struct(...)`
- Mix a Ruby module
 - into a Ruby class
 - `rb_include_module(VALUE class, VALUE module)`
 - into a specific Ruby object
 - `rb_extend_object(VALUE object, VALUE module)`

Additional Features (Cont'd)

- Pass a “block” to a function
 - `VALUE rb_iterate(...)`
- Invoke block passed in with a given parameter
 - `rb_yield(VALUE value)`

Pass/Return Ruby Objects - dealer.c

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

static VALUE dealerInit(VALUE self, VALUE name) {
    rb_iv_set(self, "@name", name);
}

static VALUE tradeCar(VALUE self, VALUE old_car) {
    // Print information about the Ruby Car object passed in.
    VALUE make = rb_iv_get(old_car, "@make");
    VALUE model = rb_iv_get(old_car, "@model");
    VALUE year = rb_iv_get(old_car, "@year");
    printf("tradeCar received %d %s %s\n",
        FIX2INT(year),
        RSTRING(make)->ptr,
        RSTRING(model)->ptr);

    // Modify one of its instance variables just to show we can.
    rb_iv_set(old_car, "@year", INT2FIX(2007));
}
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

27

Pass/Return Ruby Objects - dealer.c (Cont'd)

```
// Create a new Ruby Car object.
make = rb_str_new2("BMW");
model = rb_str_new2("Z3");
year = INT2FIX(2001);
VALUE argv[] = {make, model, year};
int argc = sizeof argv / sizeof argv[0]; // 3
ID class_id = rb_intern("Car");
VALUE class = rb_const_get(rb_cObject, class_id);
VALUE new_car = rb_class_new_instance(argc, argv, class);

return new_car;
} // end of tradeCar

void Init_dealer() {
    VALUE class = rb_define_class("Dealer", rb_cObject);
    int arg_count = 1;
    rb_define_method(class, "initialize", dealerInit, arg_count);
    rb_define_method(class, "trade_car", tradeCar, arg_count);
}
```

Copyright © 2007 by Object Computing, Inc. (OCI).
All rights reserved.

Ruby C Extensions

28

Pass/Return Ruby Objects - client.rb

```
require 'dealer'

class Car
  attr_accessor :make, :model, :year

  def initialize(make, model, year)
    @make, @model, @year = make, model, year
  end

  def to_s
    "#{year} #{make} #{model}"
  end
end

dealer = Dealer.new("Bud's Used Cars")
old_car = Car.new("Saturn", "SC2", 1997)
new_car = dealer.trade_car(old_car)
puts "traded #{old_car} for #{new_car}"
```

Output

```
tradeCar received 1997 Saturn SC2
traded 2007 Saturn SC2 for 2001 BMW Z3
```

What About C++?

- Can call loose C++ functions (not in a class) just like C functions
 - but name mangling must be disabled by wrapping function definitions in

```
extern "C" {
  ...
}
```
- These can use C++ class and instance methods

Ruby Inline

- Allows C code to be imbedded in Ruby code
- See documentation at
 - <http://www.zenspider.com/ZSS/Products/RubyInline/>
- Setup
 - `gem install rubyinline`

SWIG

- Simplified Wrapper and Interface Generator (SWIG)
- See documentation at
 - <http://www.swig.org>
 - <http://www.swig.org/Doc1.3/Ruby.html>
- Email me for my slides on this

More Information

- **README.EXT**
 - in Ruby distribution
- **Programming Ruby, 2nd Edition**
 - Chapter 21