Extended ruby

Extending Ruby: Ruby Objects in C, the Jukebox extension, Memory allocation, Ruby Type System, Embedding Ruby to Other Languages, Embedding a Ruby Interpreter

Writing Ruby in C

- One of the joys of Ruby is that you can write Ruby programs almost directly in C. That is, you can use the same methods and the same logic, but with slightly different syntax to accommodate C. For instance, here is a small, fairly test class written in Ruby.

```
class Test
    def initialize
      @arr = Array.new
    end
    def add(anObject)
       @arr.push(anObject)
    end
end
```

- The equivalent code in C should look somewhat familiar.

```
#include "ruby.h"
static VALUE t_init(VALUE self)
{
VALUE arr; arr = rb_ary_new();
rb_iv_set(self, "@arr", arr);
return self; }
static VALUE t_add(VALUE self, VALUE anObject)
 {
 VALUE arr; arr = rb_iv_get(self, "@arr");
rb_ary_push(arr, anObject); return arr;
}
VALUE cTest;
void Init_Test() {
 cTest = rb_define_class("Test", rb_cObject);
 rb_define_method(cTest, "initialize", t_init, 0);
rb_define_method(cTest, "add", t_add, 1);
}
```

Memory Allocation

- You may sometimes need to allocate memory in an extension that won't be used for object storage—perhaps you've got a giant bitmap for a Bloom filter, or an image, or a whole bunch of little structures that Ruby doesn't use directly.
- In order to work correctly with the garbage collector, you should use the following memory allocation routines. These routines do a little bit more work than the standard malloc. For instance, if ALLOC_N determines that it cannot allocate the desired amount of memory, it will invoke the garbage collector to try to reclaim some space. It will raise a NoMemError if it can't or if the requested amount of memory is invalid.

- *type* * **ALLOC_N**(*c-type*, n)Allocates *n c-type* objects, where *c-type* is the literal name of the C type, not a variable of that type.
- *type* * **ALLOC**(*c-type*)Allocates a *c-type* and casts the result to a pointer of that type.**REALLOC_N**(*var*, *c-type*, n)Reallocates *n c-type*s and assigns the result to *var*, a pointer to a *c-type*.
- *type* * **ALLOCA_N**(*c-type*, n)Allocates memory for *n* objects of *c-type* on the stack—this memory will be automatically freed when the function that invokes ALLOCA_N returns.

Creating an Extension

- Having written the source code for an extension, we now need to compile it so Ruby can use it. We can either do this as a shared object, which is dynamically loaded at runtime, or statically link the extension into the main Ruby interpreter itself. The basic procedure is the same:
- Create the C source code file(s) in a given directory.
- Create extconf.rb.
- Run extconf.rb to create a Makefile for the C files in this directory.
- Run make.
- Run make install.

Creating an Extension

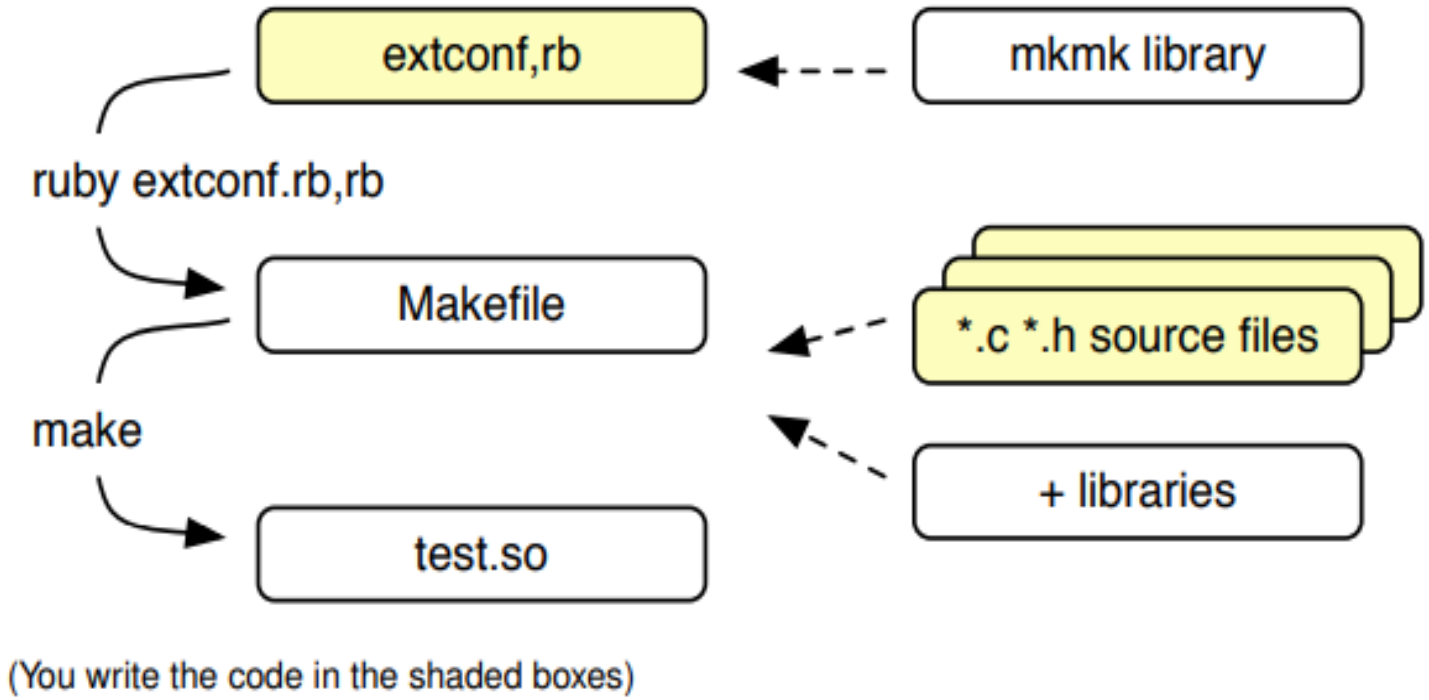

Figure 2.5: Building an Extension.

Embedding a Ruby Interpreter

- In addition to extending Ruby by adding C code, you can also turn the problem around and embed Ruby itself within your application. Here's an example.
- #include "ruby.h"
- main()
- { /* … our own application stuff … */
- ruby_init();
- ruby_script("embedded");
- rb_load_file("start.rb");
- while (1) {
- if (need_to_do_ruby) {
- ruby_run(); } /* … run our app stuff */
- } }

- To initialize the Ruby interpreter, you need to call ruby_init(). But on some platforms, you may need to take special steps before that:

```
#if defined(NT)
    NtInitialize(&argc, &argv);
#endif
#if defined(__MACOS__) && defined(__MWERKS__)
    argc = ccommand(&argv);
#endif
```

- See main.c in the Ruby distribution for any other special defines or setup needed for your platform.

Embedded Ruby API

void **ruby_init**()

Sets up and initializes the interpreter. This function should be called before any other Ruby-related functions.

void **ruby_options**(int argc, char **argv)

Gives the Ruby interpreter the command-line options.

void **ruby_script**(char *name)

Sets the name of the Ruby script (and $0) to *name*.

void **rb_load_file**(char *file)

Loads the given file into the interpreter.

void **ruby_run**()

Runs the interpreter.

Bridging Ruby to Other Languages

- So far, we've discussed extending Ruby by adding routines written in C. However, you can write extensions in just about any language, as long as you can bridge the two languages with C. Almost anything is possible, including awkward marriages of Ruby and C++, Ruby and Java, and so on.

Ruby C Language API

- So far, we've discussed extending Ruby by adding routines written in C. However, you can write extensions in just about any language, as long as you can bridge the two languages with C. Almost anything is possible, including awkward marriages of Ruby and C++, Ruby and Java, and so on.
- But you may be able to accomplish the same thing without resorting to C code. For example, you could bridge to other languages using middleware such as CORBA or COM.
- Last, but by no means least, here are several C-level functions that you may find useful when writing an extension.
- Some functions require an ID: you can obtain an ID for a string by using rb_intern and reconstruct the name from an ID by using rb_id2name.

- As most of these C functions have Ruby equivalents, the descriptions here will be brief.

- Also note that the following listing is not complete. There are many more functions available—too many to document them all, as it turns out.

- If you need a method that you can't find here, check “ruby.h” or “intern.h” for likely candidates. Also, at or near the bottom of each source file is a set of method definitions that describe the binding from Ruby methods to C functions. You may be able to call the C function directly, or search for a wrapper function that calls the function you are looking for. The following list, based on the list in README.EXT, shows the main source files in the interpreter.

Ruby Language Core

class.c error.c eval.c gc.c object.c parse.y variable.c

Utility Functions

dln.c regex.c st.c util.cRuby

Interpreter

dmyext.c inits.c keywords main.c ruby.c version.c

Base Library

array.c bignum.c compar.c dir.c enum.c file.c hash.c io.c marshal.c math.c numeric.c pack.c prec.c process.c random.c range.c re.c signal.c sprintf.c string.c struct.c time.c

Ruby C Language API

Defining Objects

`VALUE` **`rb_define_class`**`(char *name, VALUE superclass")`

Defines a new class at the top level with the given *name* and *superclass* (for class `Object`, use `rb_cObject`).

`VALUE` **`rb_define_module`**`(char *name")`

Defines a new module at the top level with the given *name*.

`VALUE` **`rb_define_class_under`**`(VALUE under, char *name, VALUE superclass")`

Defines a nested class under the class or module *under*.

`VALUE` **`rb_define_module_under`**`(VALUE under, char *name")`

Defines a nested module under the class or module *under*.

`void` **`rb_include_module`**`(VALUE parent, VALUE module")`

Includes the given *module* into the class or module *parent*.

`void` **`rb_extend_object`**`(VALUE obj, VALUE module")`

Extends *obj* with *module*.

`VALUE` **`rb_require`**`(const char *name")`

Equivalent to ``` ``require` *name*." Returns `Qtrue` or `Qfalse`.

Ruby C Language API

Defining Methods

`void` **`rb_define_method`**`(VALUE classmod, char *name, VALUE(*func)(), int argc")`

Defines an instance method in the class or module *classmod* with the given *name*, implemented by the C function *func* and taking *argc* arguments.

`void` **`rb_define_module_function`**`(VALUE classmod, char *name, VALUE(*func)(), int argc)")`

Defines a method in class *classmod* with the given *name*, implemented by the C function *func* and taking *argc* arguments.

`void` **`rb_define_global_function`**`(char *name, VALUE(*func)(), int argc")`

Defines a global function (a private method of `Kernel`) with the given *name*, implemented by the C function *func* and taking *argc* arguments.

`void` **`rb_define_singleton_method`**`(VALUE classmod, char *name, VALUE(*func)(), int argc")`

Defines a singleton method in class *classmod* with the given *name*, implemented by the C function *func* and taking *argc* arguments.

`int` **`rb_scan_args`**`(int argcount, VALUE *argv, char *fmt, ...")`

Scans the argument list and assigns to variables similar to `scanf`: *fmt* is a string containing zero, one, or two digits followed by some flag characters. The first digit indicates the count of mandatory arguments; the second is the count of optional arguments. A ``*" means to pack the rest of the arguments into a Ruby array. A ``&" means that an attached code block will be taken and assigned to the given variable (if no code block was given, `Qnil` will be assigned). After the *fmt* string, pointers to `VALUE` are given (as with `scanf`) to which the arguments are assigned.

```
VALUE name, one, two, rest;
rb_scan_args(argc, argv, "12", &name, &one, &two);
rb_scan_args(argc, argv, "1*", &name, &rest);
```

`void` **`rb_undef_method`**`(VALUE classmod, const char *name")`

Undefines the given method *name* in the given *classmod* class or module.

`void` **`rb_define_alias`**`(VALUE classmod, const char *newname, const char *oldname")`

Defines an alias for *oldname* in class or module *classmod*.

Ruby C Language API

Defining Variables and Constants

`void` **`rb_define_const`**`(VALUE classmod, char *name, VALUE value")`

Defines a constant in the class or module *classmod*, with the given *name* and *value*.

`void` **`rb_define_global_const`**`(char *name, VALUE value")`

Defines a global constant with the given *name* and *value*.

`void` **`rb_define_variable`**`(const char *name, VALUE *object")`

Exports the address of the given *object* that was created in C to the Ruby namespace as *name*. From Ruby, this will be a global variable, so *name* should start with a leading dollar sign. Be sure to honor Ruby's rules for allowed variable names; illegally named variables will not be accessible from Ruby.

`void` **`rb_define_class_variable`**`(VALUE class, const char *name, VALUE val")`

Defines a class variable *name* (which must be specified with a ``@@" prefix) in the given *class*, initialized to *value*.

`void` **`rb_define_virtual_variable`**`(const char *name, VALUE(*getter)(), void(*setter)()")`

Exports a virtual variable to Ruby namespace as the global $*name*. No actual storage exists for the variable; attempts to get and set the value will call the given functions with the prototypes:

```
VALUE getter(ID id, VALUE *data,
             struct global_entry *entry);
void setter(VALUE value, ID id, VALUE *data,
            struct global_entry *entry);
```

You will likely not need to use the *entry* parameter and can safely omit it from your function declarations.

`void` **`rb_define_hooked_variable`**`(const char *name, VALUE *variable, VALUE(*getter)(), void(*setter)()")`

Defines functions to be called when reading or writing to *variable*. See also `rb_define_virtual_variable`.

`void` **`rb_define_readonly_variable`**`(const char *name, VALUE *value")`

Same as `rb_define_variable`, but read-only from Ruby.

`void` **`rb_define_attr`**`(VALUE variable, const char *name, int read, int write")`

Creates accessor methods for the given *variable*, with the given *name*. If *read* is nonzero, create a read method; if *write* is nonzero, create a write method.

`void` **`rb_global_variable`**`(VALUE *obj")`

Registers the given address with the garbage collector.

Ruby C Language API

Accessing Variables

`VALUE` **`rb_iv_get`**`(VALUE obj, char *name")`

Returns the instance variable *name* (which must be specified with a ``@'' prefix) from the given *obj*.

`VALUE` **`rb_ivar_get`**`(VALUE obj, ID name")`

Returns the instance variable *name* from the given *obj*.

`VALUE` **`rb_iv_set`**`(VALUE obj, char *name, VALUE value")`

Sets the value of the instance variable *name* (which must be specified with a ``@'' prefix) in the given *obj* to *value*. Returns *value*.

`VALUE` **`rb_ivar_set`**`(VALUE obj, ID name, VALUE value")`

Sets the value of the instance variable *name* in the given *obj* to *value*. Returns *value*.

`VALUE` **`rb_gv_set`**`(const char *name, VALUE value")`

Sets the global variable *name* (the ``$'' prefix is optional) to *value*. Returns *value*.

`VALUE` **`rb_gv_get`**`(const char *name")`

Returns the global variable *name* (the ``$'' prefix is optional).

`void` **`rb_cvar_set`**`(VALUE class, ID name, VALUE val")`

Sets the class variable *name* in the given *class* to *value*.

`VALUE` **`rb_cvar_get`**`(VALUE class, ID name")`

Returns the class variable *name* from the given *class*.

`int` **`rb_cvar_defined`**`(VALUE class, ID name")`

Returns `Qtrue` if the given class variable *name* has been defined for *class*; otherwise, returns `Qfalse`.

`void` **`rb_cv_set`**`(VALUE class, const char *name, VALUE val")`

Sets the class variable *name* (which must be specified with a ``@@'' prefix) in the given *class* to *value*.

`VALUE` **`rb_cv_get`**`(VALUE class, const char *name")`

Returns the class variable *name* (which must be specified with a ``@@'' prefix) from the given *class*.