Chapter 29

Extending Ruby

It is easy to extend Ruby with new features by writing code in Ruby. But every now and then you need to interface to things at a lower level. Once you start adding in low-level code written in C, the possibilities are endless. Having said this, the stuff in this chapter is pretty advanced and should probably be skipped the first time through the book.

Extending Ruby with C is pretty easy. For instance, suppose we are building a custom Internet-ready jukebox for the Sunset Diner and Grill. It will play MP3 audio files from a hard disk or audio CDs from a CD jukebox. We want to be able to control the jukebox hardware from a Ruby program. The hardware vendor gave us a C header file and a binary library to use; our job is to construct a Ruby object that makes the appropriate C function calls.

Much of the information in this chapter is taken from the `README.EXT` file that is included in the distribution. If you are planning on writing a Ruby extension, you may want to refer to that file for more details as well as the latest changes.

Your First Extension

Just to introduce extension writing, let's write one. This extension is purely a test of the process—it does nothing that you couldn't do in pure Ruby. We'll also present some stuff without too much explanation; all the messy details will be given later.

The extension we write will have the same functionality as the following Ruby class:

Download **samples/extruby_1.rb**

```
class MyTest
  def initialize
    @arr = Array.new
  end
  def add(obj)
    @arr.push(obj)
  end
end
```

That is, we'll be writing an extension in C that is plug-compatible with that Ruby class. The equivalent code in C should look somewhat familiar:

Download **samples/extruby_2.rb**

```
#include "ruby.h"
static int id_push;
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 obj)
{
  VALUE arr;
  arr = rb_iv_get(self, "@arr");
  rb_funcall(arr, id_push, 1, obj);
  return arr;
}
VALUE cTest;
void Init_my_test() {
  cTest = rb_define_class("MyTest", rb_cObject);
  rb_define_method(cTest, "initialize", t_init, 0);
  rb_define_method(cTest, "add", t_add, 1);
  id_push = rb_intern("push");
}
```

Let's go through this example in detail, because it illustrates many of the important concepts in this chapter. First, we need to include the header file `ruby.h` to obtain the necessary Ruby definitions.

Now look at the last function, Init_my_test. Every extension defines a C global function named Init_*name*. This function will be called when the interpreter first loads the extension *name* (or on startup for statically linked extensions). It is used to initialize the extension and to insinuate it into the Ruby environment. (Exactly how Ruby knows that an extension is called *name* we'll cover later.) In this case, we define a new class named MyTest, which is a subclass of Object (represented by the external symbol rb_cObject; see `ruby.h` for others).

Next we set up add and initialize as two instance methods for class MyTest. The calls to rb_define_method establish a binding between the Ruby method name and the C function that will implement it. If Ruby code calls the add method on one of our objects, the interpreter will in turn call the C function t_add with one argument.

Similarly, when new is called for this class, Ruby will construct a basic object and then call initialize, which we have defined here to call the C function t_init with no (Ruby) arguments.

Now go back and look at the definition of t_init. Even though we said it took no arguments, it has a parameter here! In addition to any Ruby arguments, every method is passed an initial VALUE argument that contains the receiver for this method (the equivalent of self in Ruby code).

The first thing we'll do in t_init is create a Ruby array and set the instance variable @arr to point to it. Just as you would expect if you were writing Ruby source, referencing an instance variable that doesn't exist creates it. We then return a pointer to ourselves.

WARNING: Every C function that is callable from Ruby *must* return a VALUE, even if it's just Qnil. Otherwise, a core dump (or GPF) will be the likely result.

Finally, the function t_add gets the instance variable @arr from the current object and calls Array#push to push the passed value onto that array. When accessing instance variables in this way, the @ prefix is mandatory—otherwise, the variable is created but cannot be referenced from Ruby.

Despite the extra, clunky syntax that C imposes, you're still writing in Ruby—you can manipulate objects using all the method calls you've come to know and love, with the added advantage of being able to craft tight, fast code when needed.

Building Our Extension

We'll have a lot more to say about building extensions later. For now, though, all we have to do is follow these steps:

1. Create a file called `extconf.rb` in the same directory as our `my_test.c` C source file. The file `extconf.rb` should contain the following two lines:

   Download **samples/extruby_3.rb**

   ```
   require 'mkmf'
   create_makefile("my_test")
   ```

2. Run `extconf.rb`. This will generate a Makefile:

   ```
   % ruby extconf.rb
   creating Makefile
   ```

3. Use make to build the extension. On an OS X system, you'd see:

   ```
   % make
   gcc -fno-common -g -O2 -pipe -fno-common  -I.
       -I/usr/lib/ruby/1.9/powerpc-darwin7.4.0
      -I/usr/lib/ruby/1.9/powerpc-darwin7.4.0 -I.   -c my_test.c
   cc -dynamic -bundle -undefined suppress -flat_namespace
       -L'/usr/lib' -o my_test.bundle my_test.o  -ldl -lobjc
   ```

The result of all this is the extension, all nicely bundled up in a shared object (a `.so`, a `.dll`, or [on OS X] a `.bundle`).

Running Our Extension

We can use our extension from Ruby simply by require-ing it dynamically at runtime (on most platforms). We can wrap this up in a test to verify that things are working as we expect:

Download **samples/extruby_4.rb**

```
require 'my_test'
require 'test/unit'
class TestTest < Test::Unit::TestCase
  def test_test
    t = MyTest.new
    assert_equal(Object, MyTest.superclass)
    assert_equal(MyTest, t.class)
    t.add(1)
    t.add(2)
    assert_equal([1,2], t.instance_eval("@arr"))
  end
end
```

produces:

```
Finished in 0.000412 seconds.
1 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

Once we're happy that our extension works, we can then install it globally by running make install.

Ruby Objects in C

When we wrote our first extension, we cheated, because it didn't really do anything with the Ruby objects. For example, it didn't do calculations based on Ruby numbers. Before we can do this, we need to find out how to represent and access Ruby data types from within C.

Everything in Ruby is an object, and all variables are references to objects. When we're looking at Ruby objects from within C code, the situation is pretty much the same. Most Ruby objects are represented as C pointers to an area in memory that contains the object's data and other implementation details. In C code, all these references are via variables of type VALUE, so when you pass Ruby objects around, you'll do it by passing VALUEs.

This has one exception. For performance reasons, Ruby implements Fixnums, Symbols, true, false, and nil as so-called immediate values. These are still stored in variables of type VALUE, but they aren't pointers. Instead, their value is stored directly in the variable.

So, sometimes VALUEs are pointers, and sometimes they're immediate values. How does the interpreter pull off this magic? It relies on the fact that all pointers point to areas of memory aligned on 4- or 8-byte boundaries. This means that it can guarantee that the low 2 bits in a pointer will always be zero. When it wants to store an immediate value, it arranges to have at least one of these bits set, allowing the rest of the interpreter code to distinguish immediate values from pointers. Although this sounds tricky, it's actually easy to use in

practice, largely because the interpreter comes with a number of macros and methods that simplify working with the type system.

This is how Ruby implements object-oriented code in C: a Ruby object is an allocated structure in memory that contains a table of instance variables and information about the class. The class itself is another object (an allocated structure in memory) that contains a table of the methods defined for that class. Ruby is built upon this foundation.

Working with Immediate Objects

As we said earlier, immediate values are not pointers: Fixnum, Symbol, true, false, and nil are stored directly in VALUE.

Fixnum values are stored as 31-bit numbers[1] that are formed by shifting the original number left 1 bit and then setting the LSB, or least significant bit (bit 0), to 1. When VALUE is used as a pointer to a specific Ruby structure, it is guaranteed always to have an LSB of zero; the other immediate values also have LSBs of zero. Thus, a simple bit test can tell you whether you have a Fixnum. This test is wrapped in a macro, FIXNUM_P. Similar tests let you check for other immediate values.

```
FIXNUM_P(value) → nonzero if value is a Fixnum
SYMBOL_P(value) → nonzero if value is a Symbol
NIL_P(value)    → nonzero if value is nil
RTEST(value)    → nonzero if value is neither nil nor false
```

Several useful conversion macros for numbers as well as other standard data types are shown in Table 29.1 on the next page.

The other immediate values (true, false, and nil) are represented in C as the constants Qtrue, Qfalse, and Qnil, respectively. You can test VALUE variables against these constants directly or use the conversion macros (which perform the proper casting).

Working with Strings

In C, we're used to working with null-terminated strings. Ruby strings, however, are more general and may well include embedded nulls. The safest way to work with Ruby strings, therefore, is to do what the interpreter does and use both a pointer and a length. In fact, Ruby String objects are actually references to an RString structure, and the RString structure contains both a length and a pointer field. You can access the structure via the RSTRING macros. This is a change in Ruby 1.9—prior to this you manipulated the C structure directly.

1.9

```
VALUE str;
RSTRING_LEN(str) → length of the Ruby string
RSTRING_PTR(str) → pointer to string storage
RSTRING_END(str) → pointer to end of string
```

1. Or 63-bit on wider CPU architectures

Table 29.1. C/Ruby Data Type Conversion Functions and Macros

C Data Types to Ruby Objects:		
	INT2NUM(*int*)	→ *Fixnum* or *Bignum*
	INT2FIX(*int*)	→ *Fixnum* (faster)
	LONG2NUM(*long*	→ *Fixnum* or *Bignum*
	LONG2FIX(*int*)	→ *Fixnum* (faster)
	LL2NUM(*long long*)	→ *Fixnum* or *Bignum* (if native system supports *long long* type)
	ULL2NUM(*long long*)	→ *Fixnum* or *Bignum* (if native system supports *long long* type)
	CHR2FIX(*char*)	→ *Fixnum*
	rb_str_new2(*char* *)	→ *String*
	rb_float_new(*double*)	→ *Float*
Ruby Objects to C Data Types:		
int	NUM2INT(*Numeric*)	(Includes type check)
int	FIX2INT(*Fixnum*)	(Faster)
unsigned int	NUM2UINT(*Numeric*)	(Includes type check)
unsigned int	FIX2UINT(*Fixnum*)	(Includes type check)
long	NUM2LONG(*Numeric*)	(Includes type check)
long	FIX2LONG(*Fixnum*)	(Faster)
unsigned long	NUM2ULONG(*Numeric*)	(Includes type check)
char	NUM2CHR(*Numeric* or *String*)	(Includes type check)
double	NUM2DBL(*Numeric*)	
	see text for strings...	

However, life is slightly more complicated than that. Rather than using the VALUE object directly when you need a string value, you probably want to call the method StringValue, passing it the original value. It'll return an object that you can use the RSTRING_ macros on or throw an exception if it can't derive a string from the original. This is all part of Ruby duck typing conventions, described in more detail on pages 853 and 370. The StringValue method checks to see whether its operand is a String. If not, it tries to invoke to_str on the object, throwing a TypeError exception if it can't.

So, if you want to write some code that iterates over all the characters in a String object, you may write the following:

Download **samples/extruby_5.rb**

```
static VALUE iterate_over(VALUE original_str) {
  int i;
  char *p;
  VALUE str = StringValue(original_str);
  p = RSTRING_PTR(str);   // may be null
  for (i = 0; i < RSTRING_LEN(str); i++, p++) {
    // process *p
  }
  return str;
}
```

If you want to bypass the length and just access the underlying string pointer, you can use the convenience method StringValuePtr, which both resolves the string reference and then returns the C pointer to the contents.

If you plan to use a string to access or control some external resource, you probably want to hook into Ruby's tainting mechanism. In this case, you'll use the method SafeStringValue, which works like StringValue but throws an exception if its argument is tainted and the safe level is greater than zero.

Working with Other Objects

When VALUEs are not immediate; they are pointers to one of the defined Ruby object structures—you can't have a VALUE that points to an arbitrary area of memory. The structures for the basic built-in classes are defined in ruby.h and are named R*Classname*: RArray, RBignum, RClass, RData, RFile, RFloat, RHash, RObject, RRegexp, RString, and RStruct.

You can check to see what type of structure is used for a particular VALUE in a number of ways. The macro TYPE(*obj*) will return a constant representing the C type of the given object: T_OBJECT, T_STRING, and so on. Constants for the built-in classes are defined in ruby.h. Note that the *type* we are referring to here is an implementation detail—it is not the same as the class of an object.

If you want to ensure that a VALUE pointer points to a particular structure, you can use the macro Check_Type, which will raise a TypeError exception if *value* is not of the expected *type* (which is one of the constants T_STRING, T_FLOAT, and so on):

```
Check_Type(VALUE value, int type)
```

Having said all this, you need to be careful about building too much dependence on checking types into your extension code. We have more to say about extensions and the Ruby type system on page 853.

Again, note that we are talking about "type" as the C structure that represents a particular built-in type. The class of an object is a different beast entirely. The class objects for the built-in classes are stored in C global variables named rb_c*Classname* (for instance, rb_cObject); modules are named rb_m*Modulename*.

It isn't advisable to alter the data in these C structures directly, however—you may look, but don't touch. Instead, you'll normally use the supplied C functions to manipulate Ruby data (we'll talk more about this in just a moment).

However, in the interests of efficiency, you may need to dig into these structures to obtain data. To dereference members of these C structures, you have to cast the generic VALUE to the proper structure type. ruby.h contains a number of macros that perform the proper casting for you, allowing you to dereference structure members easily. These macros are named R*CLASSNAME* (which returns the whole structure) and R*CLASSNAME_xxx*, which let you access the specific fields in the structure. For example, RSTRING(obj) will return a reference to the RString structure pointed to by obj; RSTRING_PTR(obj) and RSTRING_LEN(obj) return the contents and length. There are similar accessors for hashes (RHASH), files (RFILE), and so on. The full list is shown in 29.2 on page 841.

1.9

Pre-1.9 String Access

Prior to Ruby 1.8.6, you'd access RSTRING fields directly to get the length and string data pointer. So, how do you write an extension that works with either technique? Perhaps like this:

Download **samples/extruby_6.rb**

```
#if !defined(RSTRING_LEN)
#  define RSTRING_LEN(x) (RSTRING(x)->len)
#  define RSTRING_PTR(x) (RSTRING(x)->ptr)
#endif
```

Global Variables

Most of the time, your extensions will implement classes, and the Ruby code uses those classes. The data you share between the Ruby code and the C code will be wrapped tidily inside objects of the class. This is how it should be.

Sometimes, though, you may need to implement a global variable, accessible by both your C extension and by Ruby code.

The easiest way to do this is to have the variable be a VALUE (that is, a Ruby object). You then bind the address of this C variable to the name of a Ruby variable. In this case, the $ prefix is optional, but it helps clarify that this is a global variable. Make sure you don't make a stack-based variable a Ruby global: it won't exist once the stack frame is gone.

```
static  VALUE hardware_list;
static VALUE Init_SysInfo() {
  rb_define_class(....);
  hardware_list = rb_ary_new();
  rb_define_variable("$hardware", &hardware_list);
  ...
  rb_ary_push(hardware_list, rb_str_new2("DVD"));
  rb_ary_push(hardware_list, rb_str_new2("CDPlayer1"));
  rb_ary_push(hardware_list, rb_str_new2("CDPlayer2"));
}
```

The Ruby side can then access the C variable hardware_list as $hardware:

```
$hardware   # =>   ["DVD", "CDPlayer1", "CDPlayer2"]
```

Sometimes, though, life is more complicated. Perhaps you want to define a global variable whose value must be calculated when it is accessed. You do this by defining *hooked* and *virtual* variables. A hooked variable is a real variable that is initialized by a named function when the corresponding Ruby variable is accessed. Virtual variables are similar but are never stored; their value purely comes from evaluating the hook function. See the API section that begins on page 868 for details.

Table 29.2. Object Accessor Macros

RClass c;	
RCLASS_M_TBL(c)	Pointer to table of methods
RCLASS_SUPER(c)	Pointer to superclass
RModule m;	
RMODULE_IV_TBL(m)	
RMODULE_M_TBL(m)	Same as classes
RMODULE_SUPER(m)	
RFloat v;	
RFLOAT_VALUE(v)	The value as a native float
RString str;	
RSTRING_LEN(str)	Length of the string
RSTRING_PTR(str)	Pointer to start of string data
RSTRING_END(str)	Pointer to end of string data
RArray a;	
RARRAY_LEN(a)	Number of elements in the array
RARRAY_PTR(a)	Pointer to start of array data (treat as being read-only)
RHash h;	
RHASH_TBL(h)	Pointer to the hash data (see st.c
RHASH_ITER_LEV(h)	If nonzero, hash is being traversed by an iterator
RHASH_IFNONE(h)	Default value for hash (may be a proc object)
RHASH_SIZE(h)	Number of entries in hash
RHASH_EMPTY_P(h)	True if RHASH_SIZE(h) is zero

If you create a Ruby object from C and store it in a C global variable *without* exporting it to Ruby, you must at least tell the garbage collector about it, lest ye be reaped inadvertently:

```
static VALUE obj;
// ...
obj = rb_ary_new();
rb_global_variable(obj);
```

The Threading Model

Previous Ruby interpreters implemented threading internally. These *green threads* relied on the interpreter periodically switching between Ruby-level threads. Even if your computer had eight cores, your Ruby program would run on only one of them at a time. And, if you called a long-running external function, your whole program would hang. DNS name resolution was a notorious culprit.

1.9

Ruby 1.9 now uses operating system threads. This allows your multithreaded Ruby programs to do more in parallel than was possible earlier. However, before you run cheering

into the streets, I have to tell you a catch. Although the core interpreter is thread-safe, extension libraries probably aren't. And problems that arise when you run non-thread-safe code multithreaded are incredibly hard to diagnose, and they have a habit of being extremely damaging to data. So, Matz made a decision: the Ruby VM can run in multiple threads, but it will execute Ruby code in only one of those threads at a time.

This means that you probably won't be writing your next high-volume telephone exchange microcode in Ruby. But it still brings benefits. Whereas previously the DNS lookup would stall all your code, in Ruby 1.9 it will run in an operating system thread. If your program has other Ruby-level threads waiting to execute, they can run while the looking is executing. Another example is the multiplication of very large Bignum values. Whereas in the old interpreter the multiplication would hog the CPU and no other threads would run, Ruby 1.9 schedules such multiplications in one operating system thread and allows other Ruby-level threads to operate in true parallel.

When you write your own Ruby extensions, you need to make sure they play well in this new world. In particular, if they are about to undertake some long-running external operation, you need to make sure that other Ruby threads are allowed to run in parallel.

Ruby controls which threads can run using the GVL, or *Giant VM Lock*. The thread that's currently executing Ruby-level code will have claimed the GVL, and no other thread will be able to claim it (and hence execute Ruby code) until the lock is relinquished. Your extension code normally doesn't worry about this—Ruby handles it for you. But if your extension is about to make (say) a long-running I/O request, it will need to temporarily relinquish the GVL until that request completes. To do this, call the method rb_thread_blocking_region (which we'll abbreviate to rb_tbr in the description that follows—in your code you have to spell out the full name). This method takes the address of a C function and a pointer to a parameter to pass to that function. It also takes a second function/parameter pair, which we'll describe shortly.

Internally, rb_tbr releases the GVL and then calls the function you specify, passing it the parameter you gave in the call. When the function terminates, rb_tbr then reacquires the GVL before returning back to you. If it were written in Ruby, an incomplete implementation might look like this:

```
def rb_thread_blocking_region(param, unblock_function, unblock_flag)
  release_lock(GVL)
  begin
    yield(param)
  ensure
    acquire_lock(GVL)
  end
end
rb_thread_blocking_region(task_data, ...) do |data|
  # some long-running external call
end
```

Before we look at the C-level implementation, we need to discuss the second function/parameter pair that you pass to rb_thread_blocking_region. These represent an *unblocking function*, or *ubf*. There are times that Ruby needs to terminate the execution of code that's being

run in a separate thread. For example, a thread may call your extension library to start some long-running operation, such as a file copy. Your code uses rb_tbr to allow the rest of the interpreter to run during this process. But maybe one of those other threads calls exit to shut down the interpreter. Ruby needs to coordinate this shutdown with the work that's taking place in your thread. It does this by calling the unblocking function that you pass to rb_tbr. This function is responsible for terminating whatever activity was initiated by rb_tbr.

Let's look at some C code that uses this feature. This example is taken from bignum.c in the main interpreter, but it applies to code that is in an extension, too.

First, the code defines a structure that is used to pass parameters to the function that executes in parallel. In this example, the same structure is also passed to the unblocking function:

```
struct big_mul_struct {
    VALUE x, y, z, stop;
};
```

Here's the body of the method that is called to multiply two bignums. I've cut out some of the boring stuff in the middle.

```
static VALUE
rb_big_mul0(VALUE x, VALUE y)
{
  struct big_mul_struct bms;
  volatile VALUE z;
  /* ... */
  bms.x = x;
  bms.y = y;
  bms.stop = Qfalse;
  if (RBIGNUM_LEN(x) + RBIGNUM_LEN(y) > 10000) {
    z = rb_thread_blocking_region(bigmul1, &bms, rb_big_stop, &bms.stop);
  }
  else {
    z = bigmul1(&bms);
  }
  return z;
}
```

The important code for us is at the end of the method. If the numbers being multiplied are big, it calls rb_thread_blocking_region, passing in the function to run (bigmul1), the parameter to pass to it (bms), and the unblocking function and parameter (rb_big_stop and bms). If instead the numbers are below the threshold, it calls bigmul1 directly.

Now let's look at the unblocking function, rb_big_stop:

```
static void
rb_big_stop(void *ptr)
{
    VALUE *stop = (VALUE*)ptr;
    *stop = Qtrue;
}
```

If called, it simply sets the location referenced by its parameter to Qtrue.

And how does this stop the calculation prematurely? Let's look at bigmul1 (again, omitting some gory details):

```
static VALUE
bigmul1(void *ptr)
{
    struct big_mul_struct *bms = (struct big_mul_struct*)ptr;
    long i, j;
    VALUE x = bms->x, y = bms->y, z = bms->z;
    /* ... */
    for (i = 0; i < RBIGNUM_LEN(x); i++) {
        if (bms->stop) return Qnil;
        /*
         * do the next digit...
         */
    }
    return z;
}
```

So, if the unblocking function is called, it sets the stop member of the structure to Qtrue. Then in the loop that's doing the multiplication and that is running in a separate thread, it notices that the flag has been set and exits early, returning nil.

Threads, Processes, and I/O

If the function you pass to rb_tbr executes I/O or synchronizes with an external process, you're left with a difficult decision when it comes time to write the unblocking function. Interrupting I/O is tricky and is most likely system-dependent. Luckily for you, Ruby provides a sledgehammer to crack that particular nut. If you pass RUBY_UBF_IO or RUBY_UBF_PROCESS as the third parameter (and NULL as the fourth), Ruby will use a default ubf function that simply kills the thread performing the processing. Here's the code from the interpreter method that reads from a file descriptor:

```
static int
rb_read_internal(int fd, void *buf, size_t count)
{
  struct io_internal_struct iis;
  iis.fd = fd;
  iis.buf = buf;
  iis.capa = count;
    return rb_thread_blocking_region(internal_read_func, &iis,
                                     RUBY_UBF_IO, 0);
}
```

You'll need to decide whether you should use these built-in ubfs in your own extension. For example, if updating a database table, you'll probably want to perform a more controlled shutdown than simply terminating the controlling thread.

The Jukebox Extension

We've covered enough of the basics now to return to our jukebox example—interfacing C code with Ruby and sharing data and behavior between the two worlds.

Wrapping C Structures

We have the vendor's library that controls the audio CD jukebox units, and we're ready to wire it into Ruby. The vendor's header file looks like this:

```
typedef struct _cdjb {
  int   statusf;
  int   request;
  void *data;
  char  pending;
  int   unit_id;
  void *stats;
} CDJukebox;
// Allocate a new CDJukebox structure
CDJukebox *new_jukebox(void);
// Assign the Jukebox to a player
void assign_jukebox(CDJukebox *jb, int unit_id);
// Deallocate when done (and take offline)
void free_jukebox(CDJukebox *jb);
// Seek to a disc, track and notify progress
void jukebox_seek(CDJukebox *jb,
                  int disc,
                  int track,
                  void (*done)(CDJukebox *jb, int percent));
// ... others...
// Report a statistic
double get_avg_seek_time(CDJukebox *jb);
```

This vendor has its act together; although they might not admit it, the code is written with an object-oriented flavor. We don't know what all those fields mean within the CDJukeBox structure, but that's OK—we can treat it as an opaque pile of bits. The vendor's code knows what to do with it; we just have to carry it around.

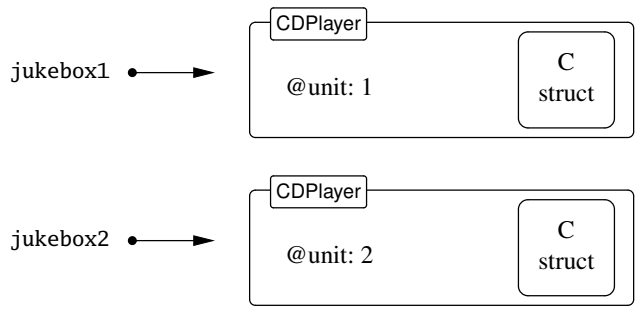
Any time you have a C-only structure that you would like to handle as a Ruby object, you should wrap it in a special, internal Ruby class called DATA (type T_DATA). Two macros do this wrapping, and one macro retrieves your structure back out again.

API: C Data Type Wrapping

VALUE **Data_Wrap_Struct**(VALUE class, void (*mark)(), void (*free)(), void *ptr)

Wraps the given C data type *ptr*, registers the two garbage collection routines (explained in a moment), and returns a VALUE pointer to a genuine Ruby object. The C type of the resulting object is T_DATA, and its Ruby class is *class*.

Figure 29.1. Wrapping Objects Around C Data Types



VALUE **Data_Make_Struct**(VALUE class, *c-type*, void (*mark)(), void (*free)(), *c-type* *)
Allocates and sets to zero a structure of the indicated type first and then proceeds as Data_Wrap_Struct. *c-type* is the name of the C data type that you're wrapping, not a variable of that type.

Data_Get_Struct(VALUE obj,*c-type*,*c-type* *)
Returns the original pointer. This macro is a type-safe wrapper around the macro DATA_PTR(obj), which evaluates the pointer.

The object created by Data_Wrap_Struct is a normal Ruby object, except that it has an additional C data type that can't be accessed from Ruby. As you can see in Figure 29.1, this C data type is separate from any instance variables that the object contains. But since it's a separate thing, how do you get rid of it when the garbage collector claims this object? What if you have to release some resource (close some file, clean up some lock or IPC mechanism, and so on)?

Ruby uses a "mark-and-sweep" garbage collection scheme. During the mark phase, Ruby looks for pointers to areas of memory. It marks these areas as "in use" (because something is pointing to them). If those areas themselves contain more pointers, the memory these pointers reference is also marked, and so on. At the end of the mark phase, all memory that is referenced will have been marked, and any orphaned areas will not have a mark. At this point, the sweep phase starts, freeing memory that isn't marked.

To participate in Ruby's mark-and-sweep garbage collection process, you must define a routine to free your structure and possibly a routine to mark any references from your structure to other structures. Both routines take a void pointer, a reference to your structure. The *mark* routine will be called by the garbage collector during its mark phase. If your structure references other Ruby objects, then your mark function needs to identify these objects using rb_gc_mark(*value*). If the structure doesn't reference other Ruby objects, you can simply pass 0 as a function pointer.

When the object needs to be disposed of, the garbage collector will call the *free* routine to free it. If you've allocated any memory yourself (for instance, by using Data_Make_Struct),

you'll need to pass a free function—even if it's just the standard C library's free routine. For complex structures that you have allocated, your free function may need to traverse the structure to free all the allocated memory.

Let's look at our CD player interface. The vendor library passes the information around between its various functions in a CDJukebox structure. This structure represents the state of the jukebox and therefore is a good candidate for wrapping within our Ruby class. You create new instances of this structure by calling the library's CDPlayerNew method. You'd then want to wrap that created structure inside a new CDPlayer Ruby object. A fragment of code to do this may look like the following. (We'll talk about that magic *klass* parameter in a minute.)

```
CDJukebox *jukebox;
VALUE obj;
// Vendor library creates the Jukebox
jukebox = new_jukebox();
// then we wrap it inside a Ruby CDPlayer object
obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
```

Once this code executed, *obj* would hold a reference to a newly allocated CDPlayer Ruby object, wrapping a new CDJukebox C structure. Of course, to get this code to compile, we'd need to do some more work. We'd have to define the CDPlayer class and store a reference to it in the variable cCDPlayer. We'd also have to define the function to free off our object, cdplayer_free. That's easy, because it just calls the vendor library dispose method:

```
static void cd_free(void *p) {
  free_jukebox(p);
}
```

However, code fragments do not a program make. We need to package all this stuff in a way that integrates it into the interpreter. And to do that, we need to look at some of the conventions the interpreter uses.

Object Creation

Let's start by looking at how you allocate the memory for a new object. The basic idea is simple. Let's say you're creating an object of class CDPlayer in your Ruby program:

```
cd = CDPlayer.new
```

Underneath the covers, the interpreter calls the class method new for CDPlayer. Because CDPlayer hasn't defined a method new, Ruby looks into its parent, class Class.

The implementation of new in class Class is fairly simple; it allocates memory for the new object and then calls the object's initialize method to initialize that memory.

So, if our CDPlayer extension is to be a good Ruby citizen, it should work within this framework. This means that we'll need to implement an allocation function and an initialize method.

Allocation Functions

The allocation function is responsible for creating the memory used by your object. If the object you're implementing doesn't use any data other than Ruby instance variables, then you don't need to write an allocation function—Ruby's default allocator will work just fine. But if your class wraps a C structure, you'll need to allocate space for that structure in the allocation function. The allocation function gets passed the class of the object being allocated. In our case, it will in all likelihood be a cCDPlayer, but we'll use the parameter as given, because this means that we'll work correctly if subclassed:

```
static VALUE cd_alloc(VALUE klass) {
  CDJukebox *jukebox;
  VALUE obj;
  // Vendor library creates the Jukebox
  jukebox = new_jukebox();
  // then we wrap it inside a Ruby CDPlayer object
  obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
  return obj;
}
```

You then need to register your allocation function in your class's initialization code:

```
void Init_CDPlayer() {
  cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
  rb_define_alloc_func(cCDPlayer, cd_alloc);
  // ...
}
```

Most objects probably need to define an initializer too. The allocation function creates an empty, uninitialized object, and we'll need to fill in specific values. In the case of the CD player, the constructor is called with the unit number of the player to be associated with this object:

```
static VALUE cd_initialize(VALUE self, VALUE unit) {
  int unit_id;
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  unit_id = NUM2INT(unit);
  assign_jukebox(jb, unit_id);
  return self;
}
```

One of the reasons for this multistep object creation protocol is that it lets the interpreter handle situations where objects have to be created by "back-door means." One example is when objects are being deserialized from their marshaled form. Here, the interpreter needs to create an empty object (by calling the allocator), but it cannot call the initializer (because it has no knowledge of the parameters to use). Another common situation is when objects are duplicated or cloned.

One further issue lurks here. Because users can choose to bypass the constructor, you need to ensure that your allocation code leaves the returned object in a valid state. It may not contain all the information it would have had, had it been set up by #initialize, but it at least needs to be usable.

Cloning Objects

All Ruby objects can be copied using one of two methods, dup and clone. The two methods are similar. Both produce a new instance of their receiver's class by calling the allocation function. Then they copy across any instance variables from the original. clone then goes a bit further and copies the original's singleton class (if it has one) and flags (such as the flag that indicates that an object is frozen). You can think of dup as being a copy of the contents and clone as being a copy of the full object.

However, the Ruby interpreter doesn't know how to handle copying the internal state of objects that you write as C extensions. For example, if your object wraps a C structure that contains an open file descriptor, it's up to the semantics of your implementation whether that descriptor should simply be copied to the new object or whether a new file descriptor should be opened.

To handle this, the interpreter delegates to your code the responsibility of copying the internal state of objects that you implement. After copying the object's instance variables, the interpreter invokes the new object's initialize_copy method, passing in a reference to the original object. It's up to you to implement meaningful semantics in this method.

For our CDPlayer class, we'll take a fairly simple approach to the cloning issue. We'll simply copy across the CDJukebox structure from the original object.

There's a wee chunk of strange code in this example. To test that the original object is indeed something we can clone the new one from, the code checks to see that the original

- has a TYPE of T_DATA (which means that it's a noncore object), and
- has a free function with the same address as our free function.

This is a relatively high-performance way of verifying that the original object is compatible with our own (as long as you don't share free functions between classes). An alternative, which is slower, would be to use rb_obj_is_kind_of and do a direct test on the class:

```
static VALUE cd_init_copy(VALUE copy, VALUE orig) {
  CDJukebox *orig_jb;
  CDJukebox *copy_jb;
  if (copy == orig)
    return copy;
  // we can initialize the copy from other CDPlayers
  // or their subclasses only
  if (TYPE(orig) != T_DATA ||
      RDATA(orig)->dfree != (RUBY_DATA_FUNC)cd_free) {
    rb_raise(rb_eTypeError, "wrong argument type");
  }
  // copy all the fields from the original
  // object's CDJukebox structure to the
  // new object
  Data_Get_Struct(orig, CDJukebox, orig_jb);
  Data_Get_Struct(copy, CDJukebox, copy_jb);
  MEMCPY(copy_jb, orig_jb, CDJukebox, 1);
  return copy;
}
```

Our copy method does not have to allocate a wrapped structure to receive the original objects CDJukebox structure; the cd_alloc method has already taken care of that.

Note that in this case it's correct to do type checking based on classes. We need the original object to have a wrapped CDJukebox structure, and the only objects that have one of these are derived from class CDPlayer.

Putting It All Together

OK, finally we're ready to write all the code for our CDPlayer class:

Download **samples/extruby_24.rb**

```
#include "ruby.h"
#include "cdjukebox.h"
static VALUE cCDPlayer;
// Helper function to free a vendor CDJukebox
static void cd_free(void *p) {
  free_jukebox(p);
}
// Allocate a new CDPlayer object, wrapping
// the vendor's CDJukebox structure
static VALUE cd_alloc(VALUE klass) {
  CDJukebox *jukebox;
  VALUE obj;
  // Vendor library creates the Jukebox
  jukebox = new_jukebox();
  // then we wrap it inside a Ruby CDPlayer object
  obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
  return obj;
}
// Assign the newly created CDPLayer to a
// particular unit
static VALUE cd_initialize(VALUE self, VALUE unit) {
  int unit_id;
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  unit_id = NUM2INT(unit);
  assign_jukebox(jb, unit_id);
  return self;
}
// Copy across state (used by clone and dup).  For jukeboxes, we
// actually create a new vendor object and set its unit number from
// the old
static VALUE cd_init_copy(VALUE copy, VALUE orig) {
  CDJukebox *orig_jb;
  CDJukebox *copy_jb;
  if (copy == orig)
    return copy;
```

```
  // we can initialize the copy from other CDPlayers or their
  // subclasses only
  if (TYPE(orig) != T_DATA ||
      RDATA(orig)->dfree != (RUBY_DATA_FUNC)cd_free) {
    rb_raise(rb_eTypeError, "wrong argument type");
  }
  // copy all the fields from the original object's CDJukebox
  // structure to the new object
  Data_Get_Struct(orig, CDJukebox, orig_jb);
  Data_Get_Struct(copy, CDJukebox, copy_jb);
  MEMCPY(copy_jb, orig_jb, CDJukebox, 1);
  return copy;
}
// The progress callback yields to the caller the percent complete
static void progress(CDJukebox *rec, int percent) {
  if (rb_block_given_p()) {
    if (percent > 100) percent = 100;
    if (percent < 0) percent = 0;
    rb_yield(INT2FIX(percent));
  }
}
// Seek to a given part of the track, invoking the progress callback
// as we go
static VALUE
cd_seek(VALUE self, VALUE disc, VALUE track) {
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  jukebox_seek(jb,
               NUM2INT(disc),
               NUM2INT(track),
               progress);
  return Qnil;
}
// Return the average seek time for this unit
static VALUE
cd_seek_time(VALUE self)
{
  double tm;
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  tm = get_avg_seek_time(jb);
  return rb_float_new(tm);
}
// Return this player's unit number
static VALUE
cd_unit(VALUE self) {
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
```

```
  return INT2NUM(jb->unit_id);
}

void Init_CDPlayer() {
  cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
  rb_define_alloc_func(cCDPlayer, cd_alloc);
  rb_define_method(cCDPlayer, "initialize", cd_initialize, 1);
  rb_define_method(cCDPlayer, "initialize_copy", cd_init_copy, 1);
  rb_define_method(cCDPlayer, "seek", cd_seek, 2);
  rb_define_method(cCDPlayer, "seek_time", cd_seek_time, 0);
  rb_define_method(cCDPlayer, "unit", cd_unit, 0);
}
```

Now we can control our jukebox from Ruby in a nice, object-oriented way:

Download **samples/extruby_25.rb**

```
require 'CDPlayer'
p = CDPlayer.new(13)
puts "Unit is #{p.unit}"
p.seek(3, 16) {|x| puts "#{x}% done" }
puts "Avg. time was #{p.seek_time} seconds"
p1 = p.dup
puts "Cloned unit = #{p1.unit}"
```

produces:

```
Unit is 13
26% done
79% done
100% done
Avg. time was 1.2 seconds
Cloned unit = 13
```

This example demonstrates most of what we've talked about so far, with one additional neat feature. The vendor's library provided a callback routine—a function pointer that is called every so often while the hardware is grinding its way to the next disc. We've set that up here to run a code block passed as an argument to seek. In the progress function, we check to see whether there is an iterator in the current context and, if there is, run it with the current percent done as an argument.

Memory Allocation

You may sometimes need to allocate memory in an extension that won't be used for object storage—perhaps you have a giant bitmap for a Bloom filter, an image, or a whole bunch of little structures that Ruby doesn't use directly.

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.

API: Memory Allocation

*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 variable of type *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.

Ruby Type System

In Ruby, we rely less on the type (or class) of an object and more on its capabilities. This is called *duck typing*. We describe it in more detail in Chapter 23 on page 370. You'll find many examples of this if you examine the source code for the interpreter itself. For example, the String class implements the method %. This treats the string object as a format specifier (just like the C sprintf function). The % method can take a single argument (if the string contains just one substitution) or an array of values (if the format string contains multiple substitutions).

```
irb(main):001:0> "You bought %d x %s" % [ 3, "widgets"]
=> "You bought 3 x widgets"
irb(main):002:0> "The total is %0.2f" % 5.678
=> "The total is 5.68"
```

Here's the code in `string.c` that implements the String.% method:

```
static VALUE
rb_str_format_m(VALUE str, VALUE arg)
{
    VALUE tmp = rb_check_array_type(arg);
    if (!NIL_P(tmp)) {
        return rb_str_format(RARRAY_LEN(tmp), RARRAY_PTR(tmp), str);
    }
    return rb_str_format(1, &arg, str);
}
```

The first parameter to this method is the *self* object—the format string. The arg argument is either an array or a single object. However, the code doesn't explicitly check the type of the argument. Instead, it first calls rb_check_array_type, passing in the argument. What does this method do? Let's see (the code is in `array.c`):

Download **samples/extruby_28.rb**

```
VALUE
rb_check_array_type(ary)
    VALUE ary;
{
    return rb_check_convert_type(ary, T_ARRAY, "Array", "to_ary");
}
```

The plot thickens. Let's track down rb_check_convert_type in `object.c`:

Download **samples/extruby_29.rb**

```
VALUE
rb_check_convert_type(VALUE val, int type, const char *tname,
                      const char *method)
{
    VALUE v;
    /* always convert T_DATA */
    if (TYPE(val) == type && type != T_DATA) return val;
    v = convert_type(val, tname, method, Qfalse);
    if (NIL_P(v)) return Qnil;
    if (TYPE(v) != type) {
        char *cname = rb_obj_classname(val);
        rb_raise(rb_eTypeError, "can't convert %s to %s (%s#%s gives %s)",
                 cname, tname, cname, method, rb_obj_classname(v));
    }
    return v;
}
```

Now we're getting somewhere. If the object is the correct type (T_ARRAY in our example), then the original object is returned. Otherwise, we don't give up quite yet. Instead, we use the convert_type method to call our original object and ask whether it can represent itself as an array (we call its to_ary method). If it can, we're happy and continue. The code is saying "I don't need an Array; I just need something that can be represented as an array." This means that String.% will accept as an array any parameter that implements a to_ary method. We discuss these conversion protocols in more detail (but from the Ruby perspective) starting on page 376.

What does all this mean to you as an extension writer? There are two messages. First, try to avoid checking the types of parameters passed to you. Instead, see whether there's a rb_check_xxx_type method that will convert the parameter into the type that you need. If not, look for an existing conversion function (such as rb_Array, rb_Float, or rb_Integer) that'll do the trick for you. Second, if you're writing an extension that implements something that may be meaningfully used as a Ruby string or array, consider implementing to_str or to_ary methods, allowing objects implemented by your extension to be used in string or array contexts.

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:

1. Create the C source code file(s) in a given directory.
2. Optionally create any supporting Ruby files in a lib subdirectory.
3. Create extconf.rb.
4. Run extconf.rb to create a Makefile for the C files in this directory.
5. Run make.
6. Run make install.

Creating a Makefile with extconf.rb

The overall workflow when building an extension is shown in Figure 29.2 on the following page. The key to the whole process is the extconf.rb program that you, as a developer, create. extconf.rb is simple program that determines what features are available on the user's system and where those features may be located. Executing extconf.rb builds a customized Makefile, tailored for both your application and the system on which it's being compiled. When you run the make command against this Makefile, your extension is built and (optionally) installed. If you have multiple versions of Ruby installed on your system, the one used when you run extconf.rb is the one your extension is built and installed against.

The simplest extconf.rb may be just two lines long, and for many extensions this is sufficient:

Download **samples/extruby_30.rb**

```
require 'mkmf'
create_makefile("Test")
```

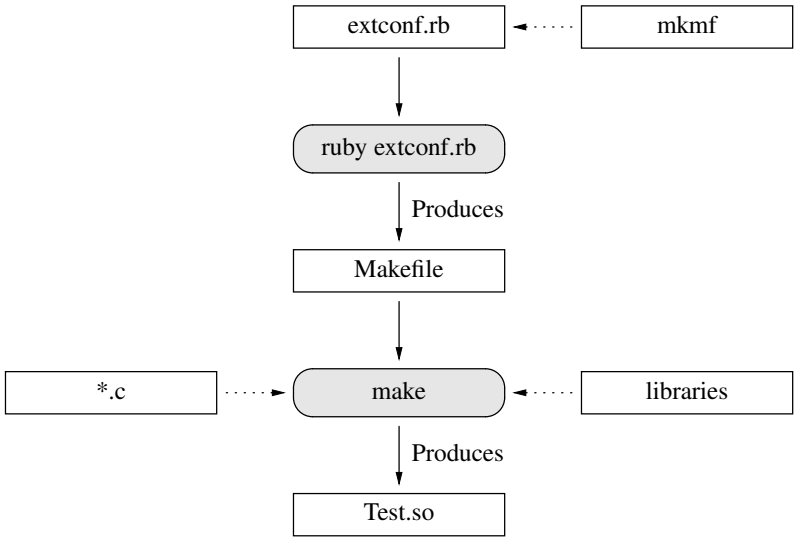
The first line brings in the mkmf library module (described starting on page 874). This contains all the commands we'll be using. The second line creates a Makefile for an extension called "Test." (Note that "Test" is the name of the extension; the file will always be called Makefile.) Test will be built from all the C source files in the current directory. When your code is loaded, Ruby will call its Init_Test method.

Let's say that we run this extconf.rb program in a directory containing a single source file, main.c. The result is a Makefile that will build our extension. On a Linux box, this executes the following commands (your commands will likely be different):

```
gcc -fPIC -I/usr/local/lib/ruby/1.9/i686-linux -g -O2  \
  -c main.c -o main.o
gcc -shared -o Test.so main.o -lc
```

The result of this compilation is Test.so, which may be dynamically linked into Ruby at runtime with require.

Figure 29.2. Building an Extension



Under Mac OS X, the commands are different, but the result is the same; a shared object (a *bundle* on the Mac) is created:

```
gcc -fno-common -g -O2 -pipe -fno-common \
    -I/usr/lib/ruby/1.9.0/powerpc-darwin   \
    -I/usr/lib/ruby/1.9.0/powerpc-darwin -c main.c
cc -dynamic -bundle -undefined suppress -flat_namespace \
    -L'/usr/lib'  -o Test.bundle main.o -lruby -lpthread -ldl -lobjc
```

See how the mkmf commands have automatically located platform-specific libraries and used options specific to the local compiler. Pretty neat, eh?

Although this basic extconf.rb program works for many simple extensions, you may have to do some more work if your extension needs header files or libraries that aren't included in the default compilation environment or if you conditionally compile code based on the presence of libraries or functions.

A common requirement is to specify nonstandard directories where include files and libraries may be found. This is a two-step process. First, your extconf.rb should contain one or more dir_config commands. This specifies a tag for a set of directories. Then, when you run the extconf.rb program, you tell mkmf where the corresponding physical directories are on the current system.

Dividing Up the Namespace

Increasingly, extension writers are being good citizens. Rather than install their work directory into one of Ruby's library directories, they're using subdirectories to group their files together. This is easy with extconf.rb. If the parameter to the create_makefile call contains forward slashes, mkmf assumes that everything before the last slash is a directory name and that the remainder is the extension name. The extension will be installed into the given directory (relative to the Ruby directory tree). In the following example, the extension will still be named Test:

```
require 'mkmf'
create_makefile("wibble/Test")
```

However, when you require this class in a Ruby program, you'd write this:

```
require 'wibble/Test'
```

If extconf.rb contains the line dir_config(*name*), then you give the location of the corresponding directories with the command-line options:

--with-*name*-include=*directory*
: Adds *directory*/include to the compile command.

--with-*name*-lib=*directory*
: Adds *directory*/lib to the link command.

If (as is common) your include and library directories are subdirectories called include and lib of some other directory, you can take a shortcut:

--with-*name*-dir=*directory*
: Adds *directory*/lib and *directory*/include to the link command and compile command, respectively.

As well as specifying all these --with options when you run extconf.rb, you can also use the --with options that were specified when Ruby was built for your machine. This means you can discover and use the locations of libraries that are used by Ruby itself. It also means that you can specify the locations of all libraries just once and then rebuild extensions as many times as you like.

To make all this concrete, let's say you need to use the vendor's CDJukebox libraries and include files for the CD player we're developing. Your extconf.rb may contain this:

```
require 'mkmf'
dir_config('cdjukebox')
# .. more stuff
create_makefile("CDPlayer")
```

You'd then run extconf.rb with something like this:

```
% ruby extconf.rb --with-cdjukebox-dir=/usr/local/cdjb
```

The generated Makefile would assume that `/usr/local/cdjb/lib` contained the libraries and `/usr/local/cdjb/include` the include files.

The dir_config command adds to the list of places to search for libraries and include files. It does not, however, link the libraries into your application. To do that, you'll need to use one or more have_library or find_library commands.

have_library looks for a given entry point in a named library. If it finds the entry point, it adds the library to the list of libraries to be used when linking your extension. find_library is similar but allows you to specify a list of directories to search for the library. Here are the contents of the extconf.rb that we use to link our CD player:

```
require 'mkmf'
dir_config("cdjukebox")
have_library("cdjukebox", "new_jukebox")
create_makefile("CDPlayer")
```

A particular library may be in different places depending on the host system. The X Window system, for example, is notorious for living in different directories on different systems. The find_library command will search a list of supplied directories to find the right one (this is different from have_library, which uses only configuration information for the search). For example, to create a Makefile that uses X Windows and a JPEG library, extconf.rb may contain this:

```
require 'mkmf'
if have_library("jpeg","jpeg_mem_init") and
   find_library("X11", "XOpenDisplay",
                "/usr/X11/lib",        # list of directories
                "/usr/X11R6/lib",      # to check
                "/usr/openwin/lib")    # for library
then
    create_makefile("XThing")
else
    puts "No X/JPEG support available"
end
```

We've added some functionality to this program. All the mkmf commands return false if they fail. This means we can write an extconf.rb that generates a Makefile only if everything it needs is present. The Ruby distribution does this so that it will try to compile only those extensions that are supported on your system.

You also may want your extension code to be able to configure the features it uses depending on the target environment. For example, our CD jukebox may be able to use a high-performance MP3 decoder if the end user has one installed.

We can check by looking for its header file:

Download samples/extruby_39.rb

```
require 'mkmf'
dir_config('cdjukebox')
have_library('cdjb', 'CDPlayerNew')
have_header('hp_mp3.h')
create_makefile("CDJukeBox")
```

We can also check to see whether the target environment has a particular function in any of the libraries we'll be using. For example, the setpriority call would be useful but isn't always available. We can check for it with this:

```
require 'mkmf'
dir_config('cdjukebox')
have_func('setpriority')
create_makefile("CDJukeBox")
```

Both have_header and have_func define preprocessor constants if they find their targets. The names are formed by converting the target name to uppercase and prepending HAVE_. Your C code can take advantage of this using constructs such as the following:

```
#if defined(HAVE_HP_MP3_H)
#  include <hp_mp3.h>
#endif
#if defined(HAVE_SETPRIORITY)
  err = setpriority(PRIOR_PROCESS, 0, -10)
#endif
```

If you have special requirements that can't be met with all these mkmf commands, your program can directly add to the global variables $CFLAGS and $LFLAGS, which are passed to the compiler and linker, respectively.

Sometimes you'll create an extconf.rb and it just doesn't seem to work. You give it the name of a library, and it swears that no such library has ever existed on the entire planet. You tweak and tweak, but mkmf still can't find the library you need. It would be nice if you could find out exactly what it's doing behind the scenes. Well, you can. Each time you run your extconf.rb script, mkmf generates a log file containing details of what it did. If you look in mkmf.log, you'll be able to see what steps the program used to try to find the libraries you requested. Sometimes trying these steps manually will help you track down the problem.

Installation Target

The Makefile produced by your extconf.rb will include an "install" target. This will copy your shared library object into the correct place on your (or your users') local file system. The destination is tied to the installation location of the Ruby interpreter you used to run extconf.rb in the first place. If you have multiple Ruby interpreters installed on your box, your extension will be installed into the directory tree of the one that ran extconf.rb.

In addition to installing the shared library, extconf.rb will check for the presence of a lib/ subdirectory. If it finds one, it will arrange for any Ruby files there to be installed along with your shared object. This is useful if you want to split the work of writing your extension between low-level C code and higher-level Ruby code.

Static Linking

Finally, if your system doesn't support dynamic linking or if you have an extension module that you want to have statically linked into Ruby itself, edit the file ext/Setup in the distribution and add your directory to the list of extensions in the file. In your extension's directory, create a file named MANIFEST containing a list of all the files in your extension (source, extconf.rb, lib/, and so on). Then rebuild Ruby. The extensions listed in Setup will be statically linked into the Ruby executable. If you want to disable any dynamic linking and link all extensions statically, edit ext/Setup to contain the following option.

Download **samples/extruby_42.rb**

```
option nodynamic
```

A Shortcut

If you are extending an existing library written in C or C++, you may want to investigate SWIG (http://www.swig.org). SWIG is an interface generator: it takes a library definition (typically from a header file) and automatically generates the glue code needed to access that library from another language. SWIG supports Ruby, meaning that it can generate the C source files that wrap external libraries in Ruby classes.

Embedding a Ruby Interpreter

1.9

In addition to extending Ruby by adding C code, you can also turn the problem around and embed Ruby itself within your application. As of Ruby 1.9, you can no longer call ruby_run to invoke loaded code. Instead, you can call it either via the C API or by evaluating strings.

Let's start with the Ruby program we want to embed. Here's a simple Ruby class that implements a method to return the sum of the numbers from 1 to *max*:

Download **samples/extruby_43.rb**

```
class Summer
  def sum(max)
    raise "Invalid maximum #{max}" if max < 0
    (max*max + max)/2
  end
end
```

Let's see how to invoke this from a C program.

First we'll interact by evaluating strings of Ruby code:

Download samples/extruby_44.rb

```
#include "ruby.h"
int main(int argc, char **argv) {
  VALUE result;
  ruby_sysinit(&argc, &argv);
  RUBY_INIT_STACK;
  ruby_init();
  ruby_init_loadpath();
  rb_require("sum");   // or sum.rb
  rb_eval_string("$summer = Summer.new");
  rb_eval_string("$result = $summer.sum(10)");
  result = rb_gv_get("result");
  printf("Result = %d\n", NUM2INT(result));
  return ruby_cleanup(0);
}
```

To initialize the Ruby interpreter, you need to call ruby_sysinit() to pick up command-line arguments used by Ruby, RUBY_INIT_STACK to set up the Ruby stack, and ruby_init to initialize the interpreter itself. The call to ruy_init_loadpath adds any directories to be searched for libraries (for example if your RUBYLIB environment variable is set).

Once this prelude is out of the way, we can start using our external Ruby file. We load it into the interpreter using rb_require and then evaluate two lines of code using rb_eval_string. Notice that we assign the results of these two lines to global variables. We could also have used instance variables, but globals are easier to manipulate at the top level. We could not have used local variables: they don't persist across calls to eval.

Once our sum has been calculated, we need to get it back into our C code. The call to rb_gv_get gets the value of a global variable but returns it as a Ruby object. We convert it to a native integer via NUM2INT before printing the result (which is 55).

In order to compile this code, you need the Ruby include and library files accessible. When writing this book on my box (Mac OS X), I have the Ruby 1.9 interpreter installed in a private directory, so my Makefile looks like the following. (Note that Ruby 1.9 changes the location of the `ruby.h` file.)

1.9

Download samples/extruby_45.rb

```
LIB=/usr/local/rubybook/lib
INC=/usr/local/rubybook/include/ruby-1.9.0/ruby
CFLAGS=-I$(INC) -g
LDFLAGS=-L$(LIB) -lruby -ldl -lobjc
embed:  embed.o
        $(CC) -o embed embed.o $(LDFLAGS)
```

This kind of hands-off manipulation of Ruby programs from within C code is easy, but it has two major drawbacks. First, it's indirect—we have to keep storing things in globals and

extracting the values from these globals out to use them. Second, we're not doing any real error checking, which will definitely bite us later.

So, the second way to interact with Ruby code is to use the C API. This gives us much finer-grained control and also lets us handle errors. You do this by initializing the interpreter as normal. Then, rather than evaluating strings, you instead invoke specific methods in your Ruby code. When these methods return, your C code gets control back.

There's a wrinkle, though. If the Ruby code raises an exception and it isn't caught, your C program will terminate. To overcome this, you need to do what the interpreter does and protect all calls that could raise an exception. This can get messy. The rb_protect method call wraps the call to another C function. That second function should invoke our Ruby method. However, the method wrapped by rb_protect is defined to take just a single parameter. Passing more involves some ugly C casting.

Let's look at an example. Let's write a C program that calls an instance of this class multiple times.

To create the instance, we'll get the class object (by looking for a top-level constant whose name is the name of our class). We'll then ask Ruby to create an instance of that class—rb_class_new_instance is actually a call to Class.new. (The two initial 0 parameters are the argument count and a dummy pointer to the arguments themselves.) Once we have that object, we can invoke its sum method using rb_funcall.

Download **samples/extruby_46.rb**

```
#include "ruby.h"
static int id_sum;
int Values[] = { 5, 10, 15, -1, 20, 0 };
static VALUE wrap_sum(VALUE args) {
  VALUE *values = (VALUE *)args;
  VALUE  summer = values[0];
  VALUE  max    = values[1];
  return rb_funcall(summer, id_sum, 1, max);
}
static VALUE protected_sum(VALUE summer, VALUE max) {
  int error;
  VALUE args[2];
  VALUE result;
  args[0] = summer;
  args[1] = max;
  result = rb_protect(wrap_sum, (VALUE)args, &error);
  return error ? Qnil : result;
}
int main(int argc, char **argv) {
  int value;
  int *next = Values;
  int error;
  ruby_sysinit(&argc, &argv);
  RUBY_INIT_STACK;
  ruby_init();
```

```
  ruby_init_loadpath();
  ruby_script("demo_embedder");  /* sets name in error messages */
  rb_protect((VALUE (*)(VALUE))rb_require, (VALUE)"sum", &error);
  // get an instance of Summer
  VALUE summer = rb_class_new_instance(0, 0,
                    rb_const_get(rb_cObject, rb_intern("Summer")));
  id_sum = rb_intern("sum");
  while (value = *next++) {
    VALUE  result = protected_sum(summer, INT2NUM(value));
    if (NIL_P(result))
      printf("Sum to %d doesn't compute!\n", value);
    else
      printf("Sum to %d is %d\n", value, NUM2INT(result));
  }
  return ruby_cleanup(0);
}
```

The ugly part of this code is the protected_sum method. We want to use rb_funcall to call the sum method in our Summar object, but rb_funcall takes four parameters. The only way to handle exceptions raised in Ruby code from the C API is to wrap the call to Ruby code using the rb_protect method. But rb_protect wraps only those methods that have the following signature:

```
VALUE *method(VALUE arg)
```

To work around this, protected_sum creates a two-element array containing the parameters it wants to pass to rb_funcall and then uses a C cast to fake out that this array is a single VALUE argument. The first parameter to rb_protect is the name of a proxy method, wrap_sum, that unbundles these arguments and the calls rb_funcall. If you do a lot of this kind of work, it would be worthwhile writing a simple wrapper library to simplify this kind of coding.

One last thing: the Ruby interpreter was not originally written with embedding in mind. Probably the biggest problem is that it maintains state in global variables, so it isn't thread-safe. You can embed Ruby—just one interpreter per process.

API: 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_init_loadpath**( )

Initializes the $: (load path) variable; necessary if your code loads any library modules.

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.

void **ruby_finalize**( )
: Shuts down the interpreter.

For another example of embedding a Ruby interpreter within another program, see also eruby, which is described beginning on page 310.

Bridging Ruby to Other Environments

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 such as Ruby and C++.

There's a bigger story here, though. In the past, there was effectively only one Ruby implementation. But at the time of writing, we now have a number of alternative implementations. As well as Matz's original Ruby interpreter (commonly called MRI), there's JRuby,[2] IronRuby,[3] Ruby.NET,[4] and Rubinius[5] (with other implementations waiting in the wings and not yet released).

So, if you're looking to integrate Ruby code and Java code, then you should probably consider JRuby. It allows you to bridge pretty much seamlessly between the two languages. It runs Ruby on Rails and provides adapters so that you can (for example) use existing entity beans as Rails model objects.

If you want to integrate Ruby into a Microsoft environment, IronRuby gives you an implementation targeted at the DLR, while Ruby.NET targets the CLR.

Rubinius is interesting—it uses the original Matz Ruby parser but contains a totally different VM implementation. Its goal it to have code run so fast that it is possible to write the majority of Ruby's libraries in Ruby itself (rather than having, for example, the String class written in C, as it is in MRI). Right now, it looks like they might be able to achieve this, which will be a major win for portability and extensibility.

2. http://jruby.codehaus.org/
3. http://www.ironruby.net/
4. http://rubydotnet.googlegroups.com/web/Home.htm
5. http://rubini.us/

However, be careful. Ruby does not really have an official specification, so these implementations may exhibit differing behaviors when dealing with edge conditions. Also, currently MRI implements only Ruby 1.9.

Ruby C Language API

Last, but by no means least, here are some 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.

Because most of these C functions have Ruby equivalents that are already described in detail elsewhere in this book, the descriptions here will be brief.

The following listing is not complete. Many more functions are 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 describes 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 need. 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, id.c, object.c, parse.y, variable.c

Utility Functions
dln.c, reg*.c, st.c, util.c

Ruby Interpreter
blockinlining.c, compile.c, debug.c, dmy*.c, inits.c, iseq.c, keywords, main.c, ruby.c, version.c, vm*.c

Encoding and Character Sets
enc/*, encoding.c, transcode.c

Base Library
array.c, bignum.c, compar.c, cont.c, dir.c, enum.c, enumerator.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, thread*.c, time.c

API: Defining Classes

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.

API: Defining Structures

VALUE **rb_struct_define**(char *name, char *attribute..., NULL)
Defines a new structure with the given attributes.

VALUE **rb_struct_new**(VALUE sClass, VALUE args..., NULL)
Creates an instance of *sClass* with the given attribute values.

VALUE **rb_struct_aref**(VALUE struct, VALUE idx)
Returns the element named or indexed by *idx*.

VALUE **rb_struct_aset**(VALUE struct, VALUE idx, VALUE val)
Sets the attribute named or indexed by *idx* to *val*.

API: Defining Methods

In some of the function definitions that follow, the parameter *argc* specifies how many arguments a Ruby method takes. It may have the following values. If the value is not negative, it specifies the number of arguments the method takes. If negative, it indicates that the method takes optional arguments. In this case, the absolute value of *argc* minus one is the number of required arguments (so -1 means all arguments are optional, -2 means one mandatory argument followed by optional arguments, and so on).

In a function that has been given a variable number of arguments, you can use the C function rb_scan_args to sort things out (see below).

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_alloc_func**(VALUE classmod, VALUE(*func)())
Identifies the allocator for *classmod*.

void **rb_define_module_function**(VALUE module, char *name, VALUE(*func)(), int argc))

Defines a method in *module* 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 (class) 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.

Download **samples/extruby_47.rb**

```
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*.

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 a 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.

Download **samples/extruby_48.rb**

```
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, creates a read method; if *write* is nonzero, creates a write method.

void **rb_global_variable**(VALUE *obj)

Registers the given address with the garbage collector.

API: Calling Methods

VALUE **rb_class_new_instance**((int argc, VALUE *argv, VALUE klass))

Return a new instance of class *klass*. *argv* is a pointer to an array of *argc* parameters.

VALUE **rb_funcall**(VALUE recv, ID id, int argc, ...)
Invokes the method given by *id* in the object *recv* with the given number of arguments *argc* and the arguments themselves (possibly none).

VALUE **rb_funcall2**(VALUE recv, ID id, int argc, VALUE *args)
Invokes the method given by *id* in the object *recv* with the given number of arguments *argc* and the arguments themselves given in the C array *args*.

VALUE **rb_funcall3**(VALUE recv, ID id, int argc, VALUE *args)
Same as rb_funcall2 but will not call private methods.

VALUE **rb_apply**(VALUE recv, ID name, VALUE args)
Invokes the method given by *id* in the object *recv* with the arguments given in the Ruby Array *args*.

ID **rb_intern**(char *name)
Returns an ID for a given *name*. If the name does not exist, a symbol table entry will be created for it.

char * **rb_id2name**(ID id)
Returns a name for the given *id*.

VALUE **rb_call_super**(int argc, VALUE *args)
Calls the current method in the superclass of the current object.

API: Exceptions

void **rb_raise**(VALUE exception, const char *fmt, ...)
Raises an *exception*. The given string *fmt* and remaining arguments are interpreted as with printf.

void **rb_fatal**(const char *fmt, ...)
Raises a Fatal exception, terminating the process. No rescue blocks are called, but ensure blocks will be called. The given string *fmt* and remaining arguments are interpreted as with printf.

void **rb_bug**(const char *fmt, ...)
Terminates the process immediately—no handlers of any sort will be called. The given string *fmt* and remaining arguments are interpreted as with printf. You should call this function only if a fatal bug has been exposed. You don't write fatal bugs, do you?

void **rb_sys_fail**(const char *msg)
Raises a platform-specific exception corresponding to the last known system error, with the given *msg*.

VALUE **rb_rescue**(VALUE (*body)(), VALUE args, VALUE(*rescue)(), VALUE rargs)

Executes *body* with the given *args*. If a StandardError exception is raised, then execute *rescue* with the given *rargs*.

VALUE **rb_ensure**(VALUE(*body)(), VALUE args, VALUE(*ensure)(), VALUE eargs)

Executes *body* with the given *args*. Whether or not an exception is raised, execute *ensure* with the given *eargs* after *body* has completed.

VALUE **rb_protect**(VALUE (*body)(), VALUE args, int *state)

Executes *body* with the given *args* and returns nonzero in *result* if any exception was raised. The value in state corresponds to the various TAG_xxx macros. In Ruby 1.9, these are defined in `eval\_intern.h` (although there's a strong argument for moving them to `ruby.h`, as `eval\_intern.h` is not accessible to extension writers).

void **rb_notimplement**( )

Raises a NotImpError exception to indicate that the enclosed function is not implemented yet or not available on this platform.

void **rb_exit**(int status)

Exits Ruby with the given *status*. Raises a SystemExit exception and calls registered exit functions and finalizers.

void **rb_warn**(const char *fmt, ...)

Unconditionally issues a warning message to standard error. The given string *fmt* and remaining arguments are interpreted as with printf.

void **rb_warning**(const char *fmt, ...)

Conditionally issues a warning message to standard error if Ruby was invoked with the -w flag. The given string *fmt* and remaining arguments are interpreted as with printf.

API: Iterators

void **rb_iter_break**( )

Breaks out of the enclosing iterator block.

VALUE **rb_each**(VALUE obj)

Invokes the each method of the given *obj*.

VALUE **rb_yield**(VALUE arg)

Transfers execution to the iterator block in the current context, passing *arg* as an argument. Multiple values may be passed in an array.

int **rb_block_given_p**( )

Returns true if yield would execute a block in the current context—that is, if a code block was passed to the current method and is available to be called.

VALUE **rb_iterate**(VALUE (*method)(), VALUE args, VALUE (*block)(), VALUE arg2)
Invokes *method* with argument *args* and block *block*. A yield from that method will invoke *block* with the argument given to yield and a second argument *arg2*.

VALUE **rb_catch**(const char *tag, VALUE (*proc)(), VALUE value)
Equivalent to Ruby catch.

void **rb_throw**(const char *tag , VALUE value)
Equivalent to Ruby throw.

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 id)
Returns the instance variable with ID *id* 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 an @ prefix) in the given *obj* to *value*. Returns *value*.

VALUE **rb_ivar_set**(VALUE obj, ID id, VALUE value)
Sets the value of the instance variable with ID *id* 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 id, VALUE val, int unused)
Sets the class variable with ID *id* in the given *class* to *value*.

VALUE **rb_cvar_get**(VALUE class, ID id)
Returns the class variable with ID *id* from the given *class*.

int **rb_cvar_defined**(VALUE class, ID id)
Returns Qtrue if the class variable with ID *id* 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*.

API: Object Status

OBJ_TAINT(VALUE obj)
Marks the given *obj* as tainted.

int **OBJ_TAINTED**(VALUE obj)
Returns nonzero if the given *obj* is tainted.

OBJ_FREEZE(VALUE obj)
Marks the given *obj* as frozen.

int **OBJ_FROZEN**(VALUE obj)
Returns nonzero if the given *obj* is frozen.

SafeStringValue(VALUE str)
Raises SecurityError if current safe level > 0 and *str* is tainted or raises a TypeError if *str* is not a T_STRING or if $SAFE >= 4.

int **rb_safe_level**( )
Returns the current safe level.

void **rb_secure**(int level)
Raises SecurityError if *level* <= current safe level.

void **rb_set_safe_level**(int newlevel)
Sets the current safe level to *newlevel*.

API: Commonly Used Methods

VALUE **rb_ary_new**( )
Returns a new Array with default size.

VALUE **rb_ary_new2**(long length)
Returns a new Array of the given *length*.

VALUE **rb_ary_new3**(long length, ...)
Returns a new Array of the given *length* and populated with the remaining arguments.

VALUE **rb_ary_new4**(long length, VALUE *values)
Returns a new Array of the given *length* and populated with the C array *values*.

void **rb_ary_store**(VALUE self, long index, VALUE value)
Stores *value* at *index* in array *self*.

VALUE **rb_ary_push**(VALUE self, VALUE value)
Pushes *value* onto the end of array *self*. Returns *value*.

VALUE **rb_ary_pop**(VALUE self)
Removes and returns the last element from the array *self*.

VALUE **rb_ary_shift**(VALUE self)
Removes and returns the first element from the array *self*.

VALUE **rb_ary_unshift**(VALUE self, VALUE value)
Pushes *value* onto the front of array *self*. Returns *value*.

VALUE **rb_ary_entry**(VALUE self, long index)
Returns array *self*'s element at *index*.

int **rb_respond_to**(VALUE self, ID method)
Returns nonzero if *self* responds to *method*.

VALUE **rb_thread_create**(VALUE (*func)(), void *data)
Runs *func* in a new thread, passing *data* as an argument.

VALUE **rb_hash_new**( )
Returns a new, empty Hash.

VALUE **rb_hash_aref**(VALUE self, VALUE key)
Returns the element corresponding to *key* in *self*.

VALUE **rb_hash_aset**(VALUE self, VALUE key, VALUE value)
Sets the value for *key* to *value* in *self*. Returns *value*.

VALUE **rb_obj_is_instance_of**(VALUE obj, VALUE klass)
Returns Qtrue if *obj* is an instance of *klass*.

VALUE **rb_obj_is_kind_of**(VALUE obj, VALUE klass)
Returns Qtrue if *klass* is the class of *obj* or *class* is one of the superclasses of the class of *obj*.

VALUE **rb_str_new**(const char *src, long length)
Returns a new String initialized with *length* characters from *src*.

VALUE **rb_str_new2**(const char *src)
Returns a new String initialized with the null-terminated C string *src*.

VALUE **rb_str_dup**(VALUE str)
Returns a new String object duplicated from *str*.

VALUE **rb_str_cat**(VALUE self, const char *src, long length)
Concatenates *length* characters from the string *src* onto the String *self*. Returns *self*.

VALUE **rb_str_concat**(VALUE self, VALUE other)
Concatenates *other* onto the String *self*. Returns *self*.

VALUE **rb_str_split**(VALUE self, const char *delim)
Returns an array of String objects created by splitting *self* on *delim*.

MKMF Reference

Module **mkmf** **require** "mkmf"

To build an extension, you create a program named extconf.rb, which may be as simple as this:

```
require 'mkmf'
create_makefile("Test")
```

When run, this script will produce a Makefile suited to the target platform. It also produces a log file, mkmf.log, which may help in diagnosing build problems.

mkmf contains several methods you can use to find libraries and include files and to set compiler flags.

mkmf takes configuration information from a variety of sources:

- The configuration used when Ruby was built
- The environment variable CONFIGURE_ARGS, a list of *key*=*value* pairs
- Command-line arguments of the form key=value or --key=value

You can examine the configuration by dumping the variable $configure_args:

```
% export CONFIGURE_ARGS="ruby=ruby18 --enable-extras"
% ruby -rmkmf -rpp -e 'pp $configure_args'  -- --with-cflags=-O3
{"--topsrcdir"=>".",
 "--topdir"=>"/Users/dave/Work/rubybook/tmp",
 "--enable-extras"=>true,
 "--with-cflags"=>"-O3",
 "--ruby"=>"ruby18"}
```

The following configuration options are recognized:

CFLAGS
Flags passed to the C compiler (overridden by --with-cflags).

CPPFLAGS
Flags passed to the C++ compiler (overridden by --with-cppflags).

curdir
Sets the global $curdir, which may be used inside the extconf.rb script. Otherwise, has no effect.

disable-xxx
Disables extension-specific option *xxx*.

enable-xxx
Enables extension-specific option *xxx*.

LDFLAGS
Flags passed to the linker (overridden by --with-ldlags).

ruby
Sets the name and/or path of the Ruby interpreter used in the Makefile.

srcdir
Sets the path to the source directory in the Makefile.

with-cflags
Flags passed to the C compiler. Overrides the CFLAGS environment variable.

with-cppflags
Flags passed to the C++ compiler. Overrides the CPPFLAGS environment variable.

with-ldflags
Flags passed to the linker compiler. Overrides the LDFLAGS environment variable.

with-make-prog
Sets the name of the make program. If running on Windows, the choice of make program affects the syntax of the generated Makefile (nmake vs. Borland make).

with-xxx-{dir|include|lib}
Controls where the dir_config method looks.

Instance methods

create_makefile

create_makefile(*target*, *srcprefix*=nil)

Creates a Makefile for an extension named *target*. The *srcprefix* can override the default source directory. If this method is not called, no Makefile is created.

dir_config

dir_config(*name*)

Looks for directory configuration options for *name* given as arguments to this program or to the original build of Ruby. These arguments may be one of the following:

--with-*name*-dir=*directory*
--with-*name*-include=*directory*
--with-*name*-lib=*directory*

The given directories will be added to the appropriate search paths (include or link) in the Makefile.

enable_config enable_config(*name*, *default*=nil) → true or false or *default*

Tests for the presence of an --enable-*name* or --disable-*name* option. Returns true if the enable option is given, false if the disable option is given, and the default value otherwise.

find_library find_library(*name*, *function*, ⟨ *path* ⟩+) → true or false

Same as have_library but will also search in the given directory paths.

have_func have_func(*function*) → true or false

If the named function exists in the standard compile environment, adds the directive -D HAVE_*FUNCTION* to the compile command in the Makefile and returns true.

have_header have_header(*header*) → true or false

If the given header file can be found in the standard search path, adds the directive -D HAVE_*HEADER* to the compile command in the Makefile and returns true.

have_library have_library(*library*, *function*) → true or false

If the given function exists in the named library, which must exist in the standard search path or in a directory added with dir_config, adds the library to the link command in the Makefile and returns true.