



Ruby Hacking Guide

This is the home page of the project to translate into English the [Ruby Hacking Guide](#). The RHG is a book that explains how the ruby interpreter (the official C implementation of the [Ruby language](#)) works internally.

To fully understand it, you need a good knowledge of C and Ruby. The original book includes a Ruby tutorial (chapter 1), but it has not been translated yet, and we think there are more important chapters to translate first. So if you have not done it yet, you should read a book like the [Pickaxe](#) first.

Please note that this book was based on the source code of ruby 1.7.3 so there are a few small differences to the current version of ruby. However, these differences may make the source code simpler to understand and the Ruby Hacking Guide is a good starting point before looking into the ruby source code. The version of the source code used can be downloaded here: <http://i.loveruby.net/ja/rhg/ar/ruby-rhg.tar.gz>.

Many thanks to [RubyForge](#) for hosting us and to Minero AOKI for letting us translate his work.

Help us!

This translation is done during our free time, do not expect too much. The book is quite big (more than 500 pages) so we need help to translate it.

People who are good at Ruby, C and Japanese or English are needed. Those good at Japanese (native Japanese speakers are of course welcome) can help translate and those good at English (preferably native speakers) can help correct mistakes, and rewrite badly written parts... Knowing Ruby and C well is really a requirement because it helps avoiding many mistranslations and misinterpretations.

People good at making diagrams would also be helpful because there is quite a lot to redo and translators would rather spend their time translating instead of making diagrams.

So if you want to help us, join the [rhg-discussion mailing list](#) and introduce yourself (who you are, your skills, how much free time you have). You can of course just join the mailing list to see what 's going on. And do not hesitate to ask questions!

The preferred way to propose corrections/improvements is to send a patch (attached to the mail, not just in the body of the message) on the mailing list. The patch should be done against the text files in the SVN repository (http://rubyforge.org/scm/?group_id=1387).

The RubyForge project page is <http://rubyforge.org/projects/rhg>.

Table of contents

Some chapters are previews. It means they have not been fully reviewed, some diagrams may be missing and some sentences may be a little rough. But it also means they are in open review, so do not hesitate to post suggestions on the mailing list.

Preface

Introduction

Part 1: Objects

Chapter 1: Ruby language minimum

[Chapter 2: Objects](#)

[Chapter 3: Names and name tables](#)

[Chapter 4: Classes and modules](#)

Chapter 5: Garbage collection

[Chapter 6: Variables and constants](#)

Chapter 7: Security

Part 2: Syntax analysis

Chapter 8: Ruby language details

Chapter 9: yacc crash course

Chapter 10: Parser

Chapter 11: Context-dependent scanner

Chapter 12: Syntax tree construction

Part 3: Evaluation

Chapter 13: Structure of the evaluator

Chapter 14: Context

Chapter 15: Methods

Chapter 16: Blocks

Chapter 17: Dynamic evaluation

Part 4: Around the evaluator

Chapter 18: Loading

Chapter 19: Threads


Final chapter: Ruby 's future

The original work is Copyright © 2002 - 2004 Minero AOKI.

Translated by Vincent ISAMBART and Clifford Escobar CAOILE



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#).



Chapter 2: Objects

Structure of Ruby objects



Guideline

Starting from this chapter we will explore the ruby source code, starting by studying the declaration of objects structures.

What are the required conditions to make sure objects can exist? Many explanations can be given but in reality there are three conditions that must be obeyed:

1. Being able to differentiate itself from the rest (having an identity)
2. Being able to reply to requests (methods)
3. Keeping an internal state (instance variables)

In this chapter, we are going to confirm these three features one by one.

The most interesting file in this quest will be `ruby.h`, but we will also briefly look at other files such as `object.c`, `class.c` or `variable.c`.



Structure of VALUE and objects

In ruby, the contents of an object is expressed by a C structure, always handled via a pointer. A different kind of structure is used for each class, but the pointer type will always be `VALUE` (figure 1).

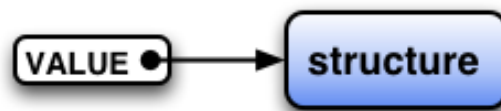


Figure 1: VALUE and structure

Here is the definition of VALUE:

VALUE

```
71 typedef unsigned long VALUE;
```

(ruby.h)

In practice, a VALUE must be casted to different types of structure pointer. Therefore if an unsigned long and a pointer have a different size, ruby will not work well. Strictly speaking, it will not work for pointer types bigger than sizeof(unsigned long). Fortunately, no recent machine feature this capability, even if some time ago there were quite a few of them.

Several structures are available according to object classes:

struct RObject	all things for which none of the following applies
struct RClass	class object
struct RFloat	small numbers
struct RString	string
struct RArray	array
struct RRegexp	regular expression
struct RHash	hash table
struct RFile	IO, File, Socket, etc...
struct RData	all the classes defined at C level, except the ones mentioned above
struct RStruct	Ruby 's Struct class

struct
RBignum big integers

For example, for an string object, struct RString is used, so we will have something like the following.



Figure 2: String object

Let 's look at the definition of a few object structures.

Examples of object structure

```

/* structure for ordinary objects */
295 struct RObject {
296     struct RBasic basic;
297     struct st_table *iv_tbl;
298 };

/* structure for strings (instance of String) */
314 struct RString {
315     struct RBasic basic;
316     long len;
317     char *ptr;
318     union {
319         long capa;
320         VALUE shared;
321     } aux;
322 };

/* structure for arrays (instance of Array) */
324 struct RArray {
325     struct RBasic basic;
326     long len;
327     union {
328         long capa;
  
```

```

329  VALUE shared;
330  } aux;
331  VALUE *ptr;
332 };

```

(ruby.h)

Before looking at every one of them in detail, let ' s begin with something more general.

First, as VALUE is defined as unsigned long, it must be casted before being used. That ' s why Rxxxx() macros have been made for each object structure. For example, for struct RString there is RSTRING(), for struct RArray there is RARRAY(), etc... These macros are used like this:

```

VALUE str = ....;
VALUE arr = ....;
RSTRING(str)->len; /* ((struct RString*)str)->len */
RARRAY(arr)->len; /* ((struct RArray*)arr)->len */

```

Another important point to mention is that all object structures start with a member basic of type struct RBasic. As a result, whatever the type of structure pointed by VALUE, if you cast this VALUE to struct RBasic*, you will be able to access the content of basic.

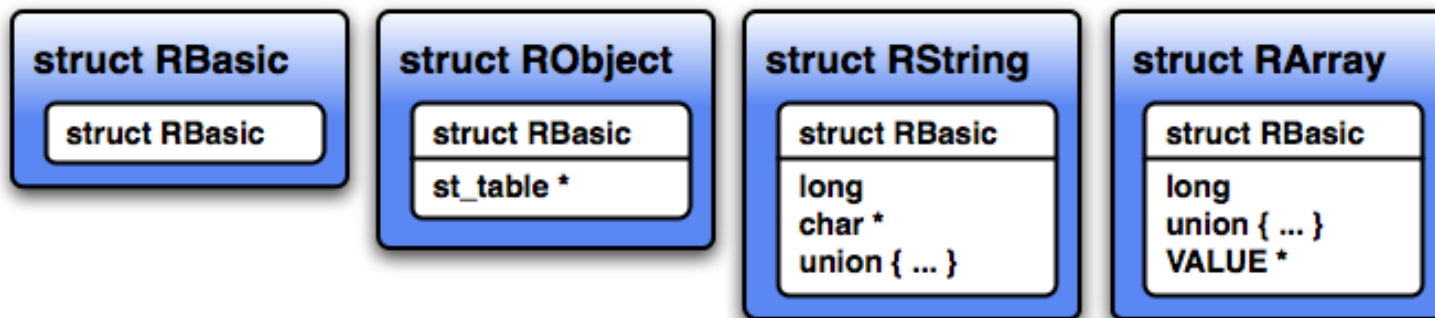


Figure 3: struct RBasic

You guessed that struct RBasic has been designed to contain some important information shared by all object structures. The definition of struct RBasic is the following:

```
struct RBasic
```

```

290 struct RBasic {
291     unsigned long flags;
292     VALUE klass;
293 };

```

(ruby.h)

flags are multipurpose flags, mostly used to register the structure type (for instance struct RObject). The type flags are named T_XXXX, and can be obtained from a VALUE using the macro TYPE(). Here is an example:

```

VALUE str;
str = rb_str_new(); /* creates a Ruby string (its structure is RString) */
TYPE(str);          /* the return value is T_STRING */

```

The names of these T_XXXX flags are directly linked to the corresponding type name, like T_STRING for struct RString and T_ARRAY for struct RArray.

The other member of struct RBasic, klass, contains the class this object belongs to. As the klass member is of type VALUE, what is stored is (a pointer to) a Ruby object. In short, it is a class object.

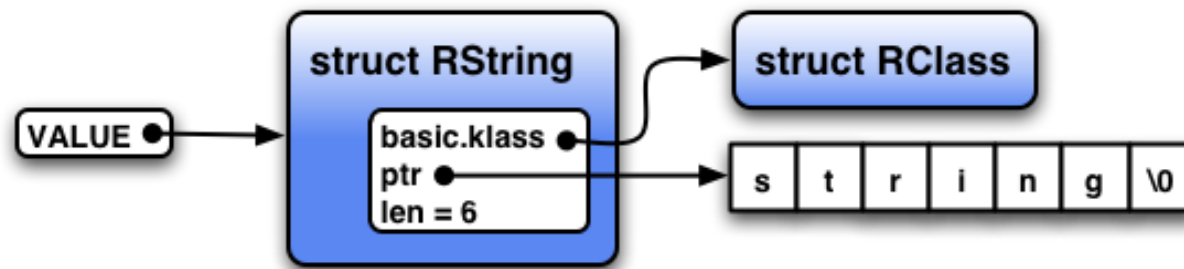


Figure 4: object and class

The relation between an object and its class will be detailed in the “ Methods ” section of this chapter.

By the way, the name of this member is not class to make sure it does not raise any conflict when the file is processed by a C++ compiler, as it is a reserved word.

About structure types

I said that the type of structure is stored in the flags member of struct Basic. But why do we have to store the type of structure? It ' s to be able to handle all different types of structure via VALUE. If you cast a pointer to a structure to VALUE, as the type information does not remain, the compiler won ' t be able to help. Therefore we have to manage the type ourselves. That ' s the consequence of being able to handle all the structure types in a unified way.

OK, but the used structure is defined by the class so why are the structure type and class are stored separately? Being able to find the structure type from the class should be enough. There are two reasons for not doing this.

The first one is (I ' m sorry for contradicting what I said before), in fact there are structures that do not have a struct RBasic (i.e. they have no klass member). For example struct RNode that will appear in the second part of the book. However, flags is guaranteed to be in the beginning members even in special structures like this. So if you put the type of structure in flags, all the object structures can be differentiated in one unified way.

The second reason is that there is no one-to-one correspondence between class and structure. For example, all the instances of classes defined at the Ruby level use struct RObject, so finding a structure from a class would require to keep the correspondence between each class and structure. That ' s why it ' s easier and faster to put the information about the type in the structure.

The use of basic.flags

As limiting myself to saying that basic.flags is used for different things including the type of structure makes me feel bad, here ' s a general illustration for it (figure 5). There is no need to understand everything right away, I just wanted to show its uses while it was bothering me.

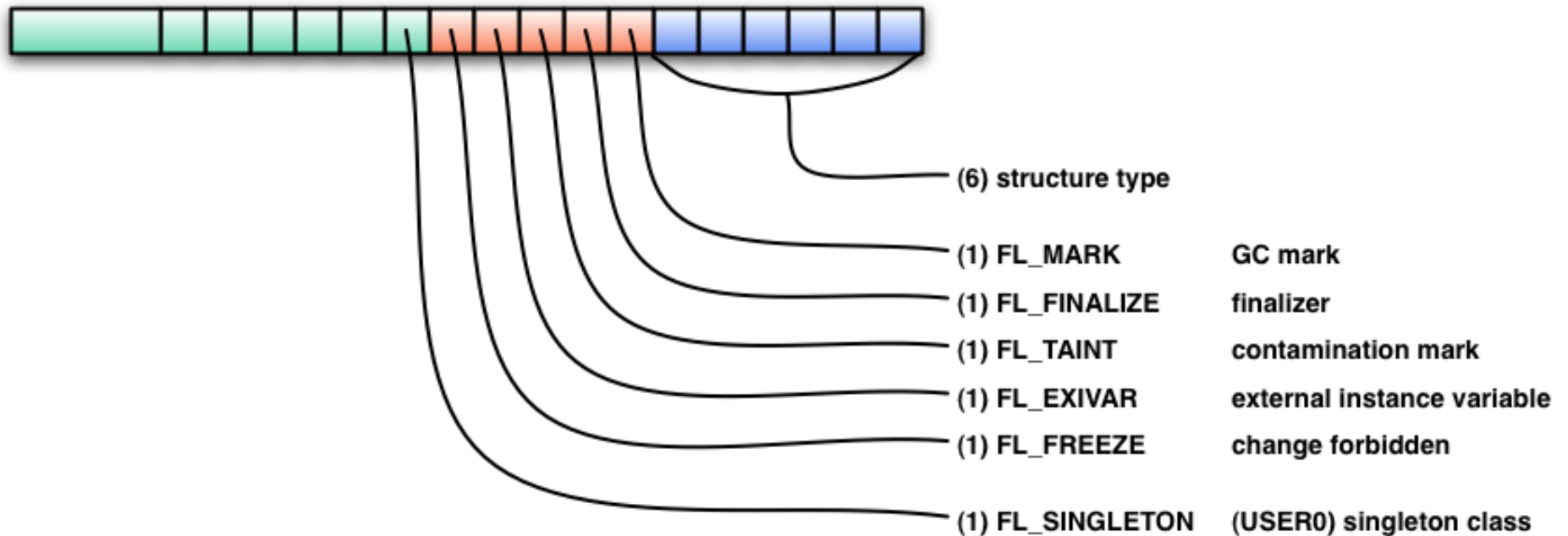


Figure 5: Use of flags

When looking at the diagram, it looks like that 21 bits are not used on 32 bit machines. On these additional bits, the flags FL_USER0 to FL_USER8 are defined, and are used for a different purpose for each structure. In the diagram I also put FL_USER0 (FL_SINGLETON) as an example.

Objects embedded in VALUE

As I said, VALUE is an unsigned long. As VALUE is a pointer, it may look like void* would also be all right, but there is a reason for not doing this. In fact, VALUE can also not be a pointer. The 6 cases for which VALUE is not a pointer are the following:

1. small integers
2. symbols
3. true
4. false
5. nil
6. Qundef

I ' ll explain them one by one.

Small integers

As in Ruby all data are objects, integers are also objects. However, as there are lots of different instances of integers, expressing them as structures would risk slowing down execution. For example, when incrementing from 0 to 50000, just for this creating 50000 objects would make us hesitate.

That 's why in ruby, to some extent, small integers are treated specially and embedded directly into VALUE. “ small ” means signed integers that can be held in $\text{sizeof(VALUE)} * 8 - 1$ bits. In other words, on 32 bits machines, the integers have 1 bit for the sign, and 30 bits for the integer part. Integers in this range will belong to the Fixnum class and the other integers will belong to the Bignum class.

Then, let 's see in practice the INT2FIX() macro that converts from a C int to a Fixnum, and confirm that Fixnum are directly embedded in VALUE.

INT2FIX

```
123 #define INT2FIX(i) ((VALUE)(((long)(i))<<1 | FIXNUM_FLAG))
122 #define FIXNUM_FLAG 0x01
```

(ruby.h)

In brief, shift 1 bit to the right, and bitwise or it with 1.

```
0110100001000 before
                  conversion
1101000010001 after conversion
```

That means that Fixnum as VALUE will always be an odd number. On the other hand, as Ruby object structures are allocated with malloc(), they are generally arranged on addresses multiple of 4. So they do not overlap with the values of Fixnum as VALUE.

Also, to convert int or long to VALUE, we can use macros like INT2NUM() or LONG2NUM(). Any conversion macro XXXX2XXXX with a name containing NUM can manage both Fixnum and Bignum. For example if INT2NUM() can 't convert an integer into a Fixnum, it will automatically convert it to Bignum. NUM2INT() will convert both Fixnum and Bignum to int. If the number can 't fit in an int, an exception will be raised, so there is not need to check the value range.

Symbols

What are symbols?

As this question is quite troublesome to answer, let 's start with the reasons why symbols were necessary. First, let 's start with the ID type used inside ruby. It 's like this:

ID

```
72 typedef unsigned long ID;
```

(ruby.h)

This ID is a number having a one-to-one association with a string. However, in this world it 's not possible to have an association between all strings and a numerical value. That 's why they are limited to the one to one relationships inside one ruby process. I 'll speak of the method to find an ID in the next chapter “ Names and name tables ” .

In language implementations, there are a lot of names to handle. Method names or variable names, constant names, file names in class names... It 's troublesome to handle all of them as strings (char*), because of memory management and memory management and memory management... Also, lots of comparisons would certainly be necessary, but comparing strings character by character will slow down the execution. That 's why strings are not handled directly, something will be associated and used instead. And generally “ something ” will be integers, as they are the simplest to handle.

These ID are found as symbols in the Ruby world. Up to ruby 1.4, the values of ID where converted to Fixnum, but used as symbols. Even today these values can be obtained using Symbol#to_i. However, as real use results came piling up, it was understood that making Fixnum and Symbol the same was not a good idea, so since 1.6 an independent class Symbol has been created.

Symbol objects are used a lot, especially as keys for hash tables. That 's why Symbol, like Fixnum, was made stored in VALUE. Let 's look at the ID2SYM() macro converting ID to Symbol object.

ID2SYM

```
158 #define SYMBOL_FLAG 0x0e
160 #define ID2SYM(x) ((VALUE)((((long)(x))<<8|SYMBOL_FLAG))
```

(ruby.h)

When shifting 8 bits left, `x` becomes a multiple of 256, that means a multiple of 4. Then after with a bitwise or (in this case it 's the same as adding) with 0x0e (14 in decimal), the `VALUE` expressing the symbol is not a multiple of 4. Or even an odd number. So it does not overlap the range of any other `VALUE`. Quite a clever trick.

Finally, let 's see the reverse conversion of `ID2SYM()`, `SYM2ID()`.

`SYM2ID()`

```
161 #define SYM2ID(x) RSHIFT((long)x,8)
```

(ruby.h)

`RSHIFT` is a bit shift to the right. As right shift may keep or not the sign depending of the platform, it became a macro.

true false nil

These three are Ruby special objects. `true` and `false` represent the boolean values. `nil` is an object used to denote that there is no object. Their values at the C level are defined like this:

`true false nil`

```
164 #define Qfalse 0    /* Ruby's false */
165 #define Qtrue  2    /* Ruby's true  */
166 #define Qnil   4    /* Ruby's nil  */
```

(ruby.h)

This time it 's even numbers, but as 0 or 2 can 't be used by pointers, they can 't overlap with other `VALUE`. It 's because usually the first bloc of virtual memory is not allocated, to make the programs dereferencing a `NULL` pointer crash.

And as `Qfalse` is 0, it can also be used as `false` at C level. In practice, in ruby, when a function returns a boolean value, it 's often made to return an int or `VALUE`, and returns `Qtrue/Qfalse`.

For `Qnil`, there is a macro dedicated to check if a `VALUE` is `Qnil` or not, `NIL_P()`.

NIL_P()

```
170 #define NIL_P(v) ((VALUE)(v) == Qnil)

(ruby.h)
```

The name ending with p is a notation coming from Lisp denoting that it is a function returning a boolean value. In other words, NIL_P means “ is the argument nil? ”. It seems the “ p ” character comes from “ predicate ”. This naming rule is used at many different places in ruby.

Also, in Ruby, false and nil are false and all the other objects are true. However, in C, nil (Qnil) is true. That ’ s why in C a Ruby-style macro, RTEST(), has been created.

RTEST()

```
169 #define RTEST(v) (((VALUE)(v) & ~Qnil) != 0)

(ruby.h)
```

As in Qnil only the third lower bit is 1, in ~Qnil only the third lower bit is 0. Then only Qfalse and Qnil become 0 with a bitwise and.

!=0 has be added to be certain to only have 0 or 1, to satisfy the requirements of the glib library that only wants 0 or 1 ([\[ruby-dev:11049\]](#)).

By the way, what is the ‘ Q ’ of Qnil? ‘ R ’ I would have understood but why ‘ Q ’ ? When I asked, the answer was “ Because it ’ s like that in Emacs ”. I did not have the fun answer I was expecting...

Qundef

Qundef

```
167 #define Qundef 6          /* undefined value for placeholder */

(ruby.h)
```

This value is used to express an undefined value in the interpreter. It can ’ t be found at all at the Ruby level.

Methods

I already brought up the three important points of a Ruby object, that is having an identity, being able to call a method, and keeping data for each instance. In this section, I ' ll explain in a simple way the structure linking objects and methods.

struct RClass

In Ruby, classes exist as objects during the execution. Of course. So there must be a structure for class objects. That structure is struct RClass. Its structure type flag is T_CLASS.

As class and modules are very similar, there is no need to differentiate their content. That ' s why modules also use the struct RClass structure, and are differentiated by the T_MODULE structure flag.

struct RClass

```
300 struct RClass {
301     struct RBasic basic;
302     struct st_table *iv_tbl;
303     struct st_table *m_tbl;
304     VALUE super;
305 };
```

(ruby.h)

First, let ' s focus on the m_tbl (Method TaBLe) member. struct st_table is an hashtable used everywhere in ruby. Its details will be explained in the next chapter “ Names and name tables ” , but basically, it is a table mapping names to objects. In the case of m_tbl, it keeps the correspondence between the name (ID) of the methods possessed by this class and the methods entity itself.

The fourth member super keeps, like its name suggests, the superclass. As it ' s a VALUE, it ' s (a pointer to) the class object of the superclass. In Ruby there is only one class that has no superclass (the root class): Object.

However I already said that all Object methods are defined in the Kernel module, Object just includes it. As modules are functionally similar to multiple inheritance, it

may seem having just super is problematic, but but in ruby some clever changes are made to make it look like single inheritance. The details of this process will be explained in the fourth chapter “Classes and modules”.

Because of this, super of the structure of Object points to struct RClass of the Kernel object. Only the super of Kernel is NULL. So contrary to what I said, if super is NULL, this RClass is the Kernel object (figure 6).

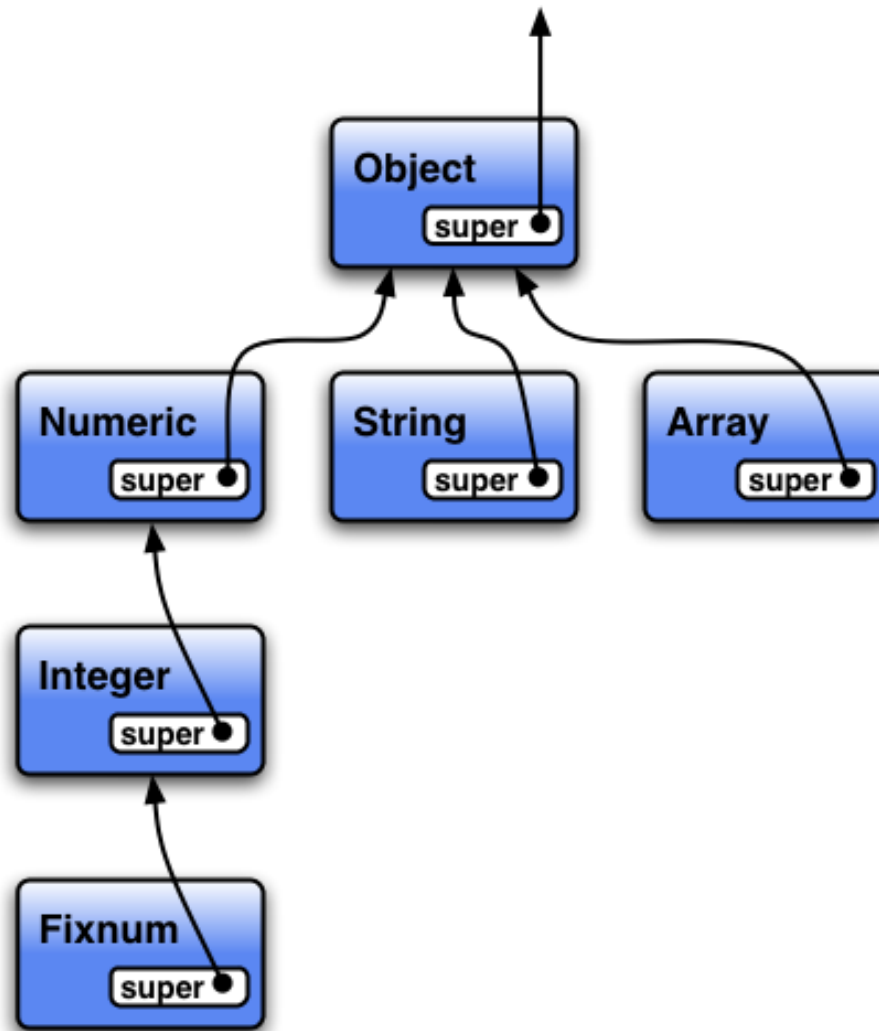


Figure 6: Class tree at the C level

Methods search

With classes structured like this, you can easily imagine the method call process. The `m_tbl` of the object 's class is searched, and if the method was not found, the `m_tbl` of super is searched, and so on. If there is no more super, that is to say the method was not found even in `Object`, then it must not be defined.

The sequential search process in `m_tbl` is done by `search_method()`.

`search_method()`

```

256 static NODE*
257 search_method(klass, id, origin)
258     VALUE klass, *origin;
259     ID id;
260 {
261     NODE *body;
262
263     if (!klass) return 0;
264     while (!st_lookup(RCLASS(klass)->m_tbl, id, &body)) {
265         klass = RCLASS(klass)->super;
266         if (!klass) return 0;
267     }
268
269     if (origin) *origin = klass;
270     return body;
271 }

```

(eval.c)

This function searches the method named `id` in the class object `klass`.

`RCLASS(value)` is the macro doing:

```
((struct RClass*)(value))
```

`st_lookup()` is a function that searches in `st_table` the value corresponding to a key. If the value is found, the function returns true and puts the found value at the address given in third parameter (`&body`).

Nevertheless, doing this search each time whatever the circumstances would be too slow. That ' s why in reality, once called, a method is cached. So starting from the second time it will be found without following super one by one. This cache and its search will be seen in the 15th chapter “ Methods ” .

Instance variables

In this section, I will explain the implementation of the third essential condition, instance variables.

rb_ivar_set()

Instance variables are what allows each object to store characteristic data. Having it stored in the object itself (i.e. in the object structure) may seem all right but how is it in practice? Let ' s look at the function `rb_ivar_set()` that puts an object in an instance variable.

`rb_ivar_set()`

```

/* write val in the id instance of obj */
984 VALUE
985 rb_ivar_set(obj, id, val)
986     VALUE obj;
987     ID id;
988     VALUE val;
989 {
990     if (!OBJ_TAINTED(obj) && rb_safe_level() >= 4)
991         rb_raise(rb_eSecurityError,
992             "Insecure: can't modify instance variable");
992     if (OBJ_FROZEN(obj)) rb_error_frozen("object");
993     switch (TYPE(obj)) {
994     case T_OBJECT:
995     case T_CLASS:
996     case T_MODULE:
997         if (!ROBJECT(obj)->iv_tbl)
998             ROBJECT(obj)->iv_tbl = st_init_numtable();
998         st_insert(ROBJECT(obj)->iv_tbl, id, val);
999         break;
1000     default:
1001         generic_ivar_set(obj, id, val);

```

```

1002     break;
1003 }
1004 return val;
1005 }

```

(variable.c)

`rb_raise()` and `rb_error_frozen()` are both error checks. Error checks are necessary, but it ' s not the main part of the treatment, so you should ignore them at first read.

After removing error treatment, only the switch remains, but this

```

switch (TYPE(obj)) {
  case T_aaaa:
  case T_bbbb:
    ...
}

```

form is characteristic of ruby. `TYPE()` is the macro returning the type flag of the object structure (`T_OBJECT`, `T_STRING`, etc.). In other words as the type flag is an integer constant, we can branch depending on it with a switch. Fixnum or Symbol do not have structures, but inside `TYPE()` a special treatment is done to properly return `T_FIXNUM` and `T_SYMBOL`, so there ' s no need to worry.

Well, let ' s go back to `rb_ivar_set()`. It seems only the treatments of `T_OBJECT`, `T_CLASS` and `T_MODULE` are different. These 3 have been chosen on the basis that their second member is `iv_tbl`. Let ' s confirm it in practice.

Structures whose second member is `iv_tbl`

```

/* TYPE(val) == T_OBJECT */
295 struct RObject {
296     struct RBasic basic;
297     struct st_table *iv_tbl;
298 };

/* TYPE(val) == T_CLASS or T_MODULE */
300 struct RClass {
301     struct RBasic basic;
302     struct st_table *iv_tbl;

```

```

303 struct st_table *m_tbl;
304 VALUE super;
305 };

```

(ruby.h)

iv_tbl is the Instance Variable TaBLe. It stores instance variable names and their corresponding value.

In rb_ivar_set(), let 's look again the code for the structures having iv_tbl.

```

if (!ROBJECT(obj)->iv_tbl)
  ROBJECT(obj)->iv_tbl = st_init_numtable();
st_insert(ROBJECT(obj)->iv_tbl, id, val);
break;

```

ROBJECT() is a macro that casts a VALUE into a struct RObject*. It 's possible that obj points to a struct RClass, but as we 're only going to access the second member no problem will occur.

st_init_numtable() is a function creating a new st_table. st_insert() is a function doing associations in a st_table.

In conclusion, this code does the following: if iv_tbl does not exist, it creates it, then stores the [variable name object] association.

Warning: as struct RClass is a class object, this instance variable table is for the use of the class object itself. In Ruby programs, it corresponds to something like the following:

```

class C
  @ivar = "content"
end

```

generic_ivar_set()

For objects for which the structure used is not T_OBJECT, T_MODULE, or T_CLASS, what happens when modifying an instance variable?

rb_ivar_set() in the case there is no iv_tbl

```

1000 default:
1001   generic_ivar_set(obj, id, val);
1002   break;

```

(variable.c)

The control is transferred to `generic_ivar_set()`. Before looking at this function, let 's first explain its general idea.

Structures that are not `T_OBJECT`, `T_MODULE` or `T_CLASS` do not have an `iv_tbl` member (the reason why they do not have it will be explained later). However, a method linking an instance to a struct `st_table` would allow instances to have instance variables. In ruby, this was solved by using a global `st_table`, `generic_iv_table` (figure 7) for these associations.

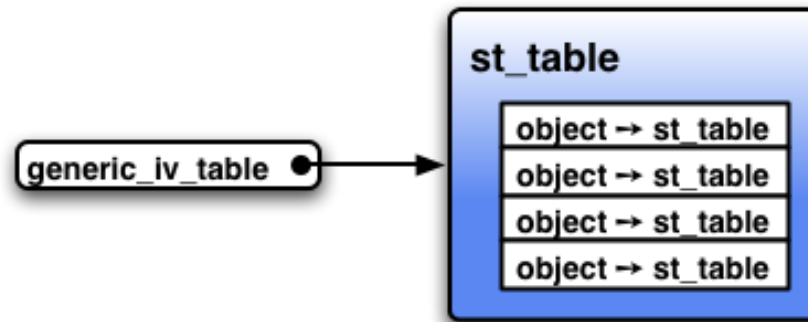


Figure 7: `generic_iv_table`

Let 's see this in practice.

```
generic_ivar_set()
```

```

801 static st_table *generic_iv_tbl;

830 static void
831 generic_ivar_set(obj, id, val)
832   VALUE obj;
833   ID id;
834   VALUE val;
835 {
836   st_table *tbl;

```

```

837
838  /* for the time being you should ignore this */
839  if (rb_special_const_p(obj)) {
840      special_generic_ivar = 1;
841  }
842  /* initialize generic_iv_tbl if it does not exist */
843  if (!generic_iv_tbl) {
844      generic_iv_tbl = st_init_numtable();
845  }
846  /* the treatment itself */
847  if (!st_lookup(generic_iv_tbl, obj, &tbl)) {
848      FL_SET(obj, FL_EXIVAR);
849      tbl = st_init_numtable();
850      st_add_direct(generic_iv_tbl, obj, tbl);
851      st_add_direct(tbl, id, val);
852      return;
853  }
854  st_insert(tbl, id, val);
855 }

```

(variable.c)

`rb_special_const_p()` is true when its parameter is not a pointer. However, as this if part requires knowledge of the garbage collector, we ' ll skip it for now. I ' d like you to check it again after reading the chapter 5 “ Garbage collection ” .

`st_init_numtable()` already appeared some time ago. It creates a new hash table.

`st_lookup()` searches a value corresponding to a key. In this case it searches for what ' s attached to `obj`. If an attached value can be found, the whole function returns true and stores the value at the address (`&tbl`) given as third parameter. In short, `!st_lookup(...)` can be read “ if a value can ' t be found ” .

`st_insert()` was also already explained. It stores a new association in a table.

`st_add_direct()` is similar to `st_insert()`, but the part before adding the association that checks if the key was already stored or not is different. In other words, in the case of `st_add_direct()`, if a key already registered is being used, two associations linked to this same key will be stored. `st_add_direct()` can be used when the check for existence has already been done, as is the case here, or when a new table has just been created.

`FL_SET(obj, FL_EXIVAR)` is the macro that sets the `FL_EXIVAR` flag in the `basic.flags` of `obj`. The `basic.flags` flags are all named `FL_xxxx` and can be set using `FL_SET()`. These flags can be unset with `FL_UNSET()`. The `EXIVAR` from `FL_EXIVAR` seems to be the abbreviation of `EXternal Instance VARiable`.

The setting of these flags is done to speed up the reading of instance variables. If `FL_EXIVAR` is not set, even without searching in `generic_iv_tbl`, we directly know if the object has instance variables. And of course a bit check is way faster than searching a struct `st_table`.

Gaps in structures

Now you should understand how the instance variables are stored, but why are there structures without `iv_tbl`? Why is there no `iv_tbl` in struct `RString` or struct `RArray`? Couldn't `iv_tbl` be part of `RBasic`?

Well, this could have been done, but there are good reasons why it was not. As a matter of fact, this problem is deeply linked to the way ruby manages objects.

In ruby, memory used by for example string data (`char[]`) is directly allocated using `malloc()`. However, the object structures are handled in a particular way. ruby allocates them by clusters, and then distribute them from these clusters. As at allocation time the diversity of types (and sizes) of structures is difficult to handle, a type (union) that combines all structures `RVALUE` was declared and an array of this type is managed. As this type's size is the same as the biggest one of its members, if there is only one big structure, there is a lot of unused space. That's why doing as much as possible to regroup structures of similar size is desirable. The details about `RVALUE` will be explained in chapter 5 "Garbage collection".

Generally the most used structure is struct `RString`. After that, in programs there are struct `RArray` (array), `RHash` (hash), `RObject` (user defined object), etc. However, this struct `RObject` only uses the space of struct `RBasic` + 1 pointer. On the other hand, struct `RString`, `RArray` and `RHash` take the space of struct `RBasic` + 3 pointers. In other words, when putting a struct `RObject` in the shared entity, the space for 2 pointers is useless. And beyond that, if `RString` had 4 pointers, `RObject` would use less than half the size of the shared entity. As you would expect, it's wasteful.

So the received merit for `iv_tbl` is more or less saving memory and speeding up. Furthermore we do not know if it is used often or not. In the facts, `generic_iv_tbl` was not introduced before ruby 1.2, so it was not possible to use instance variables in `String` or `Array` at this time. Nevertheless it was not so much of a problem. Making large amounts of memory useless just for such a functionality looks stupid.

If you take all this into consideration, you can conclude that increasing the size of object structures does not do any good.

`rb_ivar_get()`

We saw the `rb_ivar_set()` function that sets variables, so let's see quickly how to get them.

```
rb_ivar_get()
```

```

960 VALUE
961 rb_ivar_get(obj, id)
962     VALUE obj;
963     ID id;
964 {
965     VALUE val;
966
967     switch (TYPE(obj)) {
968         /* (A) */
969         case T_OBJECT:
970         case T_CLASS:
971         case T_MODULE:
972             if (ROBJECT(obj)->iv_tbl &&
973                 st_lookup(ROBJECT(obj)->iv_tbl, id, &val))
974                 return val;
975             break;
976         /* (B) */
977         default:
978             if (FL_TEST(obj, FL_EXIVAR) || rb_special_const_p(obj))
979                 return generic_ivar_get(obj, id);
980             break;
981     }
982     /* (C) */
983     rb_warning("instance variable %s not initialized", rb_id2name(id));
984     return Qnil;
985 }

```

(variable.c)

The structure is strictly the same.

(A) For struct RObject or RClass, we search the variable in iv_tbl. As iv_tbl can also be NULL, we must check it before using it. Then if st_lookup() finds the relation, it returns true, so the whole if can be read as “ If the instance variable has been set, return its value ” .

(C) If no correspondence could be found, in other words if we read an instance variable that has not been set, we first leave the if then the switch. rb_warning() will then issue a warning and nil will be returned. That ’ s because you can read instance variables that have not been set in Ruby.

(B) On the other hand, if the structure is neither struct RObject nor RClass, the instance variable table is searched in generic_iv_tbl. What generic_ivar_get() does can

be easily guessed, so I won't explain it. I'd rather want you to focus on the if.

I already told you that `generic_ivar_set()` sets the `FL_EXIVAR` flag to make the check faster.

And what is `rb_special_const_p()`? This function returns true when its parameter `obj` does not point to a structure. As no structure means no `basic.flags`, no flag can be set, and `FL_xxxx()` will always return false. That's why these objects have to be treated specially.

Structures for objects

In this section we'll see simply, among object structures, what the important ones contain and how they are handled.

struct RString

`struct RString` is the structure for the instances of the `String` class and its subclasses.

`struct RString`

```

314 struct RString {
315     struct RBasic basic;
316     long len;
317     char *ptr;
318     union {
319         long capa;
320         VALUE shared;
321     } aux;
322 };

```

(ruby.h)

`ptr` is a pointer to the string, and `len` the length of that string. Very straightforward.

Rather than a string, Ruby's string is more a byte array, and can contain any byte including NUL. So when thinking at the Ruby level, ending the string with NUL does not mean anything. As C functions require NUL, for convenience the ending NUL is there, however, it is not included in `len`.

When dealing with a string coming from the interpreter or an extension library, you can write `RSTRING(str)->ptr` or `RSTRING(str)->len`, and access `ptr` and `len`. But there are some points to pay attention to.

1. you have to check before if `str` really points to a struct `RString`
2. you can read the members, but you must not modify them
3. you can 't store `RSTRING(str)->ptr` in something like a local variable and use it later

Why is that? First, there is an important software engineering principle: Don 't arbitrarily tamper with someone 's data. Interface functions are there for a reason. However, there are concrete reasons in ruby 's design why you should not do such things as consulting or storing a pointer, and that 's related to the fourth member `aux`. However, to explain properly how to use `aux`, we have to explain first a little more of Ruby 's strings ' characteristics.

Ruby 's strings can be modified (are mutable). By mutable I mean after the following code:

```
s = "str"      # create a string and assign it to s
s.concat("ing") # append "ing" to this string object
p(s)          # show the string
```

the content of the object pointed by `s` will become " string ". It 's different from Java or Python string objects. Java 's `StringBuffer` is closer.

And what 's the relation? First, mutable means the length (`len`) of the string can change. We have to increase or decrease the allocated memory size each time the length changes. We can of course use `realloc()` for that, but generally `malloc()` and `realloc()` are heavy operations. Having to `realloc()` each time the string changes is a huge burden.

That 's why the memory pointed by `ptr` has been allocated with a size a little bigger than `len`. Because of that, if the added part can fit into the remaining memory, it 's taken care of without calling `realloc()`, so it 's faster. The structure member `aux.capa` contains the length including this additional memory.

So what is this other `aux.shared`? It 's to speed up the creation of literal strings. Have a look at the following Ruby program.

```
while true do # repeat indefinitely
  a = "str"   # create a string with "str" as content and assign it to a
  a.concat("ing") # append "ing" to the object pointed by a
  p(a)        # show "string"
end
```

Whatever the number of times you repeat the loop, the fourth line ' s p has to show "string". That ' s why the code "str" should create, each time, a string object holding a different char[]. However, if no change occurs for a lot of strings, useless copies of char[] can be created many times. It would be better to share one common char[].

The trick that allows this to happen is aux.shared. String objects created with a literal use one shared char[]. When a change occurs, the string is copied in unshared memory, and the change is done on this new copy. This technique is called “ copy-on-write ” . When using a shared char[], the flag ELTS_SHARED is set in the object structure ' s basic.flags, and aux.shared contains the original object. ELTS seems to be the abbreviation of ELeMenTS.

But, well, let ' s return to our talk about RSTRING(str)->ptr. Even if consulting the pointer is OK, you must not modify it, first because the value of len or capa will no longer agree with the content, and also because when modifying strings created as literals, aux.shared has to be separated.

To finish this section about RString, let ' s write some examples how to use it. str is a VALUE that points to RString.

```
RSTRING(str)->len;      /* length */
RSTRING(str)->ptr[0];   /* first character */
str = rb_str_new("content", 7); /* create a string with "content" as its content
                                the second parameter is the length */
str = rb_str_new2("content"); /* create a string with "content" as its content
                                its length is calculated with strlen() */
rb_str_cat2(str, "end"); /* Concatenate a C string to a Ruby string */
```

struct RArray

struct RArray is the structure for the instances of Ruby ' s array class Array.

struct RArray

```
324 struct RArray {
325     struct RBasic basic;
326     long len;
327     union {
328         long capa;
329         VALUE shared;
330     } aux;
331     VALUE *ptr;
332 };
```

(ruby.h)

Except for the type of `ptr`, this structure is almost the same as `struct RString`. `ptr` points to the content of the array, and `len` is its length. `aux` is exactly the same as in `struct RString`. `aux.capa` is the “ real ” length of the memory pointed by `ptr`, and if `ptr` is shared, `aux.shared` stores the shared original array object.

From this structure, it ’ s clear that Ruby ’ s `Array` is an array and not a list. So when the number of elements changes in a big way, a `realloc()` must be done, and if an element must be inserted at an other place than the end, a `memmove()` will occur. But even if we do it, it ’ s moving so fast it ’ s really impressive on current machines.

That ’ s why the way to access it is similar to `RString`. You can consult `RARRAY(arr)->ptr` and `RARRAY(arr)->len` members, but can ’ t set them, etc., etc. We ’ ll only look at simple examples:

```
/* manage an array from C */
VALUE ary;
ary = rb_ary_new();      /* create an empty array */
rb_ary_push(ary, INT2FIX(9)); /* push a Ruby 9 */
RARRAY(ary)->ptr[0];      /* look what's at index 0 */
rb_p(RARRAY(ary)->ptr[0]); /* do p on ary[0] (the result is 9) */

# manage an array from Ruby
ary = [] # create an empty array
ary.push(9) # push 9
ary[0] # look what's at index 0
p(ary[0]) # do p on ary[0] (the result is 9)
```

struct RRegexp

It ’ s the structure for the instances of the regular expression class `Regexp`.

`struct RRegexp`

```
334 struct RRegexp {
335     struct RBasic basic;
336     struct re_pattern_buffer *ptr;
337     long len;
```

```
338 char *str;
339 };

(ruby.h)
```

ptr is the regular expression after compilation. str is the string before compilation (the source code of the regular expression), and len is this string ' s length.

As the Regexp object handling code doesn ' t appear in this book, we won ' t see how to use it. Even if you use it in extension libraries, as long as you do not want to use it a very particular way, the interface functions are enough.

struct RHash

struct RHash is the structure for Ruby ' s Hash objects.

```
struct RHash
```

```
341 struct RHash {
342     struct RBasic basic;
343     struct st_table *tbl;
344     int iter_lev;
345     VALUE ifnone;
346 };

(ruby.h)
```

It ' s a wrapper for struct st_table. st_table will be detailed in the next chapter “ Names and name tables ” .

ifnone is the value when a key does not have an attached value, its default is nil. iter_lev is to make the hashtable reentrant (multithread safe).

struct RFile

struct RFile is a structure for instances of the built-in IO class and its subclasses.

```
struct RFile
```

```

348 struct RFile {
349     struct RBasic basic;
350     struct OpenFile *fptr;
351 };

```

(ruby.h)

OpenFile

```

19 typedef struct OpenFile {
20     FILE *f;           /* stdio ptr for read/write */
21     FILE *f2;          /* additional ptr for rw pipes */
22     int mode;           /* mode flags */
23     int pid;           /* child's pid (for pipes) */
24     int lineno;        /* number of lines read */
25     char *path;         /* pathname for file */
26     void (*finalize) _((struct OpenFile*)); /* finalize proc */
27 } OpenFile;

```

(rubyio.h)

All members have been transferred in struct OpenFile. As there aren't many instances of IO objects, it's OK to do it like this. The purpose of each member is written in the comments. Basically, it's a wrapper around C's stdio.

struct RData

struct RData has a different tenor from what we saw before. It is the structure for implementation of extension libraries.

Of course structures for classes created in extension libraries as necessary, but as the types of these structures depend of the created class, it's impossible to know their size or structure in advance. That's why a “structure for managing a pointer to a user defined structure” has been created on ruby's side to manage this. This structure is struct RData.

struct RData

```

353 struct RData {
354     struct RBasic basic;

```

```

355 void (*dmark) _((void*));
356 void (*dfree) _((void*));
357 void *data;
358 };

```

(ruby.h)

data is a pointer to the user defined structure, dfree is the function used to free this structure, and dmark is the function for when the “ mark ” of the mark and sweep occurs.

Because explaining struct RData is still too complicated, for the time being let ’ s just look at its representation (figure 8). You ’ ll read a detailed explanation of its members in chapter 5 “ Garbage collection ” where there ’ ll be presented once again.

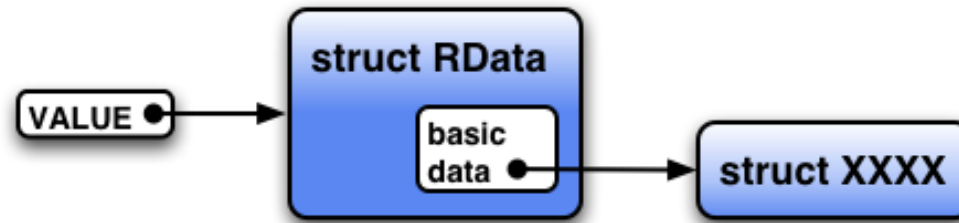


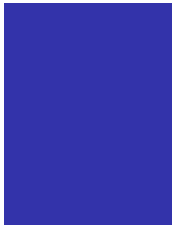
Figure 8: Representation of struct RData

The original work is Copyright © 2002 - 2004 Minero AOKI.

Translated by Vincent ISAMBART



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](https://creativecommons.org/licenses/by-nc-sa/2.5/).



Chapter 3: Names and Name Table

st_table

st_table has already been mentioned as a method table and instance table. In this chapter let us look at the detailed mechanism of the st_table.

Summary

I previously mentioned that the st_table is a hash table. What is a hash table? It is a data structure that records one-to-one relations, for example, variable name and its value, function name and its body, etc.

However, data structures other than hash tables can, of course, record one-to-one relations. For example, a list of following data structure will suffice for this purpose.

```
struct entry {
    ID key;
    VALUE val;
    struct entry *next; /* point to the next entry */
};
```

However, this method is slow. If the list contains a thousand items, in the worst case, it is necessary to traverse links a thousand times. In other words, in proportion to the number of elements, the search time increases. This is bad. Since ancient times, various speed improvement methods have been conceived. The hash table is one of those improved methods. In other words, the point is not that the hash table is necessary but that it can be made faster.

Now then, let us examine the st_table. But first, this library is not created by Matsumoto, rather:

st.c credits

```
1 /* This is a public domain general purpose hash table package
   written by Peter Moore @ UCB. */
```

```
(st.c)
```

as shown above.

By the way, when I searched Google and found another version, it mentioned that st_table is a contraction of “ STring TABLE ” . However, I find it contradictory that it has both “ general purpose ” and “ string ” aspects.

What is a hash table?

A hash table can be thought as the following: Let us think of an array with n items. For example, let us make $n=64$ (figure 1).



Figure 1: Array

Then let us specify a function f that takes a key and produces an integer i from 0 to $n-1$ (0-63). We call this f a hash function. f when given the same key always produces the i . For example, if we make the assumption that the key is limited to positive integers, then if the key is divided by 64 then the remainder will always fall between 0 and 63. This method of calculation can become function f .

When recording relationships, given a key, function f generates i , and place the value into index i of the array we have prepared. In other words, the index access into an array is very fast. Therefore the fundamental idea is to change the key into a integer.

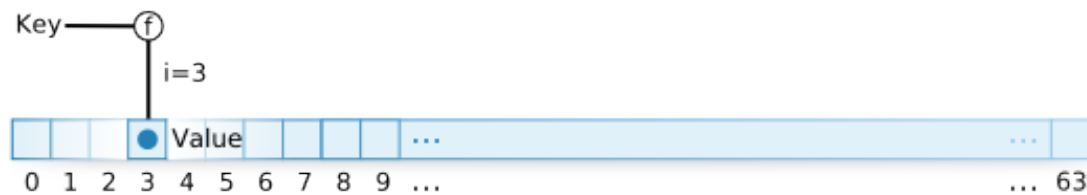


Figure 2: Array assignment

However, in the real world it isn't that easy. There is a critical problem with this idea. Because n is only 64, if there are more than 64 relationships to be recorded, it is certain that i will be the same for two different keys. It is also possible that with fewer than 64, the same thing can occur. For example, given the previous hash function "key % 64", keys 65 and 129 will have a hash value of 1. This is called a hash value collision. There are many ways to resolve a collision.

For example, if a collision occurs, then insert into the next element. This is called open addressing. (Figure 3).

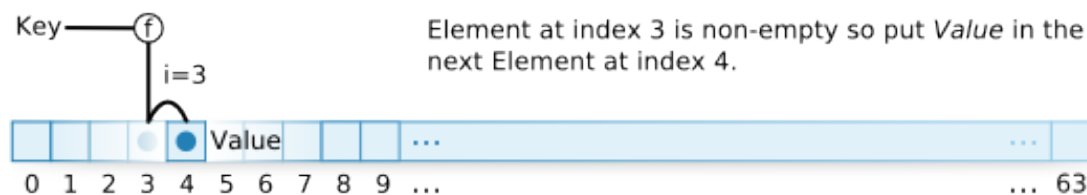


Figure 3: Open addressing

Other than using the array like this, there are other possible approaches, like using a pointer to a respective linked list in each element of the array. Then when a collision occurs, grow the linked list. This is called chaining. (Figure 4) `st_table` uses this chaining method.

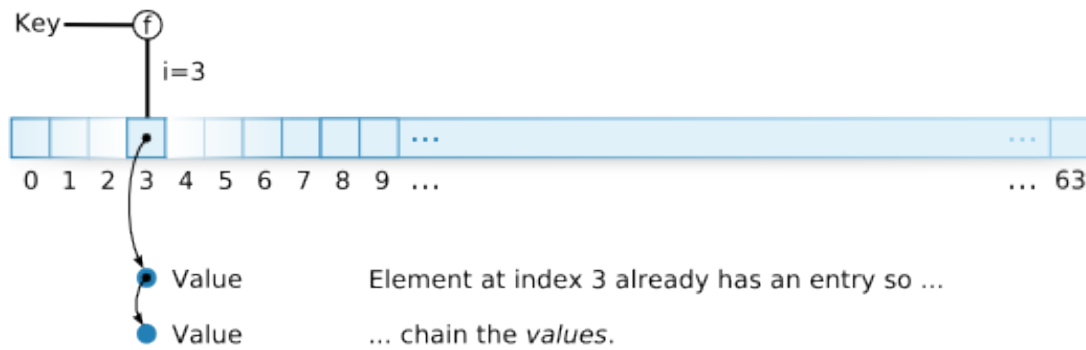


Figure 4: Chaining

However, if it can be determined a priori what set of keys will be used, it is possible to imagine a hash function that will never create collisions. This type of function is called a “ perfect hash function ” . Actually, there are tools which create a perfect hash function given a set of arbitrary strings. GNU gperf is one of those. ruby ’ s parser implementation uses GNU gperf but ... this is not the time to discuss it. We ’ ll discuss this in the second part of the book.

Data Structure

Let us start looking at the source code. As written in the introductory chapter, if there is data and code, it is better to read the data first. The following is the data type of `st_table`.

`st_table`

```
9 typedef struct st_table st_table;

16 struct st_table {
17     struct st_hash_type *type;
18     int num_bins;          /* slot count */
19     int num_entries;       /* total number of entries */
20     struct st_table_entry **bins; /* slot */
21 };
```

(st.h)

`struct st_table_entry`

```
16 struct st_table_entry {
17     unsigned int hash;
18     char *key;
19     char *record;
20     st_table_entry *next;
21 };
```

(st.c)

`st_table` is the main table structure. `st_table_entry` is a holder that stores one value. `st_table_entry` contains a member called `next` which of course is used to make `st_table_entry` into a linked list. This is the chain part of the chaining method. The `st_hash_type` data type is used, but I will explain this later. First let me explain the other parts so you can compare and understand the roles.

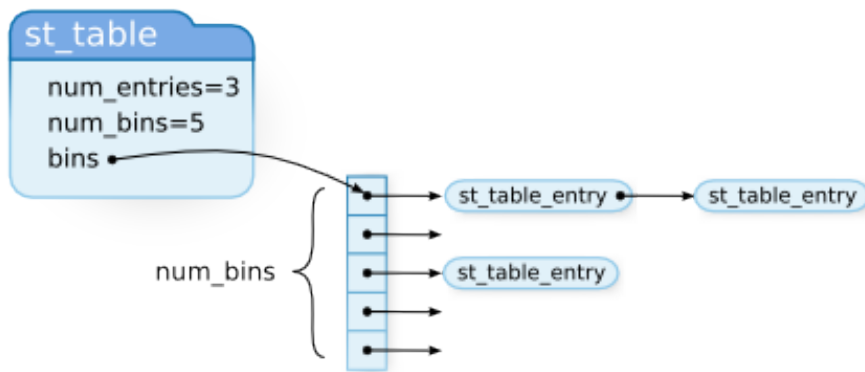


Figure 5: st_table data structure

So, let us comment on st_hash_type.

```
struct st_hash_type
```

```
11 struct st_hash_type {
12     int (*compare)(); /* comparison function */
13     int (*hash)();    /* hash function */
14 };
```

(st.h)

This is still Chapter 3 so let us examine it attentively.

```
int (*compare)()
```

This part shows, of course, the member compare which has a data type of “ a pointer to a function that returns an int ”. hash is also of the same type. This variable is substituted in the following way:

```
int
great_function(int n)
{
    /* ToDo: Do something great! */
    return n;
}

{
    int (*f)();
    f = great_function;
```

And it is called like this:

```
(*f)(7);
}
```

Here let us return to the st_hash_type commentary. Of the two members hash and compare, hash is the hash function f explained

previously.

On the other hand, `compare` is a function that evaluates if the key is actually the same or not. With the chaining method, in the spot with the same hash value `n`, multiple elements can be inserted. To know exactly which element is being searched for, this time it is necessary to use a comparison function that we can absolutely trust. `compare` will be that function.

This `st_hash_type` is a good generalized technique. The hash table itself cannot determine what the stored keys' data type will be. For example, in ruby, `st_table`'s keys are `ID` or `char*` or `VALUE`, but to write the same kind of hash for each (data type) is foolish. Usually, the things that change with the different key data types are things like the hash function. For things like memory allocation and collision detection, typically most of the code is the same. Only the parts where the implementation changes with a differing data type will be bundled up into a function, and a pointer to that function will be used. In this fashion, the majority of the code that makes up the hash table implementation can use it.

In object-oriented languages, in the first place, you can attach a procedure to an object and pass it (around), so this mechanism is not necessary. Perhaps it more correct to say that this mechanism is built-in as a language's feature.

st_hash_type example

The usage of a data structure like `st_hash_type` is good as an abstraction. On the other hand, what kind of code it actually passes through may be difficult to understand. If we do not examine what sort of function is used for hash or compare, we will not grasp the reality. To understand this, it is probably sufficient to look at `st_init_numtable()` introduced in the previous chapter. This function creates a table for integer data type keys.

```
st_init_numtable()
```

```
182 st_table*
183 st_init_numtable()
184 {
185     return st_init_table(&type_numhash);
186 }
```

```
(st.c)
```

`st_init_table()` is the function that allocates the table memory and so on. `type_numhash` is an `st_hash_type` (it is the member named “`type`” of `st_table`). Regarding this `type_numhash`:

```
type_numhash
```

```
37 static struct st_hash_type type_numhash = {
38     numcmp,
39     numhash,
40 };

552 static int
553 numcmp(x, y)
554     long x, y;
555 {
556     return x != y;
557 }
```

```

559 static int
560 numhash(n)
561     long n;
562 {
563     return n;
564 }

```

(st.c)

Very simple. The table that the ruby interpreter uses is by and large this type_numhash.

st_lookup()

Now then, let us look at the function that uses this data structure. First, it ' s a good idea to look at the function that does the searching. Shown below is the function that searches the hash table, st_lookup().

st_lookup()

```

247 int
248 st_lookup(table, key, value)
249     st_table *table;
250     register char *key;
251     char **value;
252 {
253     unsigned int hash_val, bin_pos;
254     register st_table_entry *ptr;
255
256     hash_val = do_hash(key, table);
257     FIND_ENTRY(table, ptr, hash_val, bin_pos);
258
259     if (ptr == 0) {
260         return 0;
261     }
262     else {
263         if (value != 0) *value = ptr->record;
264         return 1;
265     }
266 }

```

(st.c)

The important parts are pretty much in do_hash() and FIND_ENTRY(). Let us look at them in order.

do_hash()

```

68 #define do_hash(key,table) (unsigned int)(*(table)->type->hash)((key))

```

(st.c)

Just in case, let us write down the macro body that is difficult to understand:

(table)->type->hash

is a function pointer where the key is passed as a parameter. This is the syntax for calling the function. * is not applied to table. In other words, this macro is a hash value generator for a key, using the prepared hash function type->hash for each data type.

Next, let us examine FIND_ENTRY().

FIND_ENTRY()

```

235 #define FIND_ENTRY(table, ptr, hash_val, bin_pos) do {\
236     bin_pos = hash_val%(table)->num_bins;\
237     ptr = (table)->bins[bin_pos];\
238     if (PTR_NOT_EQUAL(table, ptr, hash_val, key)) {\
239         COLLISION;\
240         while (PTR_NOT_EQUAL(table, ptr->next, hash_val, key)) {\
241             ptr = ptr->next;\
242         }\
243         ptr = ptr->next;\
244     }\
245 } while (0)

227 #define PTR_NOT_EQUAL(table, ptr, hash_val, key) ((ptr) != 0 && \
    (ptr->hash != (hash_val) || !EQUAL((table), (key), (ptr)->key)))

66 #define EQUAL(table,x,y) \
    ((x)==(y) || (*table->type->compare)((x),(y)) == 0)

(st.c)

```

COLLISION is a debug macro so we will (should) ignore it.

The parameters of FIND_ENTRY(), starting from the left are:

1. st_table
2. the found entry will be pointed to by this parameter
3. hash value
4. temporary variable

And, the second parameter will point to the found st_table_entry*.

At the outermost level, a do .. while(0) is used to safely wrap up a multiple expression macro. This is ruby 's, or rather, C language 's preprocessor idiom. In the case of if(1), there may be a danger of adding an else part. In the case of while(1), it becomes necessary to add a break at the very end.

Also, there is no semicolon added after the while(0).

FIND_ENTRY();

This is so that the semicolon that is normally written at the end of an expression will not go to waste.

 st_add_direct()

Continuing on, let us examine `st_add_direct()` which is a function that adds a new relationship to the hash table. This function does not check if the key is already registered. It always adds a new entry. This is the meaning of direct in the function name.

`st_add_direct()`

```

308 void
309 st_add_direct(table, key, value)
310     st_table *table;
311     char *key;
312     char *value;
313 {
314     unsigned int hash_val, bin_pos;
315
316     hash_val = do_hash(key, table);
317     bin_pos = hash_val % table->num_bins;
318     ADD_DIRECT(table, key, value, hash_val, bin_pos);
319 }

```

(st.c)

Just as before, the `do_hash()` macro that obtains a value is called here. After that, the next calculation is the same as at the start of `FIND_ENTRY()`, which is to exchange the hash value for a real index.

Then the insertion operation seems to be implemented by `ADD_DIRECT()`. Since the name is all uppercase, we can anticipate that is a macro.

`ADD_DIRECT()`

```

268 #define ADD_DIRECT(table, key, value, hash_val, bin_pos) \
269 do { \
270     st_table_entry *entry; \
271     if (table->num_entries / (table->num_bins) \
272         > ST_DEFAULT_MAX_DENSITY) { \
273         rehash(table); \
274         bin_pos = hash_val % table->num_bins; \
275     } \
276     /* (A) */ \
277     entry = alloc(st_table_entry); \
278     entry->hash = hash_val; \
279     entry->key = key; \
280     entry->record = value; \
281     /* (B) */ \
282     entry->next = table->bins[bin_pos]; \
283     table->bins[bin_pos] = entry; \
284     table->num_entries++; \
285 } while (0)

```

(st.c)

The first if is an exception case so I will explain it afterwards.

(A) Allocate and initialize a `st_table_entry`.

(B) Insert the entry into the start of the list. This is the idiom for handling the list. In other words,

```
entry->next = list_beg;
list_beg = entry;
```

makes it possible to insert an entry to the front of the list. This is similar to “cons-ing” in the Lisp language. Check for yourself that even if `list_beg` is `NULL`, this code holds true.

Now, let me explain the code I left aside.

`ADD_DIRECT()`-rehash

```
271  if (table->num_entries / (table->num_bins)    \
      > ST_DEFAULT_MAX_DENSITY) {              \
272      rehash(table);                          \
273      bin_pos = hash_val % table->num_bins;    \
274  }
```

(st.c)

`DENSITY` is “concentration”. In other words, this conditional checks if the hash table is “crowded” or not. In the `st_table`, as the number of values that use the same `bin_pos` increases, the longer the link list becomes. In other words, search becomes slower. That is why for a given bin count, when the average elements per bin become too many, bin is increased and the crowding is reduced.

The current `ST_DEFAULT_MAX_DENSITY` is

`ST_DEFAULT_MAX_DENSITY`

```
23 #define ST_DEFAULT_MAX_DENSITY 5
```

(st.c)

Because of this setting, if in all `bin_pos` there are 5 `st_table_entries`, then the size will be increased.

st_insert()

`st_insert()` is nothing more than a combination of `st_add_direct()` and `st_lookup()`, so if you understand those two, this will be easy.

`st_insert()`

```
286 int
287 st_insert(table, key, value)
288     register st_table *table;
289     register char *key;
290     char *value;
291 {
```

```

292 unsigned int hash_val, bin_pos;
293 register st_table_entry *ptr;
294
295 hash_val = do_hash(key, table);
296 FIND_ENTRY(table, ptr, hash_val, bin_pos);
297
298 if (ptr == 0) {
299     ADD_DIRECT(table, key, value, hash_val, bin_pos);
300     return 0;
301 }
302 else {
303     ptr->record = value;
304     return 1;
305 }
306 }

```

(st.c)

It checks if the element is already registered in the table. Only when it is not registered will it be added. If there is a insertion, return 0. If there is no insertion, return a 1.

ID and Symbols

I ' ve already discussed what an ID is. It is a correspondence between an arbitrary string of characters and a value. It is used to declare various names. The actual data type is unsigned int.

From char* to ID

The conversion from string to ID is executed by `rb_intern()`. This function is rather long, so let ' s omit the middle.

`rb_intern()` (simplified)

```

5451 static st_table *sym_tbl; /* char* to ID */
5452 static st_table *sym_rev_tbl; /* ID to char* */

5469 ID
5470 rb_intern(name)
5471     const char *name;
5472 {
5473     const char *m = name;
5474     ID id;
5475     int last;
5476
5477     /* If for a name, there is a corresponding ID that is already
5478        registered, then return that ID */
5477     if (st_lookup(sym_tbl, name, &id))
5478         return id;

5479     /* omitted ... create a new ID */

5480     /* register the name and ID relation */
5538     id_regist:

```



```

5539  name = strdup(name);
5540  st_add_direct(sym_tbl, name, id);
5541  st_add_direct(sym_rev_tbl, id, name);
5542  return id;
5543 }

(parse.y)

```

The string and ID correspondence relationship can be accomplished by using the `st_table`. There probably isn't any especially difficult part here.

What is the omitted section doing? It is treating global variable names and instance variables names as special and flagging them. This is because in the parser, it is necessary to know the variable's classification from the ID. However, the fundamental part of ID is unrelated to this, so I won't explain it here.

From ID to char*

The reverse of `rb_intern()` is `rb_id2name()`, which takes an ID and generates a `char*`. You probably know this, but the 2 in `id2name` is “to”. “To” and “two” have the same pronunciation, so “2” is used for “to”. This syntax is often seen.

This function also sets the ID classification flags so it is long. Let me simplify it.

`rb_id2name()` (simplified)

```

char *
rb_id2name(id)
    ID id;
{
    char *name;

    if (st_lookup(sym_rev_tbl, id, &name))
        return name;
    return 0;
}

```

Maybe it seems that it is a little over-simplified, but in reality if we remove the details it really becomes this simple.

The point I want to emphasize is that the found name is not copied. The ruby API does not require (or rather, it forbids) the `free()`-ing of the return value. Also, when parameters are passed, it always copies them. In other words, the creation and release is completed by one side, either by the user or by ruby.

So then, when creation and release cannot be accomplished (when passed it is not returned) on a value, then a Ruby object is used. I have not yet discussed it, but a Ruby object is automatically released when it is no longer needed, even if we are not taking care of the object.

Converting VALUE and ID

ID is shown as an instance of the `Symbol` class at the Ruby level. And it can be obtained like so: `"string".intern`. The implementation of `String#intern` is `rb_str_intern()`.

rb_str_intern()

```

2996 static VALUE
2997 rb_str_intern(str)
2998     VALUE str;
2999 {
3000     ID id;
3001
3002     if (!RSTRING(str)->ptr || RSTRING(str)->len == 0) {
3003         rb_raise(rb_eArgError, "interning empty string");
3004     }
3005     if (strlen(RSTRING(str)->ptr) != RSTRING(str)->len)
3006         rb_raise(rb_eArgError, "string contains '\\0'");
3007     id = rb_intern(RSTRING(str)->ptr);
3008     return ID2SYM(id);
3009 }

```

(string.c)

This function is quite reasonable as a ruby class library code example. Please pay attention to the part where RSTRING() is used and casted, and where the data structure 's member is accessed.

Let 's read the code. First, rb_raise() is merely error handling so we ignore it for now. The rb_intern() we previously examined is here, and also ID2SYM is here. ID2SYM() is a macro that converts ID to Symbol.

And the reverse operation is accomplished using Symbol#to_s and such. The implementation is in sym_to_s.

sym_to_s()

```

522 static VALUE
523 sym_to_s(sym)
524     VALUE sym;
525 {
526     return rb_str_new2(rb_id2name(SYM2ID(sym)));
527 }

```

(object.c)

SYM2ID() is the macro that converts Symbol (VALUE) to an ID.

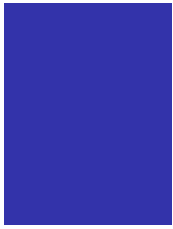
It looks like the function is not doing anything unreasonable. However, it is probably necessary to pay attention to the area around the memory handling. rb_id2name() returns a char* that must not be free(). rb_str_new2() copies the parameter 's char* and uses the copy (and does not change the parameter). In this way the policy is consistent, which allows the line to be written just by chaining the functions.

The original work is Copyright © 2002 - 2004 Minero AOKI.

Translated by Clifford Escobar CAOILE



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](https://creativecommons.org/licenses/by-nc-sa/2.5/).



Chapter 4: Classes and modules

In this chapter, we ' ll see the details of the data structures created for classes and modules.

Classes and methods definition

First, I ' d like to have a look at how Ruby classes are defined at the C level. This chapter investigates almost only particular cases, so I ' d like you to know first the way used most often.

The main API to define classes and modules consists of the following 6 functions:

```
rb_define_class()
rb_define_class_under()
rb_define_module()
rb_define_module_under()
rb_define_method()
rb_define_singleton_method()
```

There are a few other versions of these functions, but the extension libraries and even most of the core library is defined using just this API. I ' ll introduce to you these functions one by one.

Class definition

`rb_define_class()` defines a class at the top-level. Let ' s take the Ruby array class, `Array`, as an example.

Array class definition

```
19 VALUE rb_cArray;

1809 void
1810 Init_Array()
1811 {
1812   rb_cArray = rb_define_class("Array", rb_cObject);

(array.c)
```

`rb_cObject` and `rb_cArray` correspond respectively to `Object` and `Array` at the Ruby level. The added prefix `rb` shows that it belongs to ruby and the `c` that it is a class object. These naming rules are used everywhere in ruby.

This call to `rb_define_class()` allows to define `Array` that inherits from `Object`. At the same time as `rb_define_class()` creates the class object, it also defines the constant. That means that after this you can already access `Array` from a Ruby program. It corresponds to the

following Ruby program:

```
class Array < Object
```

I ' d like you to note the fact that there is no end. It was written like this on purpose. It is because with `rb_define_class()` the body of the class has not been executed.

Nested class definition

After that, there ' s `rb_define_class_under()`. This function defines a class nested in an other class or module. This time the example is what is returned by `stat(2)`, `File::Stat`.

Definition of `File::Stat`

```
78 VALUE rb_cFile;
80 static VALUE rb_cStat;

2581 rb_cFile = rb_define_class("File", rb_cIO);
2674 rb_cStat = rb_define_class_under(rb_cFile, "Stat", rb_cObject);

(file.c)
```

This code corresponds to the following Ruby program;

```
class File < IO
  class Stat < Object
```

This time again I omitted the end on purpose.

Module definition

`rb_define_module()` is simple so let ' s end this quickly.

Definition of `Enumerable`

```
17 VALUE rb_mEnumerable;

492 rb_mEnumerable = rb_define_module("Enumerable");

(enum.c)
```

The `m` in the beginning of `rb_mEnumerable` is similar to the `c` for classes: it shows that it is a module. The corresponding Ruby program is:

```
module Enumerable
```

`rb_define_module_under()` is not used much so we ' ll skip it.

Method definition

This time the function is the one for defining methods, `rb_define_method()`. It ' s used very often. We ' ll take once again an example from Array.

Definition of Array#to_s

```
1818 rb_define_method(rb_cArray, "to_s", rb_ary_to_s, 0);

(array.c)
```

With this the `to_s` method is defined in Array. The method body is given by a function pointer (`rb_ary_to_s`). The fourth parameter is the number of parameters taken by the method. As `to_s` does not take any parameters, it ' s 0. If we write the corresponding Ruby program, we ' ll have this:

```
class Array < Object
  def to_s
    # content of rb_ary_to_s()
  end
end
```

Of course the class part is not included in `rb_define_method()` and only the `def` part is accurate. But if there is no class part, it will look like the method is defined like a function, so I also wrote the enclosing class part.

One more example, this time taking a parameter:

Definition of Array#concat

```
1835 rb_define_method(rb_cArray, "concat", rb_ary_concat, 1);

(array.c)
```

The class for the definition is `rb_cArray` (Array), the method name is `concat`, its body is `rb_ary_concat()` and the number of parameters is 1. It corresponds to writing the corresponding Ruby program:

```
class Array < Object
  def concat( str )
    # content of rb_ary_concat()
  end
end
```

Singleton methods definition

We can define methods that are specific to an instance of an object. They are called singleton methods. As I used `File.unlink` as an example in chapter 1 “ Ruby language minimum ” , I first wanted to show it here, but for a particular reason we ' ll look at `File.link`

instead.

Definition of File.link

```
2624 rb_define_singleton_method(rb_cFile, "link", rb_file_s_link, 2);

(file.c)
```

It ' s used like `rb_define_method()`. The only difference is that here the first parameter is just the *object* where the method is defined. In this case, it ' s defined in `rb_cFile`.

Entry point

Being able to make definitions like before is great, but where are these functions called from, and by what means are they executed? These definitions are grouped in functions named `Init_xxxx()`. For instance, for `Array` a function `Init_Array()` like this has been made:

Init_Array

```
1809 void
1810 Init_Array()
1811 {
1812     rb_cArray = rb_define_class("Array", rb_cObject);
1813     rb_include_module(rb_cArray, rb_mEnumerable);
1814
1815     rb_define_singleton_method(rb_cArray, "allocate",
                                rb_ary_s_alloc, 0);
1816     rb_define_singleton_method(rb_cArray, "[]", rb_ary_s_create, -1);
1817     rb_define_method(rb_cArray, "initialize", rb_ary_initialize, -1);
1818     rb_define_method(rb_cArray, "to_s", rb_ary_to_s, 0);
1819     rb_define_method(rb_cArray, "inspect", rb_ary_inspect, 0);
1820     rb_define_method(rb_cArray, "to_a", rb_ary_to_a, 0);
1821     rb_define_method(rb_cArray, "to_ary", rb_ary_to_a, 0);
1822     rb_define_method(rb_cArray, "frozen?", rb_ary_frozen_p, 0);

(array.c)
```

The `Init` for the built-in functions are explicitly called during the startup of ruby. This is done in `inits.c`.

rb_call_inits()

```
47 void
48 rb_call_inits()
49 {
50     Init_sym();
51     Init_var_tables();
52     Init_Object();
53     Init_Comparable();
54     Init_Enumerable();
55     Init_Precision();
56     Init_eval();
57     Init_String();
58     Init_Exception();
```

```

59  Init_Thread();
60  Init_Numeric();
61  Init_Bignum();
62  Init_Array();

```

(inits.c)

This way, `Init_Array()` is called properly.

That explains it for built-in libraries, but what about extension libraries? In fact, for extension libraries the convention is the same. Take the following code:

```
require "myextension"
```

With this, if the loaded extension library is `myextension.so`, at load time, the (extern) function named `Init_myextension()` is called. How they are called is beyond the scope of this chapter. For that, you should read the chapter 18 “Load”. Here we ’ ll just end this with an example of `Init`.

The following example is from `stringio`, an extension library provided with ruby, that is to say not from a built-in library.

`Init_stringio()` (beginning)

```

895 void
896 Init_stringio()
897 {
898     VALUE StringIO = rb_define_class("StringIO", rb_cData);
899     rb_define_singleton_method(StringIO, "allocate",
          strio_s_allocate, 0);
900     rb_define_singleton_method(StringIO, "open", strio_s_open, -1);
901     rb_define_method(StringIO, "initialize", strio_initialize, -1);
902     rb_enable_super(StringIO, "initialize");
903     rb_define_method(StringIO, "become", strio_become, 1);
904     rb_define_method(StringIO, "reopen", strio_reopen, -1);

```

(ext/stringio/stringio.c)

Singleton classes

`rb_define_singleton_method()`

You should now be able to more or less understand how normal methods are defined. Somehow making the body of the method, then registering it in `m_tbl` will do. But what about singleton methods? We ’ ll now look into the way singleton methods are defined.

`rb_define_singleton_method()`

```

721 void
722 rb_define_singleton_method(obj, name, func, argc)

```

```

723  VALUE obj;
724  const char *name;
725  VALUE (*func)();
726  int argc;
727  {
728    rb_define_method(rb_singleton_class(obj), name, func, argc);
729  }

```

(class.c)

As I explained, `rb_define_method()` is a function used to define normal methods, so the difference from normal methods is only `rb_singleton_class()`. But what on earth are singleton classes?

In brief, singleton classes are virtual classes that are only used to execute singleton methods. Singleton methods are functions defined in singleton classes. Classes themselves are in the first place (in a way) the “ implementation ” to link objects and methods, but singleton classes are even more on the implementation side. In the Ruby language way, they are not formally included, and don ’ t appear much at the Ruby level.

rb_singleton_class()

Well, let ’ s confirm what the singleton classes are made of. It ’ s too simple to each time just show you the code of function so this time I ’ ll use a new weapon, a call graph.

```

rb_define_singleton_method
  rb_define_method
    rb_singleton_class
      SPECIAL_SINGLETON
        rb_make_metaclass
          rb_class_boot
            rb_singleton_class_attached

```

Call graphs are graphs showing calling relationships among functions (or more generally procedures). The call graphs showing all the calls written in the source code are called static call graphs. The ones expressing only the calls done during an execution are called dynamic call graphs.

This diagram is a static call graph and the indentation expresses which function calls which one. For instance, `rb_define_singleton_method()` calls `rb_define_method()` and `rb_singleton_class()`. And this `rb_singleton_class()` itself calls `SPECIAL_SINGLETON()` and `rb_make_metaclass()`.

Let ’ s go back to the code. When looking at the call graph, you can see that the calls made by `rb_singleton_class()` go very deep. Until now all call levels were shallow, so we could simply look at the functions without getting too lost. But at this depth, I easily forget what is going on in the code. That ’ s why in those situations I check the call graph to have a better understanding. This time, we ’ ll decode in parallel what the procedures below `rb_singleton_class()` do. The two points to look out for are the following ones:

What exactly are singleton classes?

What is the purpose of singleton classes?

Normal classes and singleton classes

Singleton classes are special classes: they 're basically the same as normal classes, but there are a few differences. We can say that finding these differences is explaining concretely singleton classes.

What should we do to find them? We should find the differences between the function creating normal classes and the one creating singleton classes. For this, we have to find the function for creating normal classes. That is as normal classes can be defined by `rb_define_class()`, it must call in a way or another a function to create normal classes. For the moment, we 'll not look at the content of `rb_define_class()` itself. I have some reasons to be interested in something that 's deeper. That 's why we will first look at the call graph of `rb_define_class()`.

```
rb_define_class
  rb_class_inherited
  rb_define_class_id
  rb_class_new
  rb_class_boot
  rb_make_metaclass
  rb_class_boot
  rb_singleton_class_attached
```

I 'm interested by `rb_class_new()`. Doesn 't this name means it creates a new class? Let 's confirm that.

```
rb_class_new()
```

```
37 VALUE
38 rb_class_new(super)
39   VALUE super;
40 {
41   Check_Type(super, T_CLASS);
42   if (super == rb_cClass) {
43     rb_raise(rb_eTypeError, "can't make subclass of Class");
44   }
45   if (FL_TEST(super, FL_SINGLETON)) {
46     rb_raise(rb_eTypeError, "can't make subclass of virtual class");
47   }
48   return rb_class_boot(super);
49 }
```

```
(class.c)
```

`Check_Type()` is checks the type of object structure, so we can ignore it. `rb_raise()` is error handling so we can ignore it. Only `rb_class_boot()` remains. So let 's look at it.

```
rb_class_boot()
```

```
21 VALUE
22 rb_class_boot(super)
23   VALUE super;
24 {
25   NEWOBJ(klass, struct RClass); /* allocates struct RClass */
26   OBJSETUP(klass, rb_cClass, T_CLASS); /* initialization of the RBasic part */
27
28   klass->super = super; /* (A) */
29   klass->iv_tbl = 0;
```

```

30  klass->m_tbl = 0;
31  klass->m_tbl = st_init_numtable();
32
33  OBJ_INFECT(klass, super);
34  return (VALUE)klass;
35 }

```

(class.c)

NEWOBJ() and OBJSETUP() are fixed expressions used when creating Ruby objects that possess one of the internal structure types (struct Rxxxx). They are both macros. In NEWOBJ(), struct RClass is created and the pointer is put in its first parameter klass. In OBJSETUP(), the struct RBasic member of the RClass (and thus basic.class and basic.flags) is initialized.

OBJ_INFECT() is a macro related to security. From now on, we ' ll ignore it.

At (A), the super member of klass is set to the super parameter. It looks like rb_class_boot() is a function that creates a class inheriting from super.

So, as rb_class_boot() is a function that creates a class, what does rb_class_new() is very similar.

Then, let ' s once more look at rb_singleton_class() ' s call graph:

```

rb_singleton_class
  SPECIAL_SINGLETON
    rb_make_metaclass
      rb_class_boot
      rb_singleton_class_attached

```

Here also rb_class_boot() is called. So up to that point, it ' s the same as in normal classes. What ' s going on after is what ' s different between normal classes and singleton classes, in other words the characteristics of singleton classes. If you everything ' s clear so far, we just need to read rb_singleton_class() and rb_make_metaclass().

Compressed rb_singleton_class()

rb_singleton_class() is a little long so we ' ll first remove its non-essential parts.

```
rb_singleton_class()
```

```

678 #define SPECIAL_SINGLETON(x,c) do {\
679     if (obj == (x)) {\
680         return c;\
681     }\
682 } while (0)

684 VALUE
685 rb_singleton_class(obj)
686     VALUE obj;
687 {
688     VALUE klass;
689
690     if (FIXNUM_P(obj) || SYMBOL_P(obj)) {

```

```

691     rb_raise(rb_eTypeError, "can't define singleton");
692 }
693 if (rb_special_const_p(obj)) {
694     SPECIAL_SINGLETON(Qnil, rb_cNilClass);
695     SPECIAL_SINGLETON(Qfalse, rb_cFalseClass);
696     SPECIAL_SINGLETON(Qtrue, rb_cTrueClass);
697     rb_bug("unknown immediate %ld", obj);
698 }
699
700 DEFER_INTS;
701 if (FL_TEST(RBASIC(obj)->klass, FL_SINGLETON) &&
702     (BUILTIN_TYPE(obj) == T_CLASS ||
703     rb_iv_get(RBASIC(obj)->klass, "__attached__") == obj)) {
704     klass = RBASIC(obj)->klass;
705 }
706 else {
707     klass = rb_make_metaclass(obj, RBASIC(obj)->klass);
708 }
709 if (OBJ_TAINTED(obj)) {
710     OBJ_TAINT(klass);
711 }
712 else {
713     FL_UNSET(klass, FL_TAINT);
714 }
715 if (OBJ_FROZEN(obj)) OBJ_FREEZE(klass);
716 ALLOW_INTS;
717
718 return klass;
719 }

```

(class.c)

The first and the second half are separated by a blank line. The first half handles a special case and the second half handles the general case. In other words, the second half is the trunk of the function. That 's why we ' ll keep it for later and talk about the first half.

Everything that is handled in the first half are non-pointer VALUES, in other words objects without an existing C structure. First, Fixnum and Symbol are explicitly picked. Then, `rb_special_const_p()` is a function that returns true for non-pointer VALUES, so there only Qtrue, Qfalse and Qnil should get caught. Other than that, there are no valid non-pointer value so a bug is reported with `rb_bug()`.

`DEFER_INTS()` and `ALLOW_INTS()` both end with the same INTS so you should see a pair in them. That 's the case, and they are macros related to signals. Because they are defined in `rubysig.h`, you can guess that INTS is the abbreviation of interrupts. You can ignore them.

Compressed `rb_make_metaclass()`

`rb_make_metaclass()`

```

142 VALUE
143 rb_make_metaclass(obj, super)
144     VALUE obj, super;
145 {
146     VALUE klass = rb_class_boot(super);
147     FL_SET(klass, FL_SINGLETON);
148     RBASIC(obj)->klass = klass;

```

```

149  rb_singleton_class_attached(klass, obj);
150  if (BUILTIN_TYPE(obj) == T_CLASS) {
151      RBASIC(klass)->klass = klass;
152      if (FL_TEST(obj, FL_SINGLETON)) {
153          RCLASS(klass)->super =
              RBASIC(rb_class_real(RCLASS(obj)->super))->klass;
154      }
155  }
156
157  return klass;
158 }

(class.c)

```

We already saw `rb_class_boot()`. It creates a (normal) class using the `super` parameter as its superclass. After that, the `FL_SINGLETON` of this class is set. This is clearly suspicious. The name of the function makes us think that it is not the indication of a singleton class.

What are singleton classes?

Continuing the simplification process, furthermore as parameters, return values, local variables are all `VALUE`, we can throw away the declarations. That makes us able to compress to the following:

`rb_singleton_class()` `rb_make_metaclass()` (after compression)

```

rb_singleton_class(obj)
{
    if (FL_TEST(RBASIC(obj)->klass, FL_SINGLETON) &&
        (BUILTIN_TYPE(obj) == T_CLASS || BUILTIN_TYPE(obj) == T_MODULE) &&
        rb_iv_get(RBASIC(obj)->klass, "__attached__") == obj) {
        klass = RBASIC(obj)->klass;
    }
    else {
        klass = rb_make_metaclass(obj, RBASIC(obj)->klass);
    }
    return klass;
}

rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj);
    if (BUILTIN_TYPE(obj) == T_CLASS) {
        RBASIC(klass)->klass = klass;
        if (FL_TEST(obj, FL_SINGLETON)) {
            RCLASS(klass)->super =
                RBASIC(rb_class_real(RCLASS(obj)->super))->klass;
        }
    }

    return klass;
}

```

The condition of the if statement of `rb_singleton_class()` seems quite complicated. However, this condition is not connected to the

mainstream of `rb_make_metaclass()` so we 'll see it later. Let 's first think about what happens on the false branch of the if.

The `BUILTIN_TYPE()` of `rb_make_metaclass()` is similar to `TYPE()` as it is a macro to get the structure type flag (`T_XXXX`). That means this check in `rb_make_metaclass` means “ if obj is a class ”. For the moment it 's better not to limit ourselves to obj being a class, so we 'll remove it.

With these simplifications, we get the the following:

`rb_singleton_class()` `rb_make_metaclass()` (after recompression)

```
rb_singleton_class(obj)
{
  klass = create a class with RBASIC(obj)->klass as superclass;
  FL_SET(klass, FL_SINGLETON);
  RBASIC(obj)->klass = klass;
  return klass;
}
```

But there is still a quite hard to understand side to it. That 's because `klass` is used too often. So let 's rename the `klass` variable to `sclass`.

`rb_singleton_class()` `rb_make_metaclass()` (variable substitution)

```
rb_singleton_class(obj)
{
  sclass = create a class with RBASIC(obj)->klass as superclass;
  FL_SET(sclass, FL_SINGLETON);
  RBASIC(obj)->klass = sclass;
  return sclass;
}
```

Now it should be very easy to understand. To make it even simpler, I 've represented what is done with a diagram (figure 1). In the horizontal direction is the “ instance – class ” relation, and in the vertical direction is inheritance (the superclasses are above).

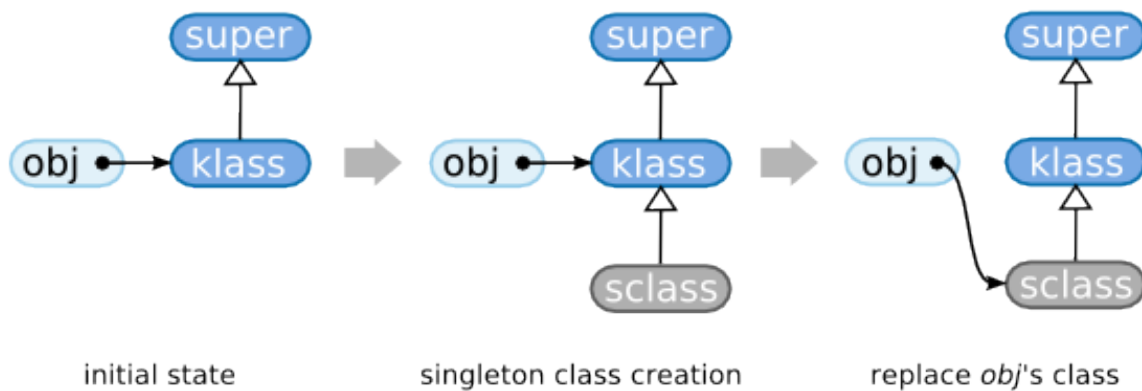


Figure 1: `rb_singleton_class`

When comparing the first and last part of this diagram, you can understand that `sclass` is inserted without changing the structure. That 's all there is to singleton classes. In other words the inheritance is increased one step. If a method is defined in a singleton class, this construction allows the other instances of `klass` to define completely different methods.

Singleton classes and instances

By the way, you must have seen that during the compression process, the call to `rb_singleton_class_attached()` was stealthily removed. Here:

```
rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj); /* THIS */
}
```

Let 's have a look at what it does.

```
rb_singleton_class_attached()
```

```
130 void
131 rb_singleton_class_attached(klass, obj)
132     VALUE klass, obj;
133 {
134     if (FL_TEST(klass, FL_SINGLETON)) {
135         if (!RCLASS(klass)->iv_tbl) {
136             RCLASS(klass)->iv_tbl = st_init_numtable();
137         }
138         st_insert(RCLASS(klass)->iv_tbl,
139                 rb_intern("__attached__"), obj);
139     }
140 }
```

(class.c)

If the `FL_SINGLETON` flag of `klass` is set... in other words if it 's a singleton class, put the `__attached__` `obj` relation in the instance variable table of `klass` (`iv_tbl`). That 's how it looks like (in our case `klass` is always a singleton class... in other words its `FL_SINGLETON` flag is always set).

`__attached__` does not have the `@` prefix, but it 's stored in the instance variables table so it 's still an instance variable. Such an instance variable can never be read at the Ruby level so it can be used to keep values for the system 's exclusive use.

Let 's now think about the relationship between `klass` and `obj`. `klass` is the singleton class of `obj`. In other words, this “invisible” instance variable allows the singleton class to remember the instance it was created from. Its value is used when the singleton class is changed, notably to call hook methods on the instance (i.e. `obj`). For example, when a method is added to a singleton class, the `obj` 's `singleton_method_added` method is called. There is no logical necessity to doing it, it was done because that 's how it was defined in the language.

But is it really all right? Storing the instance in `__attached__` will force one singleton class to have only one attached instance. For example, by getting (in some way or an other) the singleton class and calling `new` on it, won 't a singleton class end up having multiple instances?

This cannot be done because the proper checks are done to prevent the creation of an instance of a singleton class.

Singleton classes are in the first place for singleton methods. Singleton methods are methods existing only on a particular object. If singleton classes could have multiple instances, there would be the same as normal classes. That's why they are forced to only have one instance.

Summary

We've done a lot, maybe made a real mayhem, so let's finish and put everything in order with a summary.

What are singleton classes? They are classes that have the `FL_SINGLETON` flag set and that can only have one instance.

What are singleton methods? They are methods defined in the singleton class of an object.

Metaclasses

Inheritance of singleton methods

Infinite chain of classes

Even a class has a class, and it's `Class`. And the class of `Class` is again `Class`. We find ourselves in an infinite loop (figure 2).

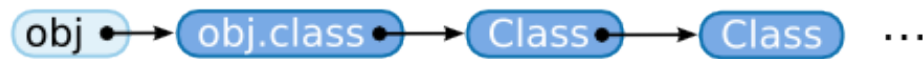


Figure 2: Infinite loop of classes

Up to here it's something we've already gone through. What's going after that is the theme of this chapter. Why do classes have to make a loop?

First, in Ruby all data are objects. And classes are data so in Ruby they have to be objects.

As they are objects, they must answer to methods. And setting the rule “to answer to methods you must belong to a class” made processing easier. That's where comes the need for a class to also have a class.

Let's base ourselves on this and think about the way to implement it. First, we can try first with the most naïve way, `Class`'s class is `ClassClass`, `ClassClass`'s class is `ClassClassClass...`, chaining classes of classes one by one. But whichever the way you look at it, this can't be implemented effectively. That's why it's common in object oriented languages where classes are objects that `Class`'s class is to `Class` itself, creating an endless virtual instance-class relationship.

I'm repeating myself, but the fact that `Class`'s class is `Class` is only to make the implementation easier, there's nothing important in this logic.

“Class is also an object”

“ Everything is an object ” is often used as advertising statement when speaking about Ruby. And as a part of that, “ Classes are also object! ” also appears. But these expressions often go too far. When thinking about these sayings, we have to split them in two:

all data are objects
classes are data

Talking about data or code makes a discussion much harder to understand. That ’ s why here we ’ ll restrict the meaning of “ data ” to “ what can be put in variables in programs ” .

Being able to manipulate classes from programs gives programs the ability to manipulate themselves. This is called reflection. It fits object oriented languages, and even more Ruby with the classes it has, to be able to directly manipulate classes.

Nevertheless, classes could be made available in a form that is not an object. For example, classes could be manipulated with function-style methods (functions defined at the top-level). However, as inside the interpreter there are data structures to represent the classes, it ’ s more natural in object oriented languages to make them available directly. And Ruby did this choice.

Furthermore, an objective in Ruby is for all data to be objects. That ’ s why it ’ s appropriate to make them objects.

By the way, there is a reason not linked to reflection why in Ruby classes had to be made objects. That is to be able to define methods independently from instances (what is called static methods in Java in C++).

And to implement static methods, another thing was necessary: singleton methods. By chain reaction, that also makes singleton classes necessary. Figure 3 shows these dependency relationships.

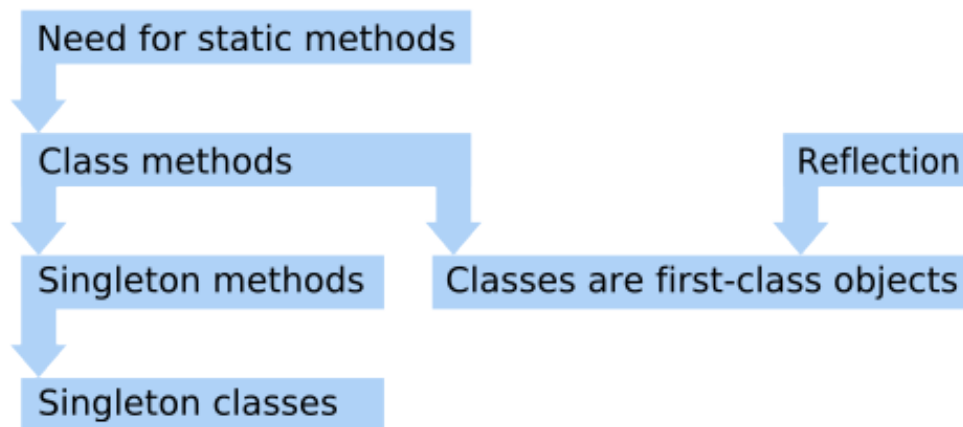


Figure 3: Requirements dependencies

Class methods inheritance

In Ruby, singleton methods defined in a class are called class methods. However, their specification is a little strange. Why are class methods inherited?

```

class A
  def A.test # defines a singleton method in A
    puts("ok")
  end
end
  
```



```
end

class B < A
end

B.test() # calls it
```

This can't occur with singleton methods from objects that are not classes. In other words, classes are the only ones handled specially. In the following section we'll see how class methods are inherited.

Singleton class of a class

Assuming that class methods are inherited, where is this operation done? At class definition (creation)? At singleton method definition? Then let's look at the code defining classes.

Class definition means of course `rb_define_class()`. Now let's take the call graph of this function.

```
rb_define_class
  rb_class_inherited
  rb_define_class_id
    rb_class_new
      rb_class_boot
    rb_make_metaclass
      rb_class_boot
      rb_singleton_class_attached
```

If you're wondering where you've seen it before, we looked at it in the previous section. At that time you did not see it but if you look closely, why does `rb_make_metaclass()` appear? As we saw before, this function introduces a singleton class. This is very suspicious. Why is this called even if we are not defining a singleton function? Furthermore, why is the lower level `rb_make_metaclass()` used instead of `rb_singleton_class()`? It looks like we have to check these surroundings again.

rb_define_class_id()

Let's first start our reading with its caller, `rb_define_class_id()`.

```
rb_define_class_id()
```

```
160 VALUE
161 rb_define_class_id(id, super)
162   ID id;
163   VALUE super;
164 {
165   VALUE klass;
166
167   if (!super) super = rb_cObject;
168   klass = rb_class_new(super);
169   rb_name_class(klass, id);
170   rb_make_metaclass(klass, RBASIC(super)->klass);
171
172   return klass;
173 }
```

```
(class.c)
```

`rb_class_new()` was a function that creates a class with `super` as its superclass. `rb_name_class()` 's name means it names a class, but for the moment we do not care about names so we 'll skip it. After that there 's the `rb_make_metaclass()` in question. I 'm concerned by the fact that when called from `rb_singleton_class()`, the parameters were different. Last time was like this:

```
rb_make_metaclass(obj, RBASIC(obj)->klass);
```

But this time is like this:

```
rb_make_metaclass(klass, RBASIC(super)->klass);
```

So as you can see it 's slightly different. How do the results change depending on that? Let 's have once again a look at a simplified `rb_make_metaclass()`.

rb_make_metaclass (once more)

`rb_make_metaclass` (after first compression)

```
rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj);
    if (BUILTIN_TYPE(obj) == T_CLASS) {
        RBASIC(klass)->klass = klass;
        if (FL_TEST(obj, FL_SINGLETON)) {
            RCLASS(klass)->super =
                RBASIC(rb_class_real(RCLASS(obj)->super))->klass;
        }
    }

    return klass;
}
```

Last time, the `if` statement was skillfully skipped, but looking once again, something is done only for `T_CLASS`, in other words classes. This clearly looks important. In `rb_define_class_id()`, as it 's called like this:

```
rb_make_metaclass(klass, RBASIC(super)->klass);
```

Let 's expand `rb_make_metaclass()` 's parameter variables with this values.

`rb_make_metaclass` (recompression)

```
rb_make_metaclass(klass, super_class /* == RBASIC(super)->klass */)
{
    sclass = create a class with super_class as superclass;
```

```

RBASIC(klass)->klass = sclass;
RBASIC(sclass)->klass = sclass;
return sclass;
}

```

Doing this as a diagram gives something like figure 4. In it, the names between parentheses are singleton classes. This notation is often used in this book so I'd like you to remember it. This means that `obj`'s singleton class is written as `(obj)`. And `(klass)` is the singleton class for `klass`. It looks like the singleton class is caught between a class and this class's superclass's class.



Figure 4: Introduction of a class's singleton class

From this result, and moreover when thinking more deeply, we can think that the superclass's class must again be the superclass's singleton class. You'll understand with one more inheritance level (figure 5).

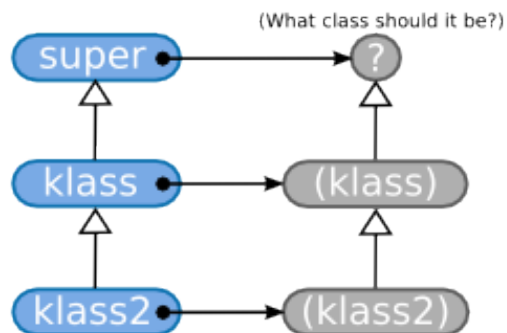


Figure 5: Hierarchy of multi-level inheritance

As the relationship between `super` and `klass` is the same as the one between `klass` and `klass2`, `c` must be the singleton class (`super`). If you continue like this, finally you'll arrive at the conclusion that `Object`'s class must be `(Object)`. And that's the case in practice. For example, by inheriting like in the following program :

```

class A < Object
end
class B < A
end

```

internally, a structure like figure 6 is created.

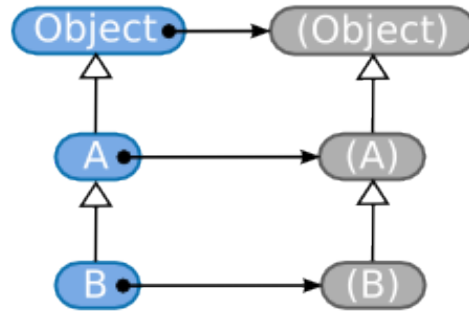


Figure 6: Class hierarchy and metaclasses

As classes and their metaclasses are linked and inherit like this, class methods are inherited.

Class of a class of a class

You've understood the working of class methods inheritance, but by doing that, in the opposite some questions have appeared. What is the class of a class's singleton class? To do this we can try debugging. I've made the figure 7 from the results of this investigation.

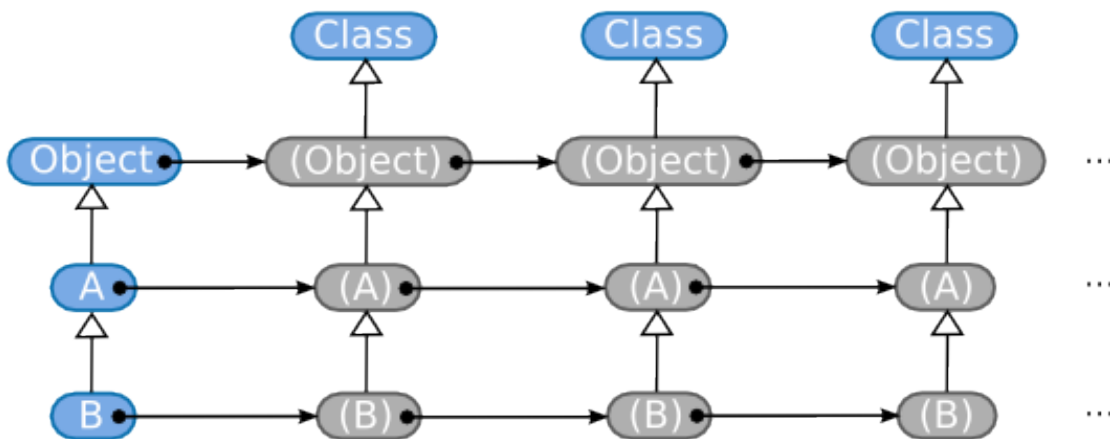


Figure 7: Class of a class's singleton class

A class's singleton class puts itself as its own class. Quite complicated.

The second question: the class of Object must be Class. Didn't I properly confirm this in chapter 1: Ruby language minimum?

```
p(Object.class()) # Class
```

Certainly, that's the case "at the Ruby level". But "at the C level", it's the singleton class (Object). If (Object) does not appear at the Ruby level, it's because Object#class skips the singleton classes. Let's look at the body of the method, rb_obj_class() to confirm that.

```
rb_obj_class()
```

```
86 VALUE
87 rb_obj_class(obj)
```

```

88  VALUE obj;
89  {
90    return rb_class_real(CLASS_OF(obj));
91  }

76  VALUE
77  rb_class_real(cl)
78    VALUE cl;
79  {
80    while (FL_TEST(cl, FL_SINGLETON) || TYPE(cl) == T_ICLASS) {
81      cl = RCLASS(cl)->super;
82    }
83    return cl;
84  }

(object.c)

```

`CLASS_OF(obj)` returns the basic class of `obj`. While in `rb_class_real()`, all singleton classes are skipped (advancing towards the superclass). In the first place, singleton class are caught between a class and its superclass, like a proxy. That 's why when a “ real ” class is necessary, we have to follow the superclass chain (figure 8).

`I_CLASS` will appear later when we will talk about include.

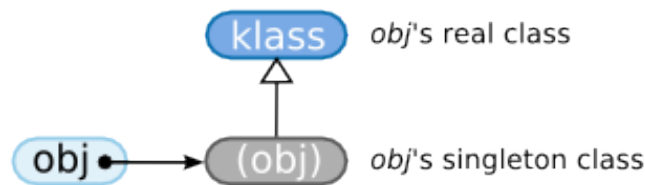


Figure 8: Singleton class and real class

Singleton class and metaclass

Well, the singleton classes that were introduced in classes is also one type of class, it 's a class 's class. So it can be called metaclass.

However, you should be wary of the fact that singleton class are not metaclasses. It 's the singleton classes introduced in classes that are metaclasses. The important fact is not that they are singleton classes, but that they are the classes of classes. I was stuck on this point when I started learning Ruby. As I may not be the only one, I would like to make this clear.

Thinking about this, the `rb_make_metaclass()` function name is not very good. When used in classes, it does indeed create a metaclass, but not in the other cases, when using objects.

Then finally, even if you understood that some class are metaclasses, it 's not as if there was any concrete gain. I 'd like you not to care too much about it.

Bootstrap

We have nearly finished our talk about classes and metaclasses. But there is still one problem left. It 's about the 3 metaobjects Object, Module and Class. These 3 cannot be created with the common use API. To make a class, its metaclass must be built, but like we saw some time ago, the metaclass 's superclass is Class. However, as Class has not been created yet, the metaclass cannot be build. So in

ruby, only these 3 classes 's creation is handled specially.

Then let 's look at the code:

Object, Module and Class creation

```
1243 rb_cObject = boot_defclass("Object", 0);
1244 rb_cModule = boot_defclass("Module", rb_cObject);
1245 rb_cClass = boot_defclass("Class", rb_cModule);
1246
1247 metaclass = rb_make_metaclass(rb_cObject, rb_cClass);
1248 metaclass = rb_make_metaclass(rb_cModule, metaclass);
1249 metaclass = rb_make_metaclass(rb_cClass, metaclass);
```

(object.c)

First, in the first half, `boot_defclass()` is similar to `rb_class_boot()`, it just creates a class with its given superclass set. These links give us something like the left part of figure 9.

And in the three lines of the second half, `(Object)`, `(Module)` and `(Class)` are created and set (right figure 9). `(Object)` and `(Module)` 's classes... that is themselves... is already set in `rb_make_metaclass()` so there is no problem. With this, the metaobjects ' bootstrap is finished.

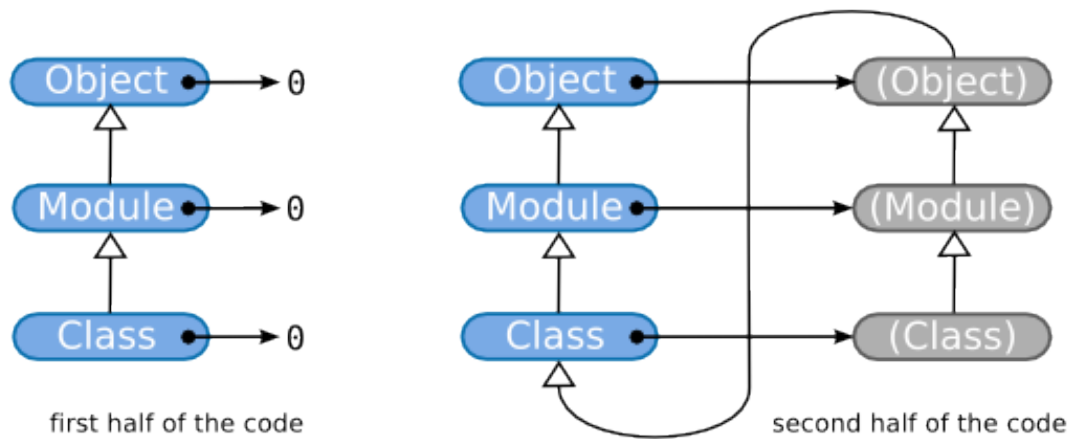


Figure 9: Metaobjects creation

After taking everything into account, it gives us a the final shape like figure 10.

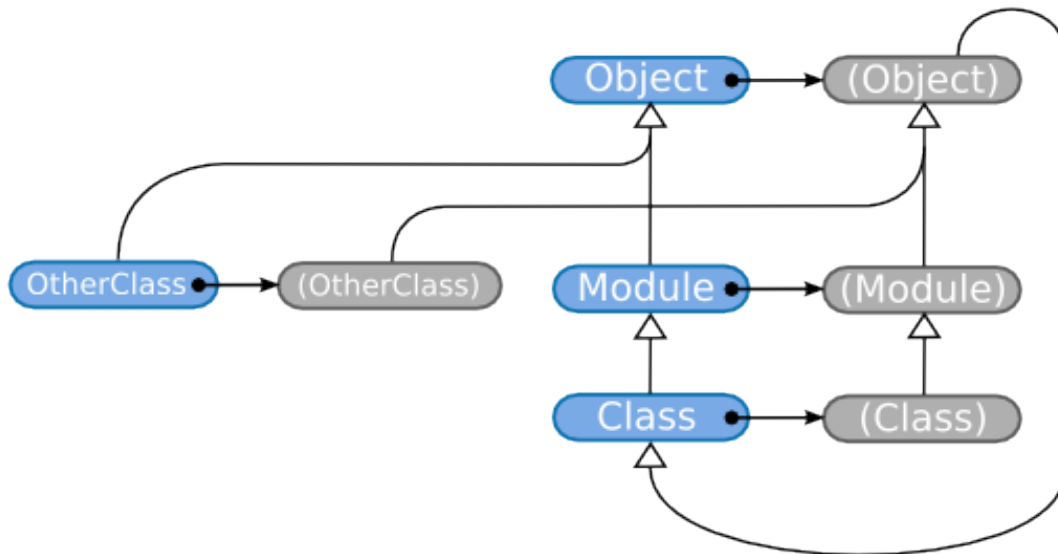


Figure 10: Ruby metaobjects

Class names

In this section, we will analyse how 's formed the reciprocal conversion between class and class names, in other words constants. Concretely, we will target `rb_define_class()` and `rb_define_class_under()`.

Name class

First we 'll read `rb_defined_class()`. After the end of this function, the class can be found from the constant.

`rb_define_class()`

```

183 VALUE
184 rb_define_class(name, super)
185     const char *name;
186     VALUE super;
187 {
188     VALUE klass;
189     ID id;
190
191     id = rb_intern(name);
192     if (rb_autoload_defined(id)) { /* (A) autoload */
193         rb_autoload_load(id);
194     }
195     if (rb_const_defined(rb_cObject, id)) { /* (B) rb_const_defined */
196         klass = rb_const_get(rb_cObject, id); /* (C) rb_const_get */
197         if (TYPE(klass) != T_CLASS) {
198             rb_raise(rb_eTypeError, "%s is not a class", name);
199         } /* (D) rb_class_real */
200         if (rb_class_real(RCLASS(klass)->super) != super) {
201             rb_name_error(id, "%s is already defined", name);
202         }

```

```

203     return klass;
204 }
205 if (!super) {
206     rb_warn("no super class for '%s', Object assumed", name);
207 }
208 klass = rb_define_class_id(id, super);
209 rb_class_inherited(super, klass);
210 st_add_direct(rb_class_tbl, id, klass);
211
212     return klass;
213 }

(class.c)

```

Many things can be understood with what 's before and after `rb_define_class_id()`... Before we acquire or create the class. After we set the constant. We will look at it in more detail below.

(A) In Ruby, there is an autoload function that automatically loads libraries when some constants are accessed. This is done in the `rb_autoload_xxxx()` function. You can ignore it without any problem.

(B) We determine whether the name constant has been defined or not in Object.

(C) Get the value of the name constant. This will be explained in detail in chapter 6.

(D) We 've seen `rb_class_real()` some time ago. If the class `c` is a singleton class or an ICLASS, it climbs the super hierarchy up to a class that is not and returns it. In short, this function skips the virtual classes that should not appear at the Ruby level.

That 's what we can read nearby.

As around constants are involved, it is very troublesome. However, we will talk about class definition in the constants chapter so for the moment we will content ourselves with a partial description.

After `rb_define_class_id`, we can find the following:

```
st_add_direct(rb_class_tbl, id, klass);
```

This part assigns the class to the constant. However, whichever the way you look at it you do not see that. In fact, top-level classes are separated from the other constants and regrouped in `rb_class_tbl()`. The split is slightly related to the GC. It 's not essential.

Class name

We understood how the class can be obtained from the class name, but how to do the opposite? By doing things like calling `p` or `Class#name`, we can get the name of the class, but how is it implemented?

In fact this was already done a long time ago by `rb_name_class()`. The call is around the following:

```

rb_define_class
  rb_define_class_id
    rb_name_class

```


Let ' s look at its content:

```
rb_name_class()
```

```
269 void
270 rb_name_class(klass, id)
271     VALUE klass;
272     ID id;
273 {
274     rb_iv_set(klass, "__classid__", ID2SYM(id));
275 }
```

(variable.c)

`__classid__` is another instance variable that can ' t be seen from Ruby. As only VALUES can be put in the instance variable table, the ID is converted to Symbol using `ID2SYM()`.

That ' s how we are able to find the constant name from the class.

Nested classes

So, in the case of classes defined at the top-level, we know how works the reciprocal link between name and class. What ' s left is the case of classes defined in modules or other classes, and for that it ' s a little more complicated. The function to define these nested classes is `rb_define_class_under()`.

```
rb_define_class_under()
```

```
215 VALUE
216 rb_define_class_under(outer, name, super)
217     VALUE outer;
218     const char *name;
219     VALUE super;
220 {
221     VALUE klass;
222     ID id;
223
224     id = rb_intern(name);
225     if (rb_const_defined_at(outer, id)) {
226         klass = rb_const_get(outer, id);
227         if (TYPE(klass) != T_CLASS) {
228             rb_raise(rb_eTypeError, "%s is not a class", name);
229         }
230         if (rb_class_real(RCLASS(klass)->super) != super) {
231             rb_name_error(id, "%s is already defined", name);
232         }
233         return klass;
234     }
235     if (!super) {
236         rb_warn("no super class for '%s::%s', Object assumed",
237             rb_class2name(outer), name);
238     }
239     klass = rb_define_class_id(id, super);
```

```

240  rb_set_class_path(klass, outer, name);
241  rb_class_inherited(super, klass);
242  rb_const_set(outer, id, klass);
243
244  return klass;
245 }

```

(class.c)

The structure is like the one of `rb_define_class()`: before the call to `rb_define_class_id()` is the redefinition check, after is the creation of the reciprocal link between constant and class. The first half is pretty boringly similar to `rb_define_class()` so we ' ll skip it. In the second half, `rb_set_class()` is new. We ' re going to look at it.

rb_set_class_path()

This function gives the name `name` to the class `klass` nested in the class `under`. “ class path ” means a name including all the nesting information starting from top-level, for example “ `Net::NetPrivate::Socket` ” .

`rb_set_class_path()`

```

210 void
211 rb_set_class_path(klass, under, name)
212     VALUE klass, under;
213     const char *name;
214 {
215     VALUE str;
216
217     if (under == rb_cObject) {
218         /* defined at top-level */
219         str = rb_str_new2(name); /* create a Ruby string from name */
220     }
221     else {
222         /* nested constant */
223         str = rb_str_dup(rb_class_path(under)); /* copy the return value */
224         rb_str_cat2(str, "::"); /* concatenate "::" */
225         rb_str_cat2(str, name); /* concatenate name */
226     }
227     rb_iv_set(klass, "__classpath__", str);
228 }

```

(variable.c)

Everything except the last line is the construction of the class path, and the last line makes the class remember its own name.

`__classpath__` is of course another instance variable that can ' t be seen from a Ruby program. In `rb_name_class()` there was `__classid__`, but `id` is different because it does not include nesting information (look at the table below).

<code>__classpath__</code>	<code>Net::NetPrivate::Socket</code>
<code>__classid__</code>	<code>Socket</code>

It means classes defined for example in `rb_defined_class()` all have `__classid__` or `__classpath__` defined. So to find `under` ' s classpath we can look up in these instance variables. This is done by `rb_class_path()`. We ' ll omit its content.

Nameless classes

Contrary to what I have just said, there are in fact cases in which neither `__classpath__` nor `__classid__` are set. That is because in Ruby you can use a method like the following to create a class.

```
c = Class.new()
```

If you create a class like this, we won't go through `rb_define_class_id()` and the `classpath` won't be set. In this case, `c` does not have any name, which is to say we get an unnamed class.

However, if later it's assigned into a constant, the name of this constant will be attached to the class.

```
SomeClass = c # the class name is SomeClass
```

Strictly speaking, the name is attached after the assignment, the first time it is requested. For instance, when calling `p` on this `SomeClass` class or when calling the `Class#name` method. When doing this, a value equal to the class is searched in `rb_class_tbl`, and a name has to be chosen. The following case can also happen:

```
class A
  class B
    C = tmp = Class.new()
    p(tmp) # here we search for the name
  end
end
```

so in the worst case we have to search for the whole constant space. However, generally, there aren't many constants so searching all constants does not take too much time.

Include

We only talked about classes so let's finish this chapter with something else and talk about module inclusion.

`rb_include_module` (1)

Includes are done by the ordinary method `Module#include`. Its corresponding function in C is `rb_include_module()`. In fact, to be precise, its body is `rb_mod_include()`, and there `Module#append_feature` is called, and this function's default implementation finally calls `rb_include_module()`. Mixing what's happening in Ruby and C gives us the following call graph.

```
Module#include (rb_mod_include)
  Module#append_features (rb_mod_append_features)
    rb_include_module
```

All usual includes are done by `rb_include_module()`. This function is a little long so we'll look at it a half at a time.

rb_include_module (first half)

```

/* include module in class */
347 void
348 rb_include_module(klass, module)
349     VALUE klass, module;
350 {
351     VALUE p, c;
352     int changed = 0;
353
354     rb_frozen_class_p(klass);
355     if (!OBJ_TAINTED(klass)) {
356         rb_secure(4);
357     }
358
359     if (NIL_P(module)) return;
360     if (klass == module) return;
361
362     switch (TYPE(module)) {
363     case T_MODULE:
364     case T_CLASS:
365     case T_ICLASS:
366         break;
367     default:
368         Check_Type(module, T_MODULE);
369     }

```

(class.c)

For the moment it 's only security and type checking, therefore we can ignore it. The process itself is below:

rb_include_module (second half)

```

371 OBJ_INFECT(klass, module);
372 c = klass;
373 while (module) {
374     int superclass_seen = Qfalse;
375
376     if (RCLASS(klass)->m_tbl == RCLASS(module)->m_tbl)
377         rb_raise(rb_eArgError, "cyclic include detected");
378     /* (A) skip if the superclass already includes module */
379     for (p = RCLASS(klass)->super; p; p = RCLASS(p)->super) {
380         switch (BUILTIN_TYPE(p)) {
381         case T_ICLASS:
382             if (RCLASS(p)->m_tbl == RCLASS(module)->m_tbl) {
383                 if (!superclass_seen) {
384                     c = p; /* move the insertion point */
385                 }
386                 goto skip;
387             }
388             break;
389         case T_CLASS:
390             superclass_seen = Qtrue;
391             break;
392         }
393     }
394     c = RCLASS(c)->super =

```

```

        include_class_new(module, RCLASS(c)->super);
395     changed = 1;
396     skip:
397     module = RCLASS(module)->super;
398 }
399 if (changed) rb_clear_cache();
400 }

(class.c)

```

First, what the (A) block does is written in the comment. It seems to be a special condition so let's first skip reading it for now. By extracting the important parts from the rest we get the following:

```

c = klass;
while (module) {
    c = RCLASS(c)->super = include_class_new(module, RCLASS(c)->super);
    module = RCLASS(module)->super;
}

```

In other words, it's a repetition of module's super. What is in module's super must be a module included by module (because our intuition tells us so). Then the superclass of the class where the inclusion occurs is replaced with something. We do not understand much what, but at the moment I saw that I felt "Ah, doesn't this look the addition of elements to a list (like LISP's cons)?" and it suddenly make the story faster. In other words it's the following form:

```
list = new(item, list)
```

Thinking about this, it seems we can expect that module is inserted between c and c->super. If it's like this, it fits module's specification.

But to be sure of this we have to look at include_class_new().

include_class_new()

```
include_class_new()
```

```

319 static VALUE
320 include_class_new(module, super)
321     VALUE module, super;
322 {
323     NEWOBJ(klass, struct RClass);      /* (A) */
324     OBJSETUP(klass, rb_cClass, T_ICLASS);
325
326     if (BUILTIN_TYPE(module) == T_ICLASS) {
327         module = RBASIC(module)->klass;
328     }
329     if (!RCLASS(module)->iv_tbl) {
330         RCLASS(module)->iv_tbl = st_init_numtable();
331     }
332     klass->iv_tbl = RCLASS(module)->iv_tbl; /* (B) */
333     klass->m_tbl = RCLASS(module)->m_tbl;
334     klass->super = super;                  /* (C) */
335     if (TYPE(module) == T_ICLASS) {      /* (D) */

```

```

336     RBASIC(klass)->klass = RBASIC(module)->klass; /* (D-1) */
337 }
338 else {
339     RBASIC(klass)->klass = module;          /* (D-2) */
340 }
341 OBJ_INFECT(klass, module);
342 OBJ_INFECT(klass, super);
343
344 return (VALUE)klass;
345 }

(class.c)

```

We're lucky there's nothing we do not know.

(A) First create a new class.

(B) Transplant module's instance variable and method tables into this class.

(C) Make the including class's superclass (super) the super class of this new class.

In other words, this function creates an include class for the module. The important point is that at (B) only the pointer is moved on, without duplicating the table. Later, if a method is added, the module's body and the include class will still have exactly the same methods (figure 11).

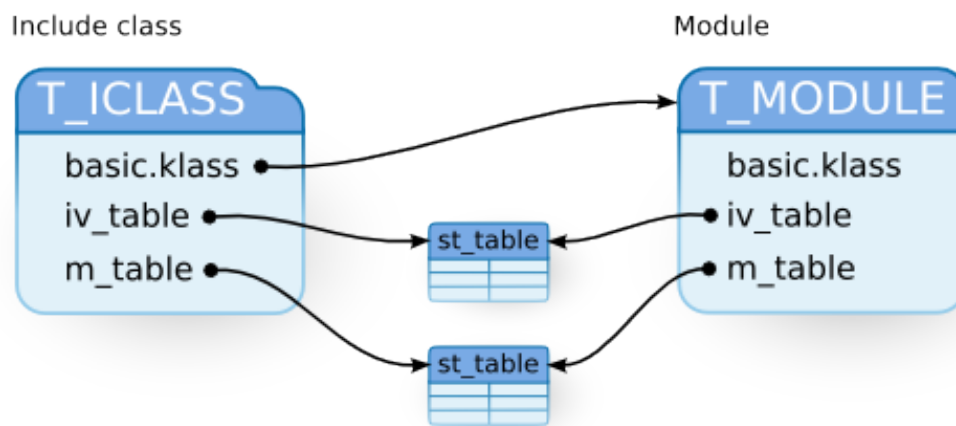


Figure 11: Include class

If you look closely at (A), the structure type flag is set to T_ICLASS. This seems to be the mark of an include class. This function's name is `include_class_new()` so ICLASS's I must be include.

And if you think about joining what this function and `rb_include_module()` do, we know that our previous expectations were not wrong. In brief, including is inserting the include class of a module between a class and its superclass (figure 12).

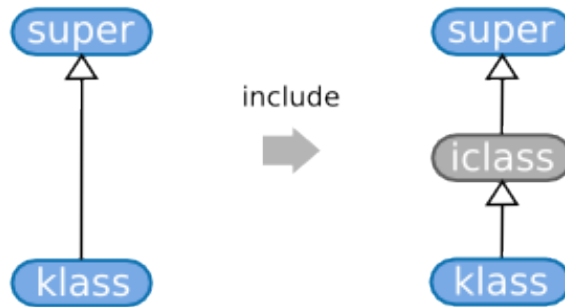


Figure 12: Include

At (D-2) the module is stored in the include class's klass. At (D-1), the module's body is taken out... at least that's what I'd like to say, but in fact this check does not have any use. The `T_ICLASS` check is already done at the beginning of this function, so when arriving here there can't still be a `T_ICLASS`. Modification to ruby piled up at a fast pace during quite a long period of time so there are quite a few small overlooks.

There is one more thing to consider. Somehow the include class's `basic.klass` is only used to point to the module's body, so for example calling a method on the include class would be very bad. So include classes must not be seen from Ruby programs. And in practice all methods skip include classes, with no exception.

Simulation

It was complicated so let's look at a concrete example. I'd like you to look at figure 13 (1). We have the `c1` class and the `m1` module that includes `m2`. From there, the changes made to include `m1` in `c1` are (2) and (3). `ims` are of course include classes.

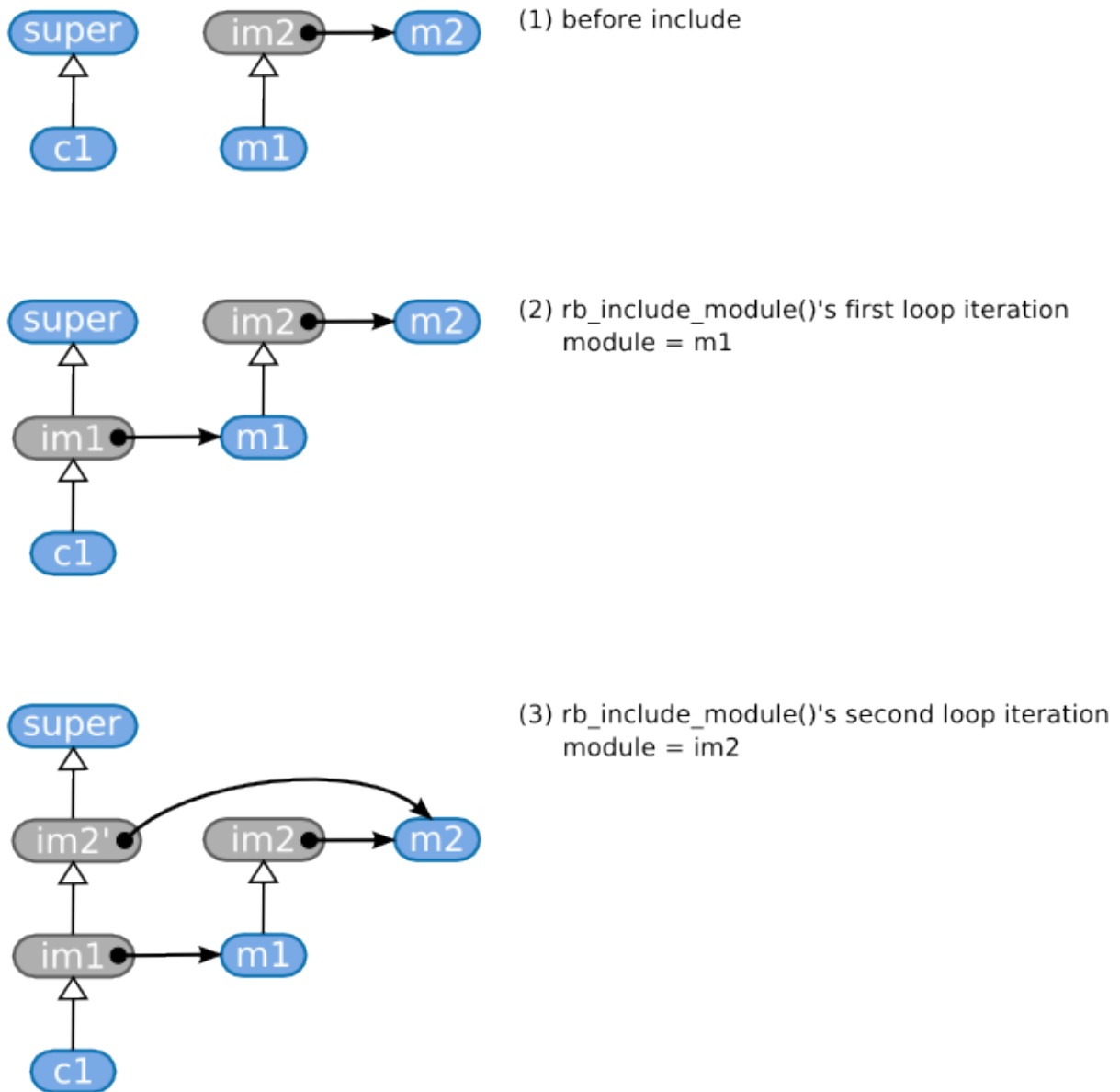


Figure 13: Include

rb_include_module (2)

Well, now we can explain the part of rb_include_module() we skipped.

rb_include_module (avoiding double inclusion)

```

378 /* (A) skip if the superclass already includes module */
379 for (p = RCLASS(klass)->super; p; p = RCLASS(p)->super) {
380     switch (BUILTIN_TYPE(p)) {
381     case T_ICLASS:
382         if (RCLASS(p)->m_tbl == RCLASS(module)->m_tbl) {
383             if (!superclass_seen) {
384                 c = p; /* the inserting point is moved */
385             }
386             goto skip;

```



```

387     }
388     break;
389     case T_CLASS:
390         superclass_seen = Qtrue;
391         break;
392     }
393 }

```

(class.c)

If one of the T_ICLASSes (include classes) that are in klass ' s superclasses (p) has the same table as one of the modules we want to include (module), it ' s an include class for module. That ' s why we skip the inclusion to not include the module twice. If this module includes an other module (module->super), we check this once more.

However, when we skip an inclusion, p is a module that has been included once, so its included modules must already be included... that ' s what I thought for a moment, but we can have the following context:

```

module M
end
module M2
end
class C
  include M # M2 is not yet included in M
end      # therefore M2 is not in C's superclasses

module M
  include M2 # as there M2 is included in M,
end
class C
  include M # I would like here to only add M2
end

```

So on the contrary, there are cases for which include does not have real-time repercussions.

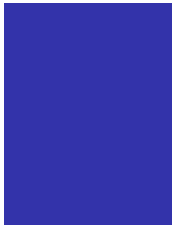
For class inheritance, the class ' s singleton methods were inherited but in the case of module there is no such thing. Therefore the singleton methods of the module are not inherited by the including class (or module). When you want to also inherit singleton methods, the usual way is to override Module#append_features.

The original work is Copyright © 2002 - 2004 Minero AOKI.

Translated by Vincent ISAMBART



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](https://creativecommons.org/licenses/by-nc-sa/2.5/).



Chapter 6: Variables and constants

Outline of this chapter

Ruby variables

In Ruby there are quite a lot of different types of variables and constants. Let 's line them up, starting from the largest scope.

- Global variables
- Constants
- Class variables
- Instance variables
- Local variables

Instance variables were already explained in chapter 2 “ Objects ” . In this chapter we ' ll talk about:

- Global variables
- Class variables
- Constants

We will talk about local variables in the third part of the book.

API for variables

The object of this chapter ' s analysis is variable.c. Let ' s first look at the available API.

```
VALUE rb_iv_get(VALUE obj, char *name)
VALUE rb_ivar_get(VALUE obj, ID name)
VALUE rb_iv_set(VALUE obj, char *name, VALUE val)
VALUE rb_ivar_set(VALUE obj, ID name, VALUE val)
```

We ' ve already spoken about those functions, but must mention them again as they are in variable.c. They are of course used for accessing instance variables.

```
VALUE rb_cv_get(VALUE klass, char *name)
VALUE rb_cvar_get(VALUE klass, ID name)
VALUE rb_cv_set(VALUE klass, char *name, VALUE val)
VALUE rb_cvar_set(VALUE klass, ID name, VALUE val)
```

These functions are the API for accessing class variables. Class variables belong directly to classes so the functions take a class as

parameter. There are in two groups, depending if their name starts with `rb_Xv` or `rb_Xvar`. The difference lies in the type of the variable “ name ” . The ones with a shorter name are generally easier to use because they take a `char*`. The ones with a longer name are more for internal use as they take a ID.

```
VALUE rb_const_get(VALUE klass, ID name)
VALUE rb_const_get_at(VALUE klass, ID name)
VALUE rb_const_set(VALUE klass, ID name, VALUE val)
```

These functions are for accessing constants. Constants also belong to classes so they take classes as parameter. `rb_const_get()` follows the superclass chain, whereas `rb_const_get_at()` does not (it just looks in klass).

```
struct global_entry *rb_global_entry(ID name)
VALUE rb_gv_get(char *name)
VALUE rb_gvar_get(struct global_entry *ent)
VALUE rb_gv_set(char *name, VALUE val)
VALUE rb_gvar_set(struct global_entry *ent, VALUE val)
```

These last functions are for accessing global variables. They are a little different from the others due to the use of `struct global_entry`. We ’ ll explain this while describing the implementation.

Important points

The most important topic of this chapter is “ Where and how are variables stored? ” , in other words: data structures.

The second most important matter is how we search for the values. The scopes of Ruby variables and constants are quite complicated because variables and constants are sometimes inherited, sometimes looked for outside of the local scope... To have a better understanding, you should first try to guess from the behavior how it could be implemented, then compare that with what is really done.

Class variables

Class variables are variables that belong to classes. In Java or C++ they are called static variables. They can be accessed from both the class or its instances. But “ from an instance ” or “ from the class ” is information only available in the evaluator, and we do not have one for the moment. So from the C level it ’ s like having no access range. We ’ ll just focus on the way these variables are stored.

Reading

The functions to get a class variable are `rb_cvar_get()` and `rb_cv_get()`. The function with the longer name takes ID as parameter and the one with the shorter one takes `char*`. Because the one taking an ID seems closer to the internals, we ’ ll look at it.

```
rb_cvar_get()
```

```
1508 VALUE
1509 rb_cvar_get(klass, id)
1510     VALUE klass;
1511     ID id;
```

```

1512 {
1513     VALUE value;
1514     VALUE tmp;
1515
1516     tmp = klass;
1517     while (tmp) {
1518         if (RCLASS(tmp)->iv_tbl) {
1519             if (st_lookup(RCLASS(tmp)->iv_tbl,id,&value)) {
1520                 if (RTEST(ruby_verbose)) {
1521                     cvar_override_check(id, tmp);
1522                 }
1523                 return value;
1524             }
1525         }
1526         tmp = RCLASS(tmp)->super;
1527     }
1528
1529     rb_name_error(id,"uninitialized class variable %s in %s",
1530                 rb_id2name(id), rb_class2name(klass));
1531     return Qnil;      /* not reached */
1532 }

```

(variable.c)

This function reads a class variable in klass.

Error management functions like `rb_raise()` can be simply ignored like I said before. The `rb_name_error()` that appears this time is a function for raising an exception, so it can be ignored for the same reasons. In ruby, you can assume that all functions ending with `_error` raise an exception.

After removing all this, we can see that while following the klass ' s superclass chain we only search in `iv_tbl`. At this point you should say “ What? `iv_tbl` is the instance variables table, isn ' t it? ” As a matter of fact, class variables are stored in the instance variable table.

We can do this because when creating IDs, the whole name of the variables is taken into account, including the prefix: `rb_intern()` will return different IDs for “ @var ” and “ @@var ” . At the Ruby level, the variable type is determined only by the prefix so there ' s no way to access a class variable called `@var` from Ruby.

Constants

It ' s a little abrupt but I ' d like you to remember the members of struct `RClass`. If we exclude the basic member, struct `RClass` contains:

```

VALUE super
struct st_table *iv_tbl
struct st_table *m_tbl

```

Then, considering that:

1. constants belong to a class
2. we can ' t see any table dedicated to constants in struct `RClass`
3. class variables and instance variables are both in `iv_tbl`

Could it mean that the constants are also...

Assignment

`rb_const_set()` is a function to set the value of constants: it sets the constant `id` in the class `klass` to the value `val`.

`rb_const_set()`

```
1377 void
1378 rb_const_set(klass, id, val)
1379     VALUE klass;
1380     ID id;
1381     VALUE val;
1382 {
1383     mod_av_set(klass, id, val, Qtrue);
1384 }
```

(variable.c)

`mod_av_set()` does all the hard work:

`mod_av_set()`

```
1352 static void
1353 mod_av_set(klass, id, val, isconst)
1354     VALUE klass;
1355     ID id;
1356     VALUE val;
1357     int isconst;
1358 {
1359     char *dest = isconst ? "constant" : "class variable";
1360
1361     if (!OBJ_TAINTED(klass) && rb_safe_level() >= 4)
1362         rb_raise(rb_eSecurityError, "Insecure: can't set %s", dest);
1363     if (OBJ_FROZEN(klass)) rb_error_frozen("class/module");
1364     if (!RCLASS(klass)->iv_tbl) {
1365         RCLASS(klass)->iv_tbl = st_init_numtable();
1366     }
1367     else if (isconst) {
1368         if (st_lookup(RCLASS(klass)->iv_tbl, id, 0) ||
1369             (klass == rb_cObject && st_lookup(rb_class_tbl, id, 0))) {
1370             rb_warn("already initialized %s %s", dest, rb_id2name(id));
1371         }
1372     }
1373
1374     st_insert(RCLASS(klass)->iv_tbl, id, val);
1375 }
```

(variable.c)

You can this time again ignore the warning checks (`rb_raise()`, `rb_error_frozen()` and `rb_warn()`). Here's what's left:

`mod_av_set()` (only the important part)

```
if (!RCLASS(klass)->iv_tbl) {
    RCLASS(klass)->iv_tbl = st_init_numtable();
}
st_insert(RCLASS(klass)->iv_tbl, id, val);
```

We ' re now sure constants also reside in the instance table. It means in the `iv_tbl` of struct `RClass`, the following are mixed together:

1. the class ' s own instance variables
2. class variables
3. constants

Reading

We now know how the constants are stored. We ' ll now check how they really work.

`rb_const_get()`

We ' ll now look at `rconst_get()`, the function to read a constant. This functions returns the constant referred to by `id` from the class `klass`.

`rb_const_get()`

```
1156 VALUE
1157 rb_const_get(klass, id)
1158     VALUE klass;
1159     ID id;
1160 {
1161     VALUE value, tmp;
1162     int mod_retry = 0;
1163
1164     tmp = klass;
1165 retry:
1166     while (tmp) {
1167         if (RCLASS(tmp)->iv_tbl &&
1168             st_lookup(RCLASS(tmp)->iv_tbl, id, &value)) {
1169             return value;
1170         }
1171         if (tmp == rb_cObject && top_const_get(id, &value))
1172             return value;
1173         tmp = RCLASS(tmp)->super;
1174     }
1175     if (!mod_retry && BUILTIN_TYPE(klass) == T_MODULE) {
1176         mod_retry = 1;
1177         tmp = rb_cObject;
1178         goto retry;
1179     }
1180     /* Uninitialized constant */
1181     if (klass && klass != rb_cObject) {
1182         rb_name_error(id, "uninitialized constant %s at %s",
```

```

1182         rb_id2name(id),
1183         RSTRING(rb_class_path(klass))->ptr);
1184     }
1185     else { /* global_uninitialized */
1186         rb_name_error(id, "uninitialized constant %s", rb_id2name(id));
1187     }
1188     return Qnil;          /* not reached */
1189 }

```

(variable.c)

There ' s a lot of code in the way. First, we should at least remove the `rb_name_error()` in the second half. In the middle, what ' s around `mod_entry` seems to be a special handling for modules. Let ' s also remove that for the time being. The function gets reduced to this:

`rb_const_get` (simplified)

```

VALUE
rb_const_get(klass, id)
    VALUE klass;
    ID id;
{
    VALUE value, tmp;

    tmp = klass;
    while (tmp) {
        if (RCLASS(tmp)->iv_tbl && st_lookup(RCLASS(tmp)->iv_tbl, id, &value)) {
            return value;
        }
        if (tmp == rb_cObject && top_const_get(id, &value)) return value;
        tmp = RCLASS(tmp)->super;
    }
}

```

Now it should be pretty easy to understand. The function searches for the constant in `iv_tbl` while climbing `klass` ' s superclass chain. That means:

```

class A
  Const = "ok"
end
class B < A
  p(Const) # can be accessed
end

```

The only problem remaining is `top_const_get()`. This function is only called for `rb_cObject` so `top` must mean “ top-level ” . If you don ' t remember, at the top-level, the class is `Object`. This means the same as “ in the class statement defining `C`, the class becomes `C` ” , meaning that “ the top-level ' s class is `Object` ” .

```

# the class of the top-level is Object
class A
  # the class is A
class B
  # the class is B

```

```
end
end
```

So `top_const_get()` probably does something specific to the top level.

top_const_get()

Let ' s look at this `top_const_get` function. It looks up the id constant writes the value in `klassp` and returns.

```
top_const_get()
```

```
1102 static int
1103 top_const_get(id, klassp)
1104     ID id;
1105     VALUE *klassp;
1106 {
1107     /* pre-defined class */
1108     if (st_lookup(rb_class_tbl, id, klassp)) return Qtrue;
1109
1110     /* autoload */
1111     if (autoload_tbl && st_lookup(autoload_tbl, id, 0)) {
1112         rb_autoload_load(id);
1113         *klassp = rb_const_get(rb_cObject, id);
1114         return Qtrue;
1115     }
1116     return Qfalse;
1117 }
```

(variable.c)

`rb_class_tbl` was already mentioned in chapter 4 “ Classes and modules ” . It ' s the table for storing the classes defined at the top-level. Built-in classes like `String` or `Array` have for example an entry in it. That ' s why we should not forget to search in this table when looking for top-level constants.

The next block is related to autoloading. This allows us to automatically load a library when accessing a top-level constant for the first time. This can be used like this:

```
autoload(:VeryBigClass, "verybigclass") # VeryBigClass is defined in it
```

After this, when `VeryBigClass` is accessed for the first time, the `verybigclass` library is loaded (with `require`). As long as `VeryBigClass` is defined in the library, execution can continue smoothly. It ' s an efficient approach, when a library is too big and a lot of time is spent on loading.

This autoload is processed by `rb_autoload_xxxx()`. We won ' t discuss autoload further in this chapter because there will probably be a big change in how it works soon (The way autoload works *did* change in 1.8: autoloaded constants do not need to be defined at top-level anymore).

Other classes?

But where did the code for looking up constants in other classes end up? After all, constants are first looked up in the outside classes, then in the superclasses.

In fact, we do not yet have enough knowledge to look at that. The outside classes change depending on the location in the program. In other words it depends of the program context. So we need first to understand how the internal state of the evaluator is handled. Specifically, this search in other classes is done in the `ev_const_get()` function of `eval.c`. We ' ll look at it and finish with the constants in the third part of the book.

Global variables

General remarks

Global variables can be accessed from anywhere. Or put the other way around, there is no need to restrict access to them. Because they are not attached to any context, the table only has to be at one place, and there ' s no need to do any check. Therefore implementation is very simple.

But there is still quite a lot of code. The reason for this is that global variables are quite different from normal variables. Functions like the following are only available for global variables:

```
you can “ hook ” access of global variables
you can alias them with alias
```

Let ' s explain this simply.

Aliases of variables

```
alias $newname $oldname
```

After this, you can use `$newname` instead of `$oldname`. `alias` for variables is mainly a counter-measure for “ symbol variables ” . “ symbol variables ” are variables inherited from Perl like `$=` or `$0`. `$=` decides if during string comparison upper and lower case letters should be differentiated. `$0` shows the name of the main Ruby program. There are some other symbol variables but anyway as their name is only one character long, they are difficult to remember for people who don ' t know Perl. So, aliases were created to make them a little easier to understand.

That said, currently symbol variables are not recommended, and are moved one by one in singleton methods of suitable modules. The current school of thought is that `$=` and others will be abolished in 2.0.

Hooks

You can “ hook ” read and write of global variables.

Hooks can be also be set at the Ruby level, but I was thinking: why not instead look at C level special variables for system use like `$KCODE`? `$KCODE` is the variable containing the encoding the interpreter currently uses to handle strings. It can only be set to special

values like "EUC" or "UTF8". But this is too bothersome so it can also be set to "e" or "u".

```
p($KCODE)  # "NONE" (default)
$KCODE = "e"
p($KCODE)  # "EUC"
$KCODE = "u"
p($KCODE)  # "UTF8"
```

Knowing that you can hook assignment of global variables, you should understand easily how this can be done. By the way, \$KCODE ' s K comes from “ kanji ” (the name of Chinese characters in Japanese).

You might say that even with alias or hooks, global variables just aren ' t used much, so it ' s functionality that doesn ' t really mater. It ' s adequate not to talk much about unused functions, and I need some pages for the analysis of the parser and evaluator. That ' s why I ' ll proceed with the explanation below throwing away what ' s not really important.

Data structure

When we were looking at how variables work, I said that the way they are stored is important. That ' s why I ' d like you to firmly grasp the structure used by global variables.

Data structure for global variables

```
21 static st_table *rb_global_tbl;

334 struct global_entry {
335     struct global_variable *var;
336     ID id;
337 };

324 struct global_variable {
325     int counter; /* reference counter */
326     void *data; /* value of the variable */
327     VALUE (*getter)(); /* function to get the variable */
328     void (*setter)(); /* function to set the variable */
329     void (*marker)(); /* function to mark the variable */
330     int block_trace;
331     struct trace_var *trace;
332 };

(variable.c)
```

rb_global_tbl is the main table. All global variables are stored in this table. The keys of this table are of course variable names (ID). A value is expressed by a struct global_entry and a struct global_variable (figure 1).

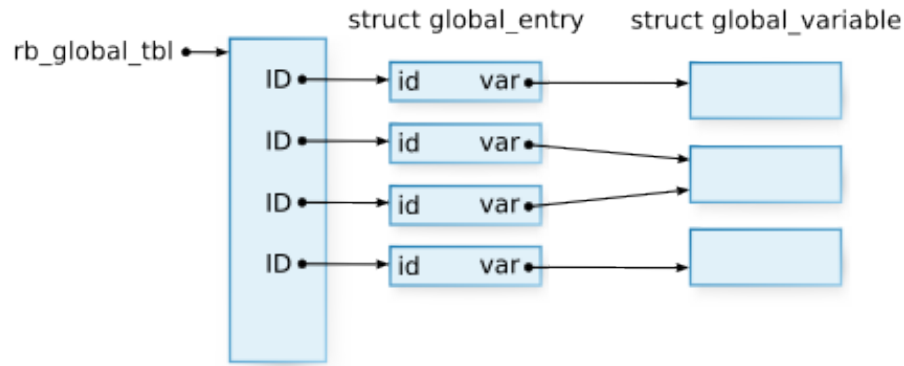


Figure 1: Global variables table at execution time

The structure representing the variables is split in two to be able to create aliases. When an alias is established, two `global_entry`s point to the same `struct global_variable`.

It ' s at this time that the reference counter (the counter member of `struct global_variable`) is necessary. I explained the general idea of a reference counter in the previous section “ Garbage collection ” . Reviewing it briefly, when a new reference to the structure is made, the counter is incremented by 1. When the reference is not used anymore, the counter is decreased by 1. When the counter reaches 0, the structure is no longer useful so `free()` can be called.

When hooks are set at the Ruby level, a list of `struct trace_vars` is stored in the `trace` member of `struct global_variable`, but I won ' t talk about it, and omit `struct trace_var`.

Reading

You can have a general understanding of global variables just by looking at how they are read. The functions for reading them are `rb_gv_get()` and `rb_gvar_get()`.

`rb_gv_get()` `rb_gvar_get()`

```

716 VALUE
717 rb_gv_get(name)
718     const char *name;
719 {
720     struct global_entry *entry;
721
722     entry = rb_global_entry(global_id(name));
723     return rb_gvar_get(entry);
724 }

649 VALUE
650 rb_gvar_get(entry)
651     struct global_entry *entry;
652 {
653     struct global_variable *var = entry->var;
654     return (*var->getter)(entry->id, var->data, var);
655 }
```

(variable.c)

A substantial part of the content seems to turn around the `rb_global_entry()` function, but that does not prevent us understanding what 's going on. `global_id` is a function that converts a `char*` to ID and checks if it 's the ID of a global variable. `(*var->getter)(...)` is of course a function call using the function pointer `var->getter`. If `p` is a function pointer, `(*p)(arg)` calls the function.

But the main part is still `rb_global_entry()`.

`rb_global_entry()`

```

351 struct global_entry*
352 rb_global_entry(id)
353     ID id;
354 {
355     struct global_entry *entry;
356
357     if (!st_lookup(rb_global_tbl, id, &entry)) {
358         struct global_variable *var;
359         entry = ALLOC(struct global_entry);
360         st_add_direct(rb_global_tbl, id, entry);
361         var = ALLOC(struct global_variable);
362         entry->id = id;
363         entry->var = var;
364         var->counter = 1;
365         var->data = 0;
366         var->getter = undef_getter;
367         var->setter = undef_setter;
368         var->marker = undef_marker;
369
370         var->block_trace = 0;
371         var->trace = 0;
372     }
373     return entry;
374 }
```

(variable.c)

The main treatment is only done by the `st_lookup()` at the beginning. What 's done afterwards is just creating (and storing) a new entry. As, when accessing a non existing global variable, an entry is automatically created, `rb_global_entry()` will never return `NULL`.

This was mainly done for speed. When the parser finds a global variable, it gets the corresponding `struct global_entry`. When reading the value of the variable, the parser just has to get the value from the entry (using `rb_gv_get()`), and has no need to do any check.

Let 's now continue a little with the code that follows. `var->getter` and others are set to `undef_xxxx`. `undef` means that the global setter/getter/marker for the variable are currently undefined.

`undef_getter()` just shows a warning and returns `nil`, as even undefined global variables can be read. `undef_setter()` is quite interesting so let 's look at it.

`undef_setter()`

```

385 static void
386 undef_setter(val, id, data, var)
387     VALUE val;
388     ID id;
```

```

389 void *data;
390 struct global_variable *var;
391 {
392     var->getter = val_getter;
393     var->setter = val_setter;
394     var->marker = val_marker;
395
396     var->data = (void*)val;
397 }

```

(variable.c)

`val_getter()` takes the value from `entry->data` and returns it. `val_getter()` just puts a value in `entry->data`. Setting handlers this way allows us not to need special handling for undefined variables (figure 2). Skillfully done, isn't it?

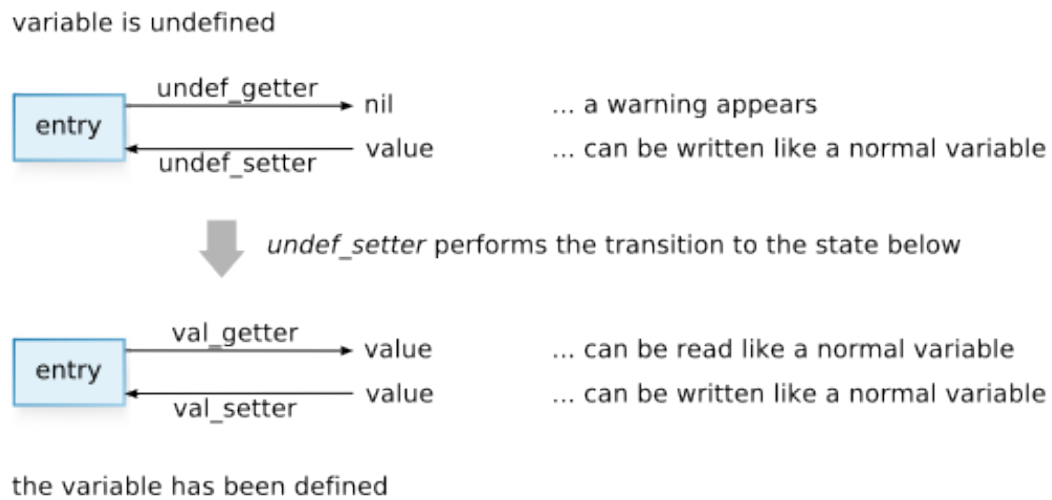


Figure 2: Setting and consultation of global variables

The original work is Copyright © 2002 - 2004 Minero AOKI.

Translated by Vincent ISAMBART



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](https://creativecommons.org/licenses/by-nc-sa/2.5/).