

Ruby/DL

立石 孝彰

平成 14 年 6 月 17 日

目次

1	チュートリアル	1
1.1	ライブラリ関数のインポート	1
1.2	型の定義	2
1.3	配列, ポインタ型データ, シンボル	2
1.4	コールバック関数	2
1.5	参照される引数	3
2	Ruby/DL 詳説	3
2.1	関数の呼び出し方	3
2.2	mutable な引数	4
2.3	ポインタの扱い方	4
2.4	配列の扱い方	5
2.5	構造体の扱い方	6
2.6	コールバック関数の扱い方	7
2.7	free 関数の定義	8
3	Ruby/DL マニュアル	8
3.1	DL モジュール	8
3.2	DL::Handle クラス	8
3.3	DL::Symbol クラス	8
3.4	DL::PtrData クラス	8
3.5	型指定子	8
3.6	DL::MemorySpace モジュール	8
4	他の拡張ライブラリとの連携	8

1 チュートリアル

ほとんどの OS では, *DLL* や *shared* ライブラリというプログラムが実行時にリンクするオブジェクトファイルがあります. 本書ではこれらのオブジェクトファイルを総称して, システムライブラリと呼ぶことにします. 本節では, Ruby/DL の一部である `import.rb` を

用いてシステムライブラリ内で定義されている関数 (以降では単純にライブラリ関数と呼ぶ) を利用する方法を紹介します. ライブラリ関数を使うには, 他にも DL モジュールに定義されている基本的な例レベルな関数を使って行うこともできます. この方法については, 後の 2 節で紹介します. `import.rb` では `DL::Importable` というモジュールが定義されています. このモジュールを使うと, 視覚的に見易く Ruby からライブラリ関数を利用することができます.

1.1 ライブラリ関数のインポート

`import.rb` を使って, Ruby からライブラリ関数を呼び出すためには, 次の 3 つのステップを行わなければなりません. (1) システムライブラリ用のモジュールの定義. (2) 使用するシステムライブラリのロード. (3) 使用するライブラリ関数を Ruby から使えるようにする. システムライブラリ用のモジュールは, `DL::Importable` というモジュールを `extend` して作る必要があります. `DL::Importable` には, ライブラリをロードするメソッドや, ライブラリ関数を利用するためのメソッドが定義されているためです. まず, システムライブラリ用のモジュールを定義することから説明します. 例えば, `libc.so` という C ライブラリを利用する場合, 次のようなモジュールを定義します.

```
require 'dl/import'  
module LIBC  
  extend DL::Importable  
end
```

ここで, `LIBC` という名前は他のものでも良いですが, 必ず `DL::Importable` を `extend` して定義して下さい. システムライブラリをロードするには, この `LIBC` 内で `dlopen` という関数を使って行います. そして, `extern`

という関数を使って、ライブラリ関数を Ruby から利用できるようにします。このようにライブラリ関数を Ruby から利用できるようにすることをインポートと呼びます。

```
require 'dl/import'
module LIBC
  extend DL::Importable
  dllload "libc.so"
  extern "int strlen(const char *)"
end
```

上記の例では、libc.so というライブラリに含まれている strlen() という関数を利用できるようにしました。strlen() は標準的な C ライブラリ関数の一つで、文字列の長さを返します。dllload は引数に、使いたいシステムライブラリを複数指定することができます。extern には、C の関数プロトタイプを与えます。しかし、あらゆる型を定義しているわけではないので、定義されていない型については、使える型だけを利用するか、1.1.2 節の方法を用いて、各自で型を新しく定義しなければなりません。この関数を LIBC モジュールを利用して Ruby から呼ぶには次のようにします。

```
LIBC.strlen("abc") # => 3
```

返り値として 3 が得られるはずですが、Ruby では、メソッド名の頭文字に大文字を使うことはできませんが、C などでは、大文字を使うことができます。そのため、インポートする関数名の頭文字が大文字の場合は、自動的に小文字に置き換わるようになっています。例えば、GetString() というライブラリ関数であれば、getString というメソッドを Ruby から使うことになります。

1.2 型の定義

Ruby/DL では一般的な C の型は解釈しますが、計算機環境やライブラリに特有の型などはほとんどサポートしていません。そこで、プロトタイプを記述するときに、それらのサポートされていない型を使いたい場合には予めモジュール内で定義をしておく必要があります。たとえば、size_t 型は Ruby/DL では定義されていません。これを unsigned int 型として定義するためには次の一行を加えます。

```
typealias("size_t", "unsigned int")
```

括弧を省略して次のように書くこともできます。

```
typealias "size_t", "unsigned int"
```

1.3 配列、ポインタ型データ、シンボル

1.4 コールバック関数

いくつかのライブラリ関数では、コールバック関数を必要とすることがあります。たとえば、C のライブラリ関数 qsort() は、要素の比較を行うための関数へのポインタを必要とします。

```
void qsort(
  void *base,
  size_t nmem,
  size_t size,
  int (*compar)(const void *, const void *)
)
```

まず qsort() を Ruby から使えるように以下の LIBC モジュールを定義します。

```
require 'dl/import'
module LIBC
  extend DL::Importable
  dllload "libc.so"
  typealias("size_t", "unsigned int")
  extern "void *qsort(size_t, size_t, void*)"
end
```

関数の型 int (*)(...) は Ruby/DL では解釈されないためポインタ型として定義することにしています。次に、コールバック関数を Ruby で定義します。

```
module LIBC
  def my_compare(ptr1, ptr2)
    ptr1.ptr.to_s <=> ptr2.ptr.to_s
  end

  MY_COMPARE = callback "int my_compare(char**, char**)"
end
```

まず、my_compare という Ruby のメソッドを定義しています。このメソッドは、2つの文字列へのポインタ

を受け取り、それぞれの文字列の比較を行うものです。次に、定義したメソッドを C の関数に対応づけるために `callback` メソッドを使います。引数には、定義したメソッドと同名の関数名を使ってプロトタイプを記述します。callback による戻り値は Symbol オブジェクトという C のシンボルを表すオブジェクトです。

```
ary = ["a", "c", "b"]
ptr = ary.to_ptr
LIBC.qsort(ptr, ary.length, LIBC::MY_COMPARE)
```

1.5 参照される引数

2 Ruby/DL 詳説

本節では、C ライブラリの中の関数を例として、Ruby/DL の使い方を一通り説明していくことにします。C ライブラリは、多くの Unix オペレーティングシステムでは、`/lib/libc.so` あるいは `/usr/lib/libc.so` という名前で存在すると思いますが、ここでは `/usr/lib/libc.so` を使うものとします。対象とする読者としては、C についての基礎程度の知識はあるものとしています。少なくとも、ポインタに関する知識がないと理解するのは難しいと思います。

2.1 関数の呼び出し方

C ライブラリ内の関数 `isdigit()` と `atoi()` を Ruby から利用する例を挙げます。まず、`libc.so` というライブラリを扱うために、`DL.dlopen` というモジュール関数を使います。

```
require 'dl'
libc = DL.dlopen('/usr/lib/libc.so')
```

`DL.dlopen(path)` によって、`path` で示されたライブラリをオープンして、`DL::Handle` クラスのインスタンスである `Handle` オブジェクトを返します。上記の例では変数 `libc` は `Handle` オブジェクトを表すことになります。

次に、C ライブラリ関数である `atoi()`、`isdigit()` の関数へのポインタを取り出します。関数ポインタを取得するには `Handle#sym` というメソッドを使うか、あるいは短縮形である `Handle#[]` を使います。

```
atoi = libc['atoi', 'IS']
isdigit = libc['isdigit', 'II']
```

Ruby/DL では、この関数へのポインタを `DL::Symbol` というクラスで扱っています。このため、変数 `atoi`、`isdigit` は `DL::Symbol` のインスタンスを表すこととなります。第 1 引数では、取得する関数のシンボルを文字列で指定します。そして、第 2 引数では、その関数の型を指定します。型は文字列で表現し、型指定子と呼ばれる文字から構成します。1 番目の文字が関数の戻り値の型を表し、`k` 番目の文字が関数の (`k-1`) 番目の引数の型を表しています。`atoi` の場合、戻り値の型は `'I'` なので C の型 `'int'` であることを表し、`'S'` は `'const char *'` であることを表しています。戻り値が必要ない場合には 1 番目の文字に `'0'` (ゼロ) を使います。型指定子については、3.5 節で詳しく解説しています。

取得した関数ポインタに引数を与えて実行するには `call` というメソッドを呼びます。

```
r1 = isdigit.call(?1) # => [2048, [49]]
r2 = isdigit.call(?a) # => [0, [97]]
r3 = atoi.call('10') # => [10, ["10"]]
```

また、`call` の代わりに `[]` を使うこともできます。

```
r1 = isdigit[?1] # => [2048, [49]]
r2 = isdigit[?a] # => [0, [97]]
r3 = atoi['10'] # => [10, ["10"]]
```

`r1`、`r2`、`r3` には、戻り値と与えた引数の関数適用後の値のリストが配列となって代入されています。それぞれの右側に書かれたコメントが `Symbol#call`、`[]` で返ってくる値の例です。

以上の一連の流れを使って、文字列がすべて数字であれば、`atoi` を使って整数へ変換する `str2int` という関数を定義する場合は次の通りです。

```
require 'dl'

module LIBC
  LIB = DL.dlopen('/usr/lib/libc.so')
  SYM = {
    :atoi => LIB['atoi', 'IS']
    :isdigit => LIB['isdigit', 'II']
  }
end
```

```

def atoi(str)
  r,rs = SYM[:atoi].call(str)
  return r
end

def isdigit(c)
  r,rs = SYM[:isdigit].call(c)
  return (r == 0)
end

include LIBC
def str2int(str)
  str.each_byte{|c|
    if( LIBC.isdigit(c) )
      return nil
    end
  }
  return LIBC.atoi(str)
end

```

この例では、LIBC モジュールによって、libc.so の機能を Ruby のモジュールとして提供しています。定数 LIB は、ライブラリへの Handle オブジェクトを表し、SYM には Symbol オブジェクトのためのハッシュを定義しています。モジュール関数 atoi, isdigit は、libc 内の関数 atoi, isdigit をそれぞれ表しています。

2.2 mutable な引数

C の関数では関数適用前と適用後で引数の値が変わる場合があります。例えば、strcat() は第 1 引数で与えた文字列に、第 2 引数で与えた文字列を接続します。このため、第 1 引数の文字列は関数適用後には変化します。このような mutable な引数を扱うために小文字の型指定子 's' があります。先ほどの LIBC モジュールにモジュール関数 strcat を加えると次の通りです。

```

module LIBC
  SYM[:strcat] = LIB['strcat', 'SsS']
  def strcat(str1, str2)
    r,rs =
      SYM[:strcat].call("#{str1}\0#{str2}", str2)
  end
end

```

```

    return rs[0]
  end
end

```

ここで注意しなければならないのは、SYM[:strcat].call の第 1 引数において、単純に str1 を与えるのではなく、("#{str1}\0#{str2}") を与えている点です。これは、C において strcat() を使う場合、第 1 引数には関数適用後に生成される文字列に対して十分なサイズのバッファを用意しておかなければならないためです。さて、このモジュール関数に引数として文字列 'abc', 'def' を与えると、結果として 'abcdef' が返ってきます。

```

include LIBC
LIBC.strcat("abc", "def") # => "abcdef"

```

2.3 ポインタの扱い方

C でプログラムを作る際に、ポインタを使うことは多いです。例えばあるファイルを開くには fopen() という関数を使います。この関数は、引数を 2 つとります。第 1 引数にはファイルの名前を文字列として与えます。第 2 引数にはファイルを開く際のモードを文字列で与えます。返り値は FILE 構造体へのポインタです。このため、fopen() の型は Ruby/DL では 'PSS' と表します。型指定子 'P' はポインタ型を表します。fopen(), fclose(), fgetc() を Ruby で扱うための LIBC のモジュール関数を以下に定義します。

```

module LIBC
  SYM[:fopen] = LIB['fopen', 'PSS']
  SYM[:fclose] = LIB['fclose', 'OP']
  SYM[:fgetc] = LIB['fgetc', 'IP']

  def fopen(filename, mode)
    r,rs = SYM[:fopen].call(filename, mode)
    return r
  end

  def fclose(ptr)
    SYM[:fclose].call(ptr)
    return nil
  end
end

```

```

def fgetc(ptr)
  r,rs = SYM[:fgetc].call(ptr)
  return r
end
end

```

LIBC.fopen では、ファイル名とオープン時のモードを指定することによってファイルポインタを取得できます。ファイルポインタは DL::PtrData クラスのインスタンスとなっています。PtrData クラスは、ポインタを Ruby から扱うためのクラスで、ポインタ操作のためのメソッドが定義されています。LIBC モジュールに定義されたこれらの関数を使ってファイルの内容を出力する関数を定義すると次の通りです。

```

include LIBC
def print_file(filename)
  fp = LIBC.fopen(filename, "r")
  if( !fp )      # -- (注1)
    return nil
  end
  while( (c = LIBC.fgetc(fp)) > 0 )
    print(c.chr)
  end
  LIBC.fclose(fp)
end

```

Cにおける NULL は、Ruby では nil として扱うようになっています。このため、ファイルのオープンに失敗したことを検出するために、LIBC.fopen から返ってくる値が nil かどうかを (注1) の部分で調べています。

Ruby/DL では、型指定子に 'P' あるいは 'p' を指定すると関数呼出時に引数に与えたオブジェクトが PtrData オブジェクトであるかどうかをまず調べます。もし PtrData オブジェクトでない場合には、PtrData オブジェクトへ変換するためのメソッド to_ptr を使って PtrData オブジェクトへ変換します。この仕組みを利用して、C の配列や文字列などもポインタとして同様に扱うこともできます。Ruby/DL モジュールを require した後は String#to_ptr や Array#to_ptr などが定義されているため、例えば、これまでに述べた 'S' や 's' などの文字列を表す型指定子の代わりに、'P' や 'p' を使うこともできます。例として、strlen() という

C のライブラリ関数を LIBC モジュールで扱うと次の通りです。

```

module LIBC
  SYM[:strlen] = LIB['strlen', 'IP']
  def strlen(str)
    r,rs = SYM[:strlen].call(str)
    return r
  end
end

```

2.4 配列の扱い方

Ruby の配列 (Array) を C の関数に渡すためには、型指定子 'A' あるいは 'a' を使います。例えば、glibc2 に含まれる C ライブラリ関数 qsort() は次の通りのプロトタイプを持ちます。

```

void qsort(
  void *base,
  size_t nmem,
  size_t size,
  int (*compar)(const void *, const void *)
)

```

このため、Ruby/DL では '0aIP' という型指定子を使います。qsart() 関数は汎用のソート関数ですが、今回は文字列の配列をソートすることだけを考えることにします。また、仮引数 compare に与える関数を Ruby から定義する方法については 2.6 節で同じ qsort の例を用いて説明しますので、ここでは次の関数を自分でライブラリ化したものを使います。この関数は 2 つの文字列を比較するための関数です。

```

#include <string.h>
int mystrcmp(char **str1, char **str2)
{
  return strcmp(*str1, *str2);
}

```

この関数を含むライブラリの名前を 'libmy.so' とすることにします。さて、このライブラリ用の次のような Ruby のモジュールを作成します。

```

module LIBMY
  LIB = DL.dlopen('libmy.so')

```

```
SYM = {:mystricmp => LIB['mystricmp', 'IPP']}
end
```

すると、LIBMY.SYM[:mystricmp] によって、mystricmp() への関数へのポインタを取得することができます。最後に、qsort() 関数を扱うための Ruby 側の LIBC モジュールの関数を次の通りに定義します。

```
module LIBC
  SYM[:qsort] = LIB['qsort', '0aIIP']
  def qsort(ary, comp)
    len = ary.length
    r,rs = SYM[:qsort].call(ary,
                           len,
                           DL.sizeof('P'),
                           comp)
    return rs[0].to_a('S', len)
  end
end
```

rs[0] は第 1 引数の関数適用後の値ですが、型指定子 'a' を使うと、Ruby 側で取得できる値は PtrData オブジェクトとなっています。そのため、rs[0] を to_a というメソッドを使って配列へと変換しています。このとき、ポインタをどのように配列に戻すのかを 2 つの引数によって決めます。第 1 引数によって配列要素の型を与え、第 2 引数によって配列の要素数を与えます。今回の場合、配列の要素の型は文字列型なので、第 1 引数には 'S' を与えています。また、配列の要素数は関数適用前と変化はないので、予め取得していた配列の要素数を第 2 引数に与えています。こうして、LIBC.qsort は、文字列の配列を与えると comp で指定した C の関数を使ってソートした結果を再び文字列の配列として返します。LIBC モジュールと LIBMY モジュールを使って、文字列の配列を mystricmp() を使ってソートする Ruby の関数を定義すると次の通りになります。

```
include LIBC
include LIBMY

def strsort(ary)
  LIBC.qsort(ary, LIBMY[:mystricmp])
end
```

2.5 構造体の扱い方

C では多くの場面で構造体を定義することによって、新しいデータ型を作ることができます。例えば、gettimeofday() で使用される timeval 構造体は次のように定義されています。

```
struct timeval {
  long tv_sec;
  long tv_usec;
};
```

このような構造体のためにメモリを確保する方法として、DL.malloc か又は、Array#pack を使うことができます。DL.malloc の引数には確保するサイズを与えます。すると、確保した領域を示す PtrData オブジェクトが返ってきます。この際、DL.sizeof を使うことで、指定した型のサイズを得ることができます。DL.sizeof を使うと、構造体に必要なサイズも計算できます。このとき、アライメントも考慮されて計算されます。下記の (1),(2) は同じことを行なっているように見えますが、(1) は char 型の変数と long 型の変数を持つ構造体に必要なサイズを表しており、(2) は単純に char 型のサイズと long 型のサイズの和を計算しているだけになります。

```
ptr = DL.malloc(DL.sizeof('CL')) # (1)
ptr = DL.malloc(DL.sizeof('C') +
                DL.sizeof('L')) # (2)
```

Array#pack を使う場合は、一度 Ruby の文字列を作ってから to_ptr によって PtrData オブジェクトへ変換します。

```
ptr = [0,0].pack('ll').to_ptr
```

pack を使うと各要素の初期化も同時に行なうことができます。

構造体を PtrData オブジェクトで表した場合、オフセットを指定することでそれぞれの要素を取り出すことができる場合があります。例えば、次の通りにして tv_sec と tv_usec に相当する要素を得ることができます。

```
sec = ptr[0].to_i
usec = ptr[DL.sizeof('L')].to_i
```

比較のために C でのコードと Ruby のコードの対応を次に示します。¹

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main()
{
    void *ptr;
    long sec, usec;

    /* ptr = [0,0].pack('ll').to_ptr */
    ptr = (void*)malloc(sizeof(struct timeval));
    ((struct timeval *)ptr)->tv_sec = 10;
    ((struct timeval *)ptr)->tv_usec = 100;

    /* sec = ptr[0].to_i */
    /* sec = (ptr + 0).ptr.to_i */
    sec = *(long*)(ptr + 0);

    /* usec = ptr[DL.sizeof('L')].to_i */
    /* usec = (ptr + DL.sizeof('L')).ptr.to_i */
    usec = *(long*)(ptr + sizeof(long));

    printf("sec = %ld, usec = %ld\n", sec, usec);
    exit(0);
}
```

しかし、通常は、構造体のメンバへのアクセスのために、構造体の先頭アドレスからのオフセットを指定して、さらにキャストを使ってアクセスするということを行いません。C で書くと次のようなコードを書くことでしょう。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main()
{
    struct timeval *ptr;
    long sec, usec;
```

¹C コンパイラによっては警告が出るか、あるいは通らないかもしれませんが。

```
ptr =
    (struct timeval *)malloc(sizeof(struct timeval));
ptr->tv_sec = 10;
ptr->tv_usec = 100;

sec = ptr->tv_sec;
usec = ptr->tv_usec;

printf("sec = %ld, usec = %ld\n", sec, usec);
exit(0);
}
```

これは、ptr という変数が timeval 構造体であるということが分かっているためにできることです。Ruby/DL でも PtrData オブジェクトに構造体の構成要素を与える struct! というメソッドが用意されています。このメソッドの第 1 引数には構造体を構成する型の組合せを与えます。第 2 引数以降にはそれぞれの要素にアクセスするためのキーをシンボルとして与えます。

```
ptr = DL.malloc(DL.sizeof('LL'))
ptr.struct!('LL', :tv_sec, :tv_usec)
ptr[:tv_sec] = 10
ptr[:tv_usec] = 100
sec = ptr[:tv_sec]
usec = ptr[:tv_usec]
```

また、PtrData オブジェクトを構造体として定義するためのメソッド struct! と同様に、共用体として定義するためのメソッド union! もあります。

2.6 コールバック関数の扱い方

C では仮引数に関数ポインタを渡すことができます。例えば前述した qsort() という関数は、ソートに必要な比較のための関数を引数にとります。先の例では、予め C で定義された関数を使いましたが、今回は Ruby 側でこのような関数を定義します。まず、C で書いた場合は次の通りの関数でした。

```
#include <string.h>
int mystrcmp(char **str1, char **str2)
{
    return strcmp(*str1, *str2);
}
```

```
} 
```

Ruby/DL ではコールバック関数を定義するには、DL.callback というモジュール関数を使います。

```
cb = DL.callback('IPP'){|ptr1, ptr2|
  str1 = ptr1.ptr.to_s
  str2 = ptr2.ptr.to_s
  str1 <=> str2
}
```

引数には関数のための型指定子を与えており、この場合、2つのポインタ型データを受け取り、整数を返す関数として定義しています。ただし Ruby/DL では、コールバック関数は有限個までしか定義できません。ptr1.ptr とすることで、C の *str1 に相当します。さらに得られたポインタ型データを文字列に変換するために to_s というメソッドを使っています。DL.callback は、戻り値として Symbol オブジェクトを返します。これを先の LIBC モジュールの関数 qsort に与えることができます。

```
include LIBC
qsort(["c","a","b"], cb1)
```

2.7 free 関数の定義

C では malloc() などによってメモリを確保すると free() を用いて解放するというのを普通に行いません。例えば、"abc" という文字列を PtrData 型に変換すると次の通りの結果となります。

```
ptr = "abc".to_ptr
# => #<DL::PtrData:0x8096f48
#   ptr=0x8096788
#   free=0x402e28c8>
```

ptr は、この PtrData 側が表すポインタの値です。free はこの PtrData オブジェクトが GC によって回収されるときに ptr に対して実行する関数へのポインタ値を表しており、この場合 free() を実行するようになっています。free() のための Symbol オブジェクトは予め Ruby/DL では DL::FREE として定義されています。

```
DL::FREE
# => #<DL::Symbol:0x8096658
```

```
#   func=0x402e28c8
#   'void free(void *);'>
```

func で示される値が、この関数へのポインタ値を表しています。

PtrData オブジェクトの free はユーザ自身の変更可能です。例えば、ptr という PtrData オブジェクトに sym という関数を free 関数として定義するには ptr.sym = sym とします。

3 Ruby/DL マニュアル

3.1 DL モジュール

3.2 DL::Handle クラス

3.3 DL::Symbol クラス

3.4 DL::PtrData クラス

3.5 型指定子

3.6 DL::MemorySpace モジュール

4 他の拡張ライブラリとの連携