

Improve extension API


C++ as better language for extension

Kouhei Sutou

ClearCode Inc.

RubyKaigi 2017

2017-09-19

A pyramid of black shocker combatmen figurines. The figurines are arranged in a pyramid shape, with the base having 6 rows of 6 figurines each, and the top row having 1 figurine. The figurines are black with white markings on their faces and chests. The text "Ad1: I'm distributing shocker combatmen to Rabbit users" is overlaid on the image.

Ad1: I'm distributing shocker combatmen to Rabbit users

宣伝1：Rabbitユーザーに
ショッカー戦闘員を配布中

Ad2: Silver sponsor

Silver Sponsors



ClearCode Inc.

<http://www.clear-code.com/>

Free software is important in ClearCode. We develop/support software with our free software development experiences. We feed back our business experiences to free software.

SOFTWARE

Ad3: Red Data Tools

- ✓ Project that provides data processing tools for Ruby

Ruby用のデータ処理ツール群を提供するプロジェクト

✓ <https://red-data-tools.github.io/>

- ✓ Workshop during the afternoon break on the 2nd day (today!)

2日目（今日！）の午後休憩中にワークショップがあるよ！

Ad4: OSS Gate

- ✓ Project that increases people who join OSS development

OSSの開発に継続的に参加する人を継続的に増やす取り組み

- ✓ <https://oss-gate.github.io/>

Ad4: OSS Gate

✓ Ruby is OSS

RubyもOSS

✓ OSS Gate wants to increase people to join Ruby itself and RubyGems development!

OSS GateではRuby本体の開発や各種RubyGemの開発に参加する人も増やしたい！

Ad4: OSS Gate

- ✓ Now, in Tokyo, Sapporo, Osaka and Kyoto

現在は東京・札幌・大阪・京都で活動中

- ✓ If you live near by, join "OSS Gate workshop"!

これらの地域に住んでいる人は「OSS Gateワークショップ」に参加しよう！

Ad4: OSS Gate

- ✓ Want to expand to Hiroshima and other areas all over the world!

広島や世界中のいろんな地域で活動したい！

- ✓ If you're interested in increasing people who join OSS development, talk to me!

OSSの開発に参加する人が増えることに興味のある人は私に声をかけて！

What I want to do

やりたいこと

Improve performance
with C/C++ libraries

C/C++のライブラリーを使った高速化

✓ Not create binding
バインディングを作りたいわけじゃない

Point of improving perf

高速化するために大事なこと

Done in C/C++
as much as possible
できるだけC/C++内で完結させる

- ✓ Don't move between
C/C++ and Ruby
C/C++とRuby間でいったりきたりしない

Example: #sum

例: #sum

```
numbers = (1..100000).to_a  
# Move between C and Ruby: 25.1ms  
numbers.inject(&:+)  
# Done in C: 0.5ms  
# 50x faster (50倍速い)  
numbers.sum
```

FYI: #inject(symbol)

参考情報: #inject(symbol)

```
numbers = (1..100000).to_a  
# Move between C and Ruby: 25.1ms  
numbers.inject(&:+)  
# Done in C: 0.5ms  
# 50x faster (50倍速い)  
numbers.inject(:+)
```

Extension and binding

拡張ライブラリーとバインディング

✓ Extension (拡張ライブラリー)

- ✓ Ruby library implemented in C
Cで実装されたRubyのライブラリー

✓ Binding (バインディング)

- ✓ Ruby library to use library
implemented in other languages
他言語実装のライブラリを使うためのRubyのライブラリ

Binding usage

バインディングの使い方

- ✓ Export each API to Ruby
それぞれのAPIをRubyで使えるようにする
- ✓ Combine exported APIs in Ruby
バインディングが提供するAPIをRubyレベルで組み合わせる

Binding usage example

バインディングの使い方例

```
require "cairo"  
include Cairo  
# Combine APIs  
s = PDFSurface.new("red.pdf", 10, 10) # API  
context = Context.new(s) # API  
context.set_source_color(:red) # API  
context.paint # API  
context.show_page # API  
s.finish # API
```

Point of improving perf

高速化するために大事なこと

Done in C/C++
as much as possible
できるだけC/C++内で完結させる

- ✓ Don't move between
C/C++ and Ruby
C/C++とRuby間でいったりきたりしない

Perf improvement example

高速化例

```
# Don't combine APIs in Ruby
# RubyレベルでAPIを組み合わせてない
## context.set_source_color(:red) # API
## context.paint                  # API
## context.show_page              # API
# Just call higher level API in Ruby
# Rubyからはもっと高レベルなAPIを呼び出す
context.show_red_page # Implemented in C
                      # ここはCで実装
```

What I want to do

やりたいこと

Improve performance
with C/C++ libraries

C/C++のライブラリーを使った高速化

✓ Not create binding
バインディングを作りたいわけじゃない

Use case

高速化したい場面

- ✓ Machine learning
機械学習
- ✓ Combine array/matrix operations
配列・行列に対する演算をまとめる

Raw C API for extension

拡張ライブラリー用の生のC API

Not bad but can be verbose
because of C

悪くないんだけどCなので冗長

✓ Need better approach

もっといい感じの方法を使いたい

Requirements

要件

✓ Easy to use C/C++ libraries

C/C++のライブラリーを簡単に使えること

✓ Easy to write as much as possible

できるだけ書きやすいこと

✓ Easy to debug as much as possible

できるだけデバッグしやすいこと

Approaches (実現方法)

- ✓ Extend language to support writing extension

拡張ライブラリーを書けるように言語を拡張

- ✓ Not based on C

C以外の言語を使う

- ✓ Provide convenient API

便利APIを提供

Recommended approach

オススメの実現方法

- ✓ Extend language to support writing extension

拡張ライブラリーを書けるように言語を拡張

- ✓ Not based on C

C以外の言語を使う

- ✓ **Provide convenient API**

便利APIを提供

Provide convenient API

便利なAPIを提供

- ✓ Rice: C++ + Ruby
- ✓ Ext++: C++11 + Ruby
- ✓ Boost.Python: C++ + Python
- ✓ pybind11: C++11 + Python

Useful C++ properties

C++の便利の性質

- ✓ C++ can use C API directory

C++ではCのAPIを直接使える

- ✓ No wrapper API or libffi

ラッパーAPIもlibffiもない

- ✓ C++**11 or later** has many convenient features

C++11以降には便利機能がたくさんある

C++ convenient feature1

C++の便利機能1

Type detection with "auto"

autoで型推論 (C++11)

```
int n = 10;  
auto square = n * n;  
// square's type is "int"  
// squareの型は「int」
```

C++ convenient feature2

C++の便利機能2

Lambda expression (C++11)

ラムダ式 (C++11)

```
// In Ruby: ->(n) {n * n}  
auto square = [](int n) {  
    return n * n; // Return type is detected  
};  
square(10); // => 100
```

C++ convenient feature3

C++の便利機能3

Default argument

デフォルト引数

```
// In Ruby: def square(n=10)  
int square(int n=10) {  
    return n * n;  
}  
// square() == square(10)
```

Convenient API example1

便利なAPI例1

```
# Ruby: Normal  
class Sample  
  def hello  
    "Hello"  
  end  
end
```

Convenient API example1

便利なAPI例1

```
# Ruby: No syntax sugar  
Sample = Class.new do  
  define_method(:hello) do  
    "Hello"  
  end  
end
```

Convenient API example1

便利なAPI例1

```
// C++: Ext++
#include <ruby.hpp>
extern "C" void Init_sample(void) {
    // Parent class (rb_cObject) is omissible
    rb::Class("Sample"). // class Sample in Ruby
        define_method("hello", // def hello in Ruby
            [](VALUE self) { // ->() {"Hello"} in Ruby
                return rb_str_new_static("Hello");
            });
}
```

Convenient API example1

便利なAPI例1

```
/* C */
#include <ruby.h>
static VALUE rb_sample_hello(VALUE self) {
    return rb_str_new_static("Hello");
} /* ↑Definition. */
void Init_sample(void) {
    /* ↓Must specify parent class. */
    VALUE sample = rb_define_class("Sample", rb_cObject);
    /* ↓Registration. They are separated. */
    rb_define_method(sample, "hello", rb_sample_hello, 0);
}
```


C++ convenient feature4

C++の便利機能4

Custom type conversion

型変換のカスタマイズ

- ✓ "Cast" is customizable
「キャスト」をカスタマイズできるということ

Custom type conversion

型変換のカスタマイズ

```
// Wrapper class of VALUE
class Object {
public:
    // def initialize(rb_object)
    //   @rb_object_ = rb_object
    // end
    Object(VALUE rb_object) : rb_object_(rb_object) {}
private:
    VALUE rb_object_;
};
```

Custom type conversion

型変換のカスタマイズ

```
class Object {  
    operator bool() const {  
        return RTEST(rb_object_);  
    }  
};  
  
// Object nil(Qnil); // Qnil wrapper  
// if (nil) { → if (RTEST(Qnil)) {
```

Custom type conversion

型変換のカスタマイズ

```
// Trap1 (罠1)  
// bool→int cast is available  
// Implicit Object→bool→ int cast  
// bool→intのキャストができるので  
// 暗黙的にObject→bool→intとキャスト  
Object(Qture) + 1; // → RTEST(Qtrue) + 1  
                // → 1 + 1  
                // → 2
```

Custom type conversion

型変換のカスタマイズ

```
class Object {  
    // Deny explicit cast (C++11)  
    // 暗黙のキャストを禁止 (C++11)  
    explicit operator bool() const {  
        return RTEST(rb_object_);  
    }  
};  
// Object(Qtrue) + 1; // → Compile error
```

Custom type conversion

型変換のカスタマイズ

```
// Trap2 (罠2)
class Object {
public:
    // Used as implicit VALUE→Object cast
    // 暗黙のVALUE→Objectキャストに使われる
    Object(VALUE rb_object);
};
```

Custom type conversion

型変換のカスタマイズ

```
// Trap2 (罠2)  
// VALUE is just a number (VALUEはただの数値)  
typedef unsigned long VALUE;  
Object identify(Object x) {return x;}  
// Implicit VALUE→Object cast  
// 暗黙的にVALUE→Objectとキャスト  
identify(100); // == identify(Object(100));
```

Custom type conversion

型変換のカスタマイズ

```
class Object {  
    // Deny explicit cast  
    // 暗黙のキャストを禁止  
    explicit Object(VALUE rb_object);  
};  
// identify(100); // → Compile error
```


Custom type conversion

型変換のカスタマイズ

```
class Object {  
    operator VALUE() const {return rb_object_;}  
};  
  
// Convenient (便利)  
rb_p(Object(Qnil)); // → rb_p(Qnil);  
// Not compile error. Hmm...  
Object(Qnil) + 1;    // → Qnil + 1;  
                    // → 8 + 1;  
                    // → 9;
```

C++ convenient feature5

C++の便利機能5

Template and specialization

テンプレートと特殊化

✓👍 Consistent cast API

一貫性のあるキャストAPIに使える

Consistent cast API

一貫性のあるキャストAPI

```
namespace rb {  
    // Consistent cast API  
    // Example: rb::cast<int32_t>(Object(NUM2INT(10)));  
    // Example: rb::cast<Object>(10);  
    // FYI: C++ cast syntax: static_cast<TYPE>(expression)  
    template <typename RETURN_TYPE,  
              typename ARGUMENT_TYPE>  
    inline RETURN_TYPE cast(ARGUMENT_TYPE object);  
}
```

Consistent cast API

一貫性のあるキャストAPI

```
template <> // rb::cast<int32_t>(Object(NUM2INT(10)));  
inline int32_t cast<int32_t, Object>(Object rb_object) {  
    return NUM2INT(rb_object);  
}  
template <> // rb::cast<Object>(10);  
inline Object cast<Object, int32_t>(int32_t n) {  
    return Object(INT2NUM(n));  
}
```

Consistent cast API

一貫性のあるキャストAPI

```
// rb::cast<const char *>(Object(rb_str_new_cstr("X")));  
template <> inline  
const char *cast<const char *, Object>(Object rb_object) {  
    VALUE rb_object_raw = rb_object;  
    return StringValueCStr(rb_object_raw);  
}  
template <> inline // rb::cast<Object>("hello");  
Object cast<Object, const char *>(const char *c_string) {  
    return Object(rb_str_new_cstr(c_string));  
}
```

C++ convenient feature6

C++の便利機能6

Initializer list (C++11)

初期化リスト (C++11)

✓ "{A,B,...}" is customizable

「{A, B, ...}」をカスタマイズできる

Initializer list

初期化リスト

```
Object Object::send(  
    ID name_id,  
    std::initializer_list<VALUE> args);  
// Ruby: "hello".send(:tr, "e", "E")  
Object(rb_str_new_cstr("hello")).  
    send(rb_intern("tr"),  
        {rb_str_new_cstr("e"),  
         rb_str_new_cstr("E")});
```

Initializer list

初期化リスト

```
Object Object::send(  
  ID name_id,  
  std::initializer_list<VALUE> args,  
  VALUE (*block)(VALUE data));  
// Ruby: array.send(:collect) {true}  
Object(array).  
  send(rb_intern("collect"),  
        {}, // Clear API than variable arguments  
        [](VALUE data) {return Qtrue;});
```


Convenient API example2

便利なAPI例2

```
// C++: Ext++  
rb::Object(rb_n).  
  send("step",  
        // Implicit Object→VALUE cast  
        {rb::cast<rb::Object>(10)},  
        [](VALUE i) {return rb_p(i)});  
// n.step(10) {|i| p i}
```

Convenient API example3

便利なAPI例3

```
// C++: Rice
#include <rice/Class.hpp>
static const char * // Not VALUE! (Auto convert)
rb_sample_hello(Rice::Object self) {
    return "Hello";
}

extern "C" void Init_sample() {
    Rice::define_class("Sample").
        define_method("hello", &rb_sample_hello);
}
```

C++ based API: Pros1

C++ベースのAPI : 長所1

Easier to write than C

Cより書きやすい

✓ Require C++11 or later

C++11以降なら

C++ based API: Pros2

C++ベースのAPI : 長所2

Easy to use for C API users

既存のC APIを使っている人なら使いやすい

✓ Use C directly incl macro

マクロも含めてCの機能を直接使える

✓ Don't need to migrate to all convenient API at once

一気に書き換えるのではなく徐々に移行できる

C++ based API: Pros3

C++ベースのAPI : 長所3

Easy to debug for C API users

既存のC APIを使っている人ならデバッグしやすい

✓ Can use GDB/LLDB directly

GDB/LLDBを直接使える

✓ GDB/LLDB have built-in C++ support

GDB/LLDBは組み込みでC++をサポート

C++ based API: Pros4

C++ベースのAPI : 長所4

Easy to optimize

最適化しやすい

✓ [Feature #13434] better
method definition in C API

Cのメソッド定義APIの改良

Better method definition

メソッド定義の改良

- ✓ Metadata for optimization
最適化のためにメタデータを付与
 - ✓ e.g.: Reduce memory allocations
例：メモリアロケーションを減らす
- ✓ Lazy method definition
必要になるまでメソッド定義を遅らせる
 - ✓ e.g.: Reduce start-up time
例：起動時間の高速化

Argument metadata

引数のメタデータ

```
class Hello
  # Default argument is just for
  # example. Other metadata will
  # be more useful for optimization.
  def hello(name, message="world")
  end
end
```


Argument metadata

引数のメタデータ

```
// C++: Rice
cHello.
  define_method(
    "hello",
    &hello,
    (Rice::Arg("name"), // ↓ Default
     Rice::Arg("message")="world"));
```

Lazy method definition

遅延メソッド定義

```
class X
  def a; end # Not define yet
  def b; end # Not define yet
end
x = X.new
x.a # Define #a and #b
```

Lazy method definition

遅延メソッド定義

```
/* One of new C API ideas */  
struct rb_method_entries entries[] = {  
    "a", ...,  
    "b", ...,  
};  
/* The definitions aren't defined at once. */  
/* They are defined when the next method call. */  
rb_define_method_with_table(rb_cX, entries);
```

Lazy method definition

遅延メソッド定義

```
// C++ implementation sample
{
  rb::Class("X").
    // Don't call rb_define_method() yet.
    define_method("a", ...).
    // Don't call rb_define_method() yet.
    define_method("b", ...);
    // Destructor is called.
    // Call rb_define_method_with_table()
    // in destructor.
}
```

Lazy method definition

遅延メソッド定義

```
// Ext++ implementation is just for test  
rb::Class("X").  
  // Call rb_define_method() immediately.  
  define_method("a", ...).  
  // Don't call rb_define_method() in  
  // the following define_method(s).  
  enable_lazy_define_method().  
  // Don't call rb_define_method() yet.  
  define_method("b", ...);
```

Lazy method definition

遅延メソッド定義

```
# Define only benchmark code in Ruby.  
# Benchmark target code is in C++.  
n = 10000  
Bench = Class.new do  
  n.times do |i|  
    define_method("method#{i}") do  
      end  
    end  
  end  
end
```

Lazy method definition

遅延メソッド定義

```
# Call benchmark code in Ruby.  
# Benchmark target code is in C++.  
n = 10000  
bench = Bench.new  
n.times do |i|  
  bench.__send__("method#{i}")  
end
```

Lazy method definition

遅延メソッド定義

Type	Define only	Called
Normal	5ms	5ms
Lazy	1ms	5ms

5x faster when any methods
aren't called

メソッドが呼ばれなければ5倍速い

C++ based API: Cons1

C++ベースのAPI : 短所1

C++ is difficult

C++は難しい

- ✓ e.g.: Template
たとえばテンプレート
- ✓ Easy to write unreadable code
簡単にリーダブルじゃないコードを書ける

C++ based API: Cons2

C++ベースのAPI : 短所2

Slower build

ビルドが遅い

✓ It may reduce try&error cycle
試行錯誤しにくくなるかも

C++ based API: Problem

C++ベースのAPI : 課題

Exception

例外

- ✓ Ruby exception breaks C++
RAII (destructor)

(Resource Acquisition Is Initialization)

Rubyの例外発生→C++のRAII（デストラクター）が動かない

- ✓ Because it uses setjmp/longjmp
Rubyの例外はsetjmp/longjmpを使っているから

Exception: Solution

例外：解決法

1. Rescue Ruby exception

Rubyの例外をrescue

2. Throw C++ exception

C++の例外にしてthrow

3. Re-raise the Ruby exception

安全な場所でRubyの例外を再raise

Conclusion

まとめ

✓ C++ based API is useful

C++ベースのAPIは便利

✓ For writing ext uses C/C++ library

C/C++のライブラリを使う拡張ライブラリを書くとき

✓ For optimizing w/ easy to use API

例：使いやすいAPIを維持したまま最適化するとき

Appendix

付録

Introduce easy to write
extension approaches

拡張ライブラリーを簡単に実装する方法を紹介

- ✓ The following contents are
used only when time is
remained

以降の内容は時間が残っている場合だけ使う

Approaches (実現方法)

- ✓ Extend language to support writing extension

拡張ライブラリーを書けるように言語を拡張

- ✓ Not based on C

C以外の言語を使う

- ✓ Provide convenient API

便利APIを提供

Extend language

言語を拡張

✓ Rubex: Extended Ruby

Rubex : Rubyを拡張

✓ Cython: Extended Python

Cython : Pythonを拡張

How to run

動かし方

- ✓ Translate extension code to C
拡張言語で書かれたコードをCにコンパイル
- ✓ Compile C code
コンパイルされたCコードをビルド
- ✓ Load the built extension
ビルドした拡張ライブラリーを読み込む

Extended syntax

拡張された構文

- ✓ Type information

型情報を書ける

- ✓ C code snippet

Cのコードを書ける

How to run: Rubex

Rubexの動かし方

```
# fibonacci.rubex  
class Fibonacci  
  # "int" is type information  
  def compute(int n)  
    # ...  
  end  
end
```

How to run: Rubex

Rubexの動かし方

```
% rubex fibonacci.rubex  
% cd fibonacci  
% ruby extconf.rb  
% make
```

How to run: Rubex

Rubexの動かし方

```
require_relative "fibonacci.so"  
p Fibonacci.new.compute(100)
```

Extend language: Pros1

拡張言語：長所1

Friendly syntax for
base language users

ベースの言語のユーザーにはなじみやすい構文

✓ Most syntax is the same

構文の大部分は同じだから

Extend language: Pros2

拡張言語：長所2

Easy to migrate from base lang

ベースの言語からの移行が容易

✓ Because upward compatibility

上位互換だから

✓ Code for base language works
without modification

ベースの言語で書かれたコードは変更なしで動く

Extend language: Pros3

拡張言語：長所3

Doesn't require much C knowledge

そんなにCの知識は必要ない

✓ Most code can be written with
base language knowledge

コードの大部分はベースの言語の知識で書ける

Extend Language: Cons1

拡張言語：短所1

You realize that it's not
friendly syntax when you use it

使うとそんなになじみやすい文法ではないと気づく

✓ Small differences will
confuse you

小さな違いがいろいろあってわかりにくい

Extend language: Cons2

拡張言語：短所2

Difficult to debug
デバッグが難しい

- ✓ Need base language, extend language and C knowledge
ベースの言語の知識も拡張言語の知識もCの知識も必要

Cython has GDB integration to solve this problem
Cythonはこの問題を解決するためにGDB用の便利機能を提供

Extend language: Cons3

拡張言語：短所2

Hard to maintain

(For maintainers, not for users)

(ユーザーではなくメンテナーが) メンテナンスが大変

- ✓ Base language introduces a new syntax then extend language should implement it
ベースの言語が新しい構文を導入→拡張言語でも実装

Not based on C

C言語以外をベースにする

- ✓ JRuby: Java + Ruby
- ✓ Helix: Rust + Ruby

How to run: JRuby

JRubyでの動かし方

```
// Fibonacci.java  
public class Fibonacci {  
    public long[] compute(int n) {  
        // ...  
    }  
}
```

How to run: JRuby

JRubyでの動かし方

```
% javac Fibonacci.java  
% jar cf fibonacci.jar Fibonacci.class
```

How to run: JRuby

JRubyでの動かし方

```
require "fibonacci.jar"  
java_import "Fibonacci"  
p Fibonacci.new.compute(100)
```

Not based on C: Pros1

C言語以外をベースにする：長所1

Easier to write than C

Cより書きやすい

- ✓ **Simpler syntax**
洗練された構文
- ✓ **Rich features compared to C**
Cより機能が多い

Not based on C: Pros2

C言語以外をベースにする：長所2

Can use libraries
in base language

ベース言語のライブラリーを使える

✓ Major languages have many libs

広く使われている言語はライブラリーも多い

Not based on C: Cons1

C言語以外をベースにする：短所1

Need base language knowledge

ベース言語の知識が必要

✓ Java for JRuby (JRubyならJava)

✓ Rust for Helix (HelixならRust)

Not based on C: Cons2

C言語以外をベースにする：短所2

May need C knowledge

(When Ruby implementation is MRI)

Rubyの実装がMRIならCの知識が必要かもしれない

✓ Base language wraps Ruby C API

ベースの言語はRubyのC APIをラップしている

✓ e.g.: `sys::RSTRING_PTR` on Helix

例：Helixなら`sys::RSTRING_PTR`がラップしたAPI

Not based on C: Cons3

C言語以外をベースにする：短所3

Hard to maintain

(When Ruby implementation is MRI)

Rubyの実装がMRIならメンテナンスが大変かも

- ✓ Ruby introduces a new API
then base language may need
to implement it

Rubyが新しいAPIを追加→ベース言語でも実装？

- ✓ e.g.: `rb_gc_adjust_memory_usage()`

Provide convenient API

便利なAPIを提供

- ✓ Rice: C++ + Ruby
- ✓ Ext++: C++11 + Ruby
- ✓ Boost.Python: C++ + Python
- ✓ pybind11: C++11 + Python

How to run: Rice

Riceの動かし方

```
#include <rice/Class.hpp>

static const char * // Not VALUE!
rb_sample_hello(Rice::Object self) {
    return "Hello";
}

extern "C" void Init_sample() {
    Rice::define_class("Sample").
        define_method("hello", &rb_sample_hello);
}
```

How to run: Rice

Riceの動かし方

```
# extconf.rb  
require "mkmf-rice"  
create_makefile("sample")
```

How to run: Rice

Riceの動かし方

```
% ruby extconf.rb  
% make
```


How to run: Rice

Riceの動かし方

```
require_relative "sample.so"  
p Sample.new.hello  
# => "Hello"
```

Provide C++ API: Pros

C++ APIを提供：長所

Omit

省略

Provide C++ API: Cons

C++ APIを提供：短所

Omit

省略

Provide C++ API: ConsN

C++ APIを提供：短所N

Conv from Ruby may be a bother

Ruby実装の移植が面倒

- ✓ C++ with convenient Ruby C API needs more code than Ruby

RubyのC APIにC++の便利APIがあっても
Rubyよりもたくさんコードが必要

From Ruby: Rice

RubyからRiceに移植

```
def fib(n)
  prev = 1
  current = 1
  1.step(n - 1).collect do
    prev, current = current, current + prev
    prev
  end
end
```

From Ruby: Rice

RubyからRiceに移植

```
std::vector<uint64_t> fib(Rice::Object self, int n) {  
    uint64_t prev = 1, current = 1;  
    std::vector<uint64_t> numbers;  
    for (int i = 1; i < n; ++i) {  
        auto temp = current; current += prev; prev = temp;  
        numbers.push_back(prev);  
    }  
    return numbers;  
}
```