

Geração de Código em C para Krakatoa

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
e-mail: jose@dc.ufscar.br

21 de outubro de 2015

Este artigo descreve a tradução dos programas de Krakatoa para C. A tradução não é direta pois as linguagens estão em paradigmas diferentes: Krakatoa é orientada a objetos e C é procedimental. Mas Krakatoa é uma linguagem de mais alto nível do que C, o que facilita a tradução. Entre as construções de alto nível de Krakatoa, temos classes, herança e envio de mensagens. São estas construções que tornam a geração de código difícil, em particular o envio de mensagem. Começaremos então mostrando como se gera código utilizando um pequeno exemplo, dado abaixo, que não possui herança ou envio de mensagem.

```
class Program {  
  
    public void run() {  
        int i, b;  
        boolean primo;  
        String msg;  
        write( "Olá, este é o meu primeiro programa" );  
        write( "Digite um número: " );  
        read(b);  
        // um meio super ineficiente de verificar se um número é primo  
        primo = true;  
        i = 2;  
        while ( i < b ) {  
            if ( b%i == 0 ) {  
                primo = false;  
                break;  
            }  
            else  
                i++;  
        }  
        if ( primo )  
            msg = "Este numero e primo";  
        else  
            msg = "Este numero nao e primo";  
        write(msg);  
    }  
}
```

Abaixo está o código em C padrão que deve ser gerado para o exemplo acima. Colocamos em comentários explicações e/ou partes do código em Krakatoa.

```
/* deve-se incluir alguns headers porque algumas funções da biblioteca
   padrão de C são utilizadas na tradução. */

#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>

/* define o tipo boolean */
typedef int boolean;
#define true 1
#define false 0

/* define um tipo Func que é um ponteiro para função */
typedef
    void (*Func)();

/* Para cada classe, deve ser gerada uma estrutura como a abaixo. Se Program
   tivesse variáveis de instância, estas seriam declaradas nesta estrutura.
   _class_Program representa em C uma classe em Krakatoa. */

typedef
    struct _St_Program {
        /* ponteiro para um vetor de métodos da classe */
        Func *vt;
    } _class_Program;

/* Este é um protótipo de método que cria um objeto da classe Program.
   Toda classe A não abstrata possui um método new_A que cria e retorna
   um objeto da classe A. O método new_Program é declarado antes do
   método main, abaixo.
   */
_class_Program *new_Program(void);

/*
   Este é o método run da classe Program. Note que o método é traduzido
   para uma função de C cujo nome é uma concatenação do nome da classe
   com o nome do método. Sempre há um primeiro parâmetro chamado this
   cujo tipo é a estrutura que representa a classe, neste caso,
   _class_Program.
   */
void _Program_run( _class_Program *this )
{
    // os nomes de variáveis locais são precedidos por _
    int _i;
    int _b;
```

```

boolean _primo;
    // Strings são mapeadas para char * em C
char *_msg;

    // write com Strings são mapeados para puts em C
puts( "Ola, este e o meu primeiro programa" );
puts( "Digite um numero: ");
    // read(b), com b inteiro é mapeado para o código entre { e } abaixo
{
    char __s[512];
    gets(__s);
    sscanf(__s, "%d", &_b);
}
    // o restante do código é praticamente igual em Krakatoa e C, a menos
    // de nomes de identificadores
_primo = true;
_i = 2;

while ( _i < _b )
    if ( _b%_i == 0 ) {
        _primo = false;
        break;
    }
    else
        _i++;
if ( _primo != false )
    _msg = "Este numero e primo";
else
    _msg = "Este numero nao e primo";
puts(_msg);
}

/*
    Para toda classe deve ser declarado um vetor de Func (vetor de
    ponteiro para funções). O nome deve ser VTclass_NomeDaClasse, como
    VTclass_Program. Este vetor é inicializado (iniciado) com as funções
    em C, como _Program_run, que representam os métodos **públicos**
    da classe. Note que o tipo de _Program_run é
        void (*)(_class_program *)
    e portanto é necessário um cast para convertê-lo para o tipo de Func,
    void (*)()
*/
Func VTclass_Program[] = {
    ( void (*)() ) _Program_run
};

/*

```

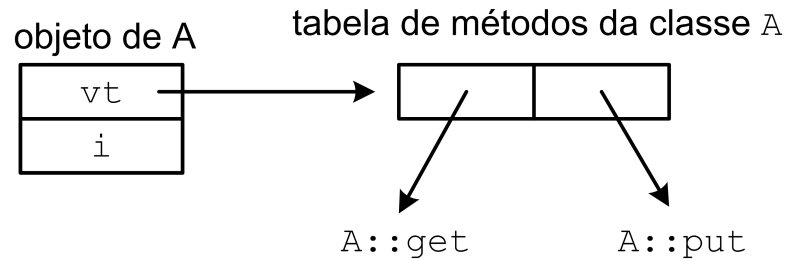


Figura 1: Representação de um objeto da classe A

Para toda classe não abstrata se declara uma função `new_NomeDaClasse` que aloca memória para um objeto da classe, que é um "objeto" da estrutura `_class_NomeDaClasse`. Note que este método é igual para todas as classes, a menos do nome da classe.

*/

```
_class_Program *new_Program()
{
    _class_Program *t;

    if ( (t = malloc(sizeof(_class_Program))) != NULL )
        // o texto explica porque vt é inicializado
        t->vt = VTclass_Program;
    return t;
}

// genC de Program da ASA deve gerar a função main exatamente como abaixo.
int main() {
    _class_Program *program;

    /* crie objeto da classe Program e envie a mensagem run para ele.
       Nem sempre o número de run no vetor é 0. */
    program = new_Program();
    ( ( void (*) (_class_Program *) ) program->vt[0] )(program);
    return 0;
}
```

Os nomes de identificadores do programa em Krakatoa, como classes, variáveis e métodos, têm os seus nomes alterados na tradução. Sempre se coloca sublinhado () no início de cada identificador no código em C.

Se o tipo da variável `pa` no programa em Krakatoa for `A`, o tipo em C será o de um ponteiro para `_class_A`, definida abaixo. Todas as variáveis cujos tipos são classes serão traduzidos para ponteiros em C. Isto será assumido no texto que se segue.

Um objeto de uma classe `A` possui todas as variáveis declaradas em `A` mais um ponteiro para um vetor de métodos. Como exemplo, a Figura 1 mostra¹ um objeto da seguinte classe `A`:

```
class A {
```

¹Usaremos `C::m` para designar método `m` da classe `C`.

```

private int i;
public int get() {
    return this.i;
}
public void put( int p_i ) {
    this.i = p_i;
}
}

```

Todos os objetos possuem um ponteiro, que chamaremos de *vt*, que aponta para a tabela de métodos *públicos* da classe.

Cada classe possui um vetor de ponteiros onde cada entrada aponta para um dos métodos *públicos* da classe. Todos os objetos de uma classe apontam, via ponteiro *vt*, para a mesma tabela de métodos (TM) da classe.

Assim, se *pa* referir-se a um objeto da classe *A*, *_pa->vt[0]* (já traduzindo *pa* para um ponteiro em C) apontará para um método público de *A* (neste caso, *A::get()*).

O compilador, ao compilar a classe *A*, transforma-a em uma estrutura contendo *vt* na primeira posição e as variáveis de instância de *A* em seguida:

```

typedef
struct _St_A {
    Func *vt;
    int _A_i;
} _class_A;

```

O tipo *Func* é definido como

```

typedef
void (*Func)();

```

Isto é, um ponteiro para uma função.

Cada método de *A*, seja ele público ou privado, é convertido em uma função que toma como parâmetro um ponteiro para *_class_A* e cujo nome é formado pela concatenação do nome da classe e do método:

```

int _A_get( _class_A *this ) {
    return this->_A_i;
}

void _A_put( _class_A *this, int _p_i ) {
    this->_A_i = _p_i;
}

```

O nome do primeiro parâmetro é sempre *this* e é através dele que são manipuladas as variáveis de instância da classe, que estão declaradas em *_class_A*. A codificação dos métodos privados é exatamente igual à dos métodos públicos.

Agora, a tabela de métodos públicos da classe *A* é declarada e inicializada com as funções acima:

```

Func VTclass_A[] = {
    _A_get,
    _A_put
};

```

De fato, a declaração acima possui erros de tipo, pois o tipo de `_A_get` (ou `_A_put`) é diferente do tipo de `Func`, mas não nos preocuparemos com isto por enquanto. `Func` é o tipo de cada um dos elementos do vetor `VTclass_A`.

Como dissemos anteriormente, cada objeto de `A` aponta para um vetor de métodos públicos de `A`, que é `VTclass_A`. Assim, quando um objeto de `A` é criado, deve-se fazer o seu campo `vt` apontar para `VTclass_A`.

O compilador transforma

```
pa = new A(); /* Krakatoa */
```

em

```
_pa = new_A(); /* C */
```

onde `new_A` é definida como

```
_class_A *new_A()
{
    _class_A *t;

    if ( (t = malloc(sizeof(_class_A))) != NULL )
        t->vt = VTclass_A;
    return t;
}
```

Observe que a ordem dos métodos em `VTclass_A` é a mesma da declaração da classe `A`. A posição 0 é associada a `get` e 1, a `put`.

Uma chamada de um método público

```
j = pa.get();
```

é transformada em uma chamada de função através de `_pa->vt`:

```
_j = (_pa->vt[0])(_pa);
```

O índice 0 foi usado porque 0 é o índice de `get` em `VTclass_A`. O primeiro parâmetro de uma chamada de métodos é sempre o objeto que recebe a mensagem. Neste caso, `pa`.

De fato, a instrução acima possui um erro de tipos: `_pa->vt[0]` não admite nenhum parâmetro e estamos lhe passando um, `pa`. Isto é corrigido colocando-se uma conversão de tipos:

```
_j = ( (int (*)(_class_A *)) _pa->vt[0] )(_pa);
```

O tipo “`int (*)(_class_A *)`” representa um ponteiro para função que toma um “`_class_A *`” como parâmetro e retorna um `int`.

Como mais um exemplo,

```
pa.put(12)
```

é transformado em

```
(_pa->vt[1])(_pa, 12)
```

ou melhor, em

```
((void (*)(_class_A *, int)) _pa->vt[1])(_pa, 12)
```

Com as informações acima, já estamos em condições de traduzir um programa de Krakatoa para C. Apresentamos abaixo um programa em Krakatoa seguido da sua tradução para C.

```
class A {
    private int i;
    public int get() {
        return this.i;
    }
    public void put( int p_i ) {
```

```

        this.i = p_i;
    }
}

class Program {
    public void run() {
        A a;
        int k;

        a = new A();
        a.put(5);
        k = a.get();
        write(k);
    }
}

```

Tradução do programa Krakatoa acima para C:

```

#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>

typedef int boolean;
#define true 1
#define false 0

typedef
    void (*Func)();

// class A { ... }
typedef
    struct _St_A {
        Func *vt;
        // variável de instância i da classe A
        int _A_i;
    } _class_A;

_class_A *new_A(void);

int _A_get( _class_A *this ) {
    return this->_A_i;
}

void _A_put( _class_A *this, int _p_i ) {
    this->_A_i = _p_i;
}

// tabela de métodos da classe A -- virtual table

```

```

Func VTclass_A[] = {
    ( void (*)() ) _A_get,
    ( void (*)() ) _A_put
};

class_A *new_A()
{
    _class_A *t;

    if ( (t = malloc(sizeof(_class_A))) != NULL )
        t->vt = VTclass_A;
    return t;
}

typedef
    struct _St_Program {
        Func *vt;
    } _class_Program;

_class_Program *new_Program(void);

void _Program_run( _class_Program *this )
{
    // A a;
    _class_A *_a;
    // int k;
    int _k;

    // a = new A();
    _a = new_A();
    // a.put(5);
    ( (void *)(_class_A *, int)) _a->vt[1] )(_a, 5);
    // k = a.get();
    k = ( (int *)(_class_A *)) _a->vt[0] )(_a);
    // write(k);
    printf("%d ", _k );
}

Func VTclass_Program[] = {
    ( void (*)() ) _Program_run
};

_class_Program *new_Program()
{
    _class_Program *t;

    if ( (t = malloc(sizeof(_class_Program))) != NULL )

```



```

    t->vt = VTclass_Program;
    return t;
}

int main() {
    _class_Program *program;

    /* crie objeto da classe Program e envie a mensagem run para ele */
    program = new_Program();
    ( ( void *)(_class_Program *) ) program->vt[0] )(program);
    return 0;
}

```

Neste programa estão colocadas as conversões de tipo (*casts*) necessárias para que o programa compile. Como definido pela linguagem, a execução do programa começa com a criação de um objeto da classe Program, que é seguida do envio da mensagem run para este objeto.

```

class B extends A {
    private int lastInc;
    private void add( int n ) {
        this.lastInc = n;
        super.put( super.get() + n );
    }
    public void print() {
        write( this.get() );
    }
    public void put( int p_i ) {
        if ( p_i > 0 )
            super.put(p_i);
    }
    public void inc() {
        this.add(1);
    }
    public int getLastInc() {
        return this.lastInc;
    }
    final public void atLast() {
    }
}

```

Considere agora a classe B do código acima. A tradução desta classe para C é mostrada abaixo.

```

typedef
struct _St_B {
    Func *vt;
    int _A_i;
    int _B_lastInc;
} _class_B;

```

```

_class_B *new_B(void);

void _B_add( _class_B *this, int _n )
{
    this->_B_lastInc = _n;
    _A_put( (_class_A *) this, _A_get( (_class_A *) this ) + _n );
}

void _B_print ( _class_B *this )
{
    printf("%d ", ((int (*)(_class_A *)) this->vt[0])( (_class_A *) this));
}

void _B_put( _class_B *this, int _p_i )
{
    if ( _p_i > 0 )
        _A_put((_class_A *) this, _p_i);
}

void _B_inc( _class_B *this )
{
    _B_add( this, 1);
}

int _B_getLastInc( _class_B *this )
{
    return this->_B_lastInc;
}

void _B_atLast( _class_B *this ) {
}

// apenas os métodos públicos
Func VTclass_B[] = {
    (void (*) ( ) ) _A_get,
    (void (*) ( ) ) _B_put,
    (void (*) ( ) ) _B_print,
    (void (*) ( ) ) _B_inc,
    (void (*) ( ) ) _B_getLastInc
};

_class_B *new_B()
{
    _class_B *t;

    if ((t = malloc (sizeof(_class_B))) != NULL)
        t->vt = VTclass_B;
    return t;
}

```

}

A classe B possui vários aspectos ainda não examinados:

1. o método `print` envia a mensagem `get` para `this`. O método `get` é público.
2. chamada a métodos privados usando `this`: `(this.add(1))`.
3. o método `put` de B chama o método `put` de A através de `super.put(p_i)`;²
4. o acréscimo de métodos nas subclasses (`print`, `inc` e `getLastInc`);
5. a redefinição de métodos nas subclasses (`put`);
6. adição de variáveis de instância nas subclasses (`lastInc`);
7. definição de métodos privados (`add`);
8. um método `atLast` final.

Veremos abaixo como gerar código para cada uma destas situações.

1. “`this.get()`” é traduzido para

```
(this->vt[0])(this)
```

ou melhor,

```
( (int (*)( _class_A *)) this->vt[0]) ( ( _class_A *) this )
```

Se o método é público, a ligação mensagem/método é dinâmica — é necessário utilizar a tabela de métodos mesmo o receptor da mensagem sendo `this`.

2. A chamada `this.add(1)` especifica qual método chamar: `add` da classe corrente. Sendo `add` um método privado, sabemos exatamente de qual método estamos falando. Então a chamada em C é estática:

```
_B_add(this, 1)
```

Não há necessidade de converter `this` pois o seu tipo é `_class_B` e `add` é declarado como

```
void _B_add( _class_B *this, int _n ) { ... }
```

Chamadas a métodos privados nunca precisarão de conversão de tipo para `this`. Recordando, envios de mensagem para `this` resultam em ligação dinâmica (usando `vt`) se o método for público ou em ligação estática se o método for privado.

3. A chamada `super.put(p_i)` especifica claramente qual método chamar: o método `put` de A.³

Portanto, esta instrução resulta em uma ligação estática, que é:

```
_A_put( this, _p_i )
```

ou

```
_A_put( ( _class_A *) this, _p_i )
```

com conversão de tipos.

Como o destino do envio de mensagem `put` não é especificado (como `a` em `a.put(5)`), assume-se que ele seja `this`. Observe que em

```
_A_put( ( _class_A *) this, _p_i )
```

é necessário converter um ponteiro para `_class_B`, que é `this`, em um ponteiro para `_class_A`.

²Observe que esta herança está errada. Métodos de subclasses não podem restringir valores de parâmetros.

³O compilador faz uma busca por método `put` começando na superclasse de B, A, onde é encontrado.

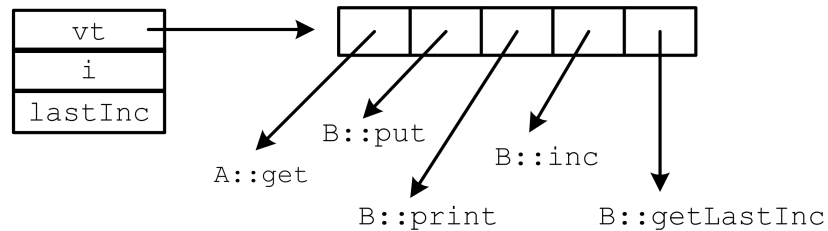


Figura 2: Objeto e tabela de métodos de B

Isto não causa problemas porque `_class_B` é um superconjunto de `_class_A`, como foi comentado acima. Note que o protótipo de `_A_put` é

```
_A_put( _class_A *, int )
```

- O acréscimo de métodos em subclasses faz com que a tabela de métodos aumente proporcionalmente. Assim, a tabela de métodos para B possui três entradas a mais do que a de A, para `print`, `inc` e `getLastInc`.
- A redefinição de `put` em B faz com que a tabela de métodos de B refira-se a `B::put` e não a `A::put`. A classe B herda o método `get` de A, redefine o `put` herdado e adiciona o método `print`. Assim, a tabela de métodos de B aponta para `A::get`, `B::put` e `B::print`, como mostrado na Figura 2.
- A declaração de `_class_B` é

```
typedef
struct _St_B {
    Func *vt;
    int _A_i;
    int _B_lastInc;
} _class_B;
```

As variáveis da superclasse (como `_A_i`) aparecem antes das variáveis da subclasse (`_B_lastInc`).

- A geração de código para um método privado é exatamente igual à de um método público. Porém, métodos privados não são colocados na tabela de métodos (veja `VTclass_B`).
- Um método `final` é colocado na tabela de métodos se ele é a redefinição de um método da superclasse. Se a superclasse não tiver um método com mesmo nome, o método final não é colocado nesta tabela. Este é o caso deste exemplo. Uma chamada “`b.atLast()`” resulta em uma chamada estática se `b` for um objeto do tipo B ou de subclasses de B. Se `atLast` fosse herdado da superclasse A, “`b.atLast()`” seria uma chamada dinâmica (com o uso da tabela de métodos) se `b` fosse do tipo A e estática se `b` fosse do tipo B ou subclasses.

Na tabela de métodos de B, a numeração de métodos de A é preservada. Assim, a posição 0 da TM de B aponta para o método `get` porque esta mesma posição da TM de A aponta para `get`. Isto acontece somente porque B herda A. As numerações em classes não relacionadas por herança não são relacionadas entre si.

A preservação da numeração em subclasses pode ser melhor compreendida se considerarmos um método polimórfico `f`:

```

class X {
    public void f( A a ) {
        a.put(5);
    }
}

```

Este método é transformado em

```

void _X_f( _class_X *this, _class_A *_a )
{
    ( (void (*)( _class_A *, int )) _a->vt[1] )(_a, 5);
}

```

É fácil ver que a chamada

```

X x = new X();
t = new A();
x.f(t);

```

causa a execução de `A::put`, já que `_t->vt` e `_a->vt`⁴ apontam para `VTclass_A` e a posição 1 de `VTclass_A` (que é `_a->vt[1]`) aponta para `A::put`.

Como B é subclasse de A, objetos de B podem ser passados a f:

```

t = new B();
x.f(t);

```

Agora, `_t->vt` e `_a->vt` apontam para `VTclass_B` e a posição 1 de `VTclass_B` aponta para `B::put`. Então, a execução de `f` causará a execução de `B::put`, que é apontado por `_a->vt[1]`.

`f` chama `put` de A ou B conforme o parâmetro seja objeto de A ou B. Isto só acontece porque `B::put` foi colocado em uma posição em `VTclass_B` igual à posição de `A::put` em `VTclass_A`.

Esclarecemos melhor este ponto através de um exemplo. Se `VTclass_B` fosse declarado como

```

Func VTclass_B [] = {
    _A_get,
    _B_print,
    _B_put,
    _B_inc,
    _B_getLastInc
};

```

a execução da função `f` chamaria `_B_print`. Isto estaria errado pois a instrução, “`a.put(5)`”, é uma chamada ao método `put`. A declaração correta é a dada anteriormente:

```

Func VTclass_B[] = {
    (void (*) ( ) ) _A_get,
    (void (*) ( ) ) _B_put,
    (void (*) ( ) ) _B_print,
    (void (*) ( ) ) _B_inc,
    (void (*) ( ) ) _B_getLastInc
};

```

⁴a é o parâmetro formal de f.

Com esta tabela de método a posição 1 é ocupada pelo método `B::put`, o método correto.

A codificação dos comandos `read` e `write` é feita da seguinte forma:

`read(b)` é transformado em

```
{ char __s[512];
  gets(__s);
  sscanf(__s, "%d", &_b);
}
```

se `b` for uma variável local do tipo `int`. Obviamente, se `b` for variável de instância de uma classe `A`, o comando `sscanf` ficaria

```
sscanf(__s, "%d", &this->_A_b);
```

Se o tipo de `b` for `String`, o código gerado deverá ser

```
{
  char __s[512];
  gets(__s);
  _b = malloc(strlen(__s) + 1);
  strcpy(_b, __s);
}
```

O comando `write(expr)` deverá gerar o código

```
printf("%d ", código para expr);
```

se o tipo de `expr` for `int`. Por exemplo, se `expr` for a variável local `b`, o código gerado seria

```
printf("%d ", _b);
```

Se o tipo de `expr` for `String`, o código gerado deve ser

```
puts(código para expr);
```

Métodos estáticos devem ser transformados em funções em C que não tomam `this` como primeiro parâmetro. Variáveis estáticas devem ser transformadas em variáveis globais. Um método estático como nome `m` de uma classe de `A`, seja ele público ou privado, é traduzido para uma função com nome `_static_A_m`. Uma variável estática com nome `x` de uma classe `A` é traduzida para uma variável global `_static_A_x`.

A chamada `A.m(a, b)` de um método estático é traduzida para `_static_A_m(a, b)` em C. Para exemplificar a codificação de métodos e variáveis estáticos, utilizaremos o seguinte exemplo:

```
class A {
  static private int n;
  static public int get() {
    return A.n;
  }
  static public void set( int n ) {
    A.n = n;
  }
}

class Program {
  public void run() {
```

```

        A.set(0);
        write(A.get());
    }
}

```

Este exemplo é traduzido para o seguinte código em C:

```

typedef
    struct _St_A {
        Func *vt;
    } _class_A;
_class_A *new_A(void);

int _static_A_n;
int _static_A_get() {
    return _static_A_n;
}
void _static_A_set(int n) {
    _static_A_n = n;
}

Func VTclass_A[] = {
};

class_A *new_A()
{
    ...
}
... // código para a classe Program

void _Program_run( _class_Program *this )
{
    _static_A_set(0);
    printf("%d ", _static_A_get() );
}

```

Outras observações sobre geração de código:

- considere que os tipos das variáveis **a** e **b** são **A** e **B**, respectivamente, sendo que a classe **B** herda de **A** (direta ou indiretamente). Então o código gerado para a atribuição “**a = b**” deve ser `_a = (_class_A *) _b;` Naturalmente, o mesmo se aplica se **a** for variável de instância e **b** uma expressão;
- não se gera código para método abstrato. Na tabela de métodos, deve-se usar **NULL** na posição correspondente ao método;
- **null** em Krakatoa deve ser traduzido para **NULL** em C;
- métodos **final** são gerados exatamente como outros métodos públicos;

- não é necessário colocar quaisquer comentários no código gerado em C;
- O código


```
if ( expr ) statement;
```

 em Krakatoa, onde o tipo de `expr` é `boolean`, deve ser traduzido para


```
if ( (expr) != false ) statement;
```

 em C. E


```
if ( ! expr ) statement;
```

 deve ser traduzido para


```
if ( (expr) == false ) statement;
```
- string literais em Krakatoa, como "Oi, tudo bem ?", deve ser traduzidos literalmente, "Oi, tudo bem ?" em C;
- coleta de lixo não deve ser implementada.

Desempenho

O código abaixo compara o tempo de execução de uma chamada de função (com corpo vazio e um ponteiro como parâmetro) com o tempo de execução de chamadas indiretas de função.⁵

```
// código em C++
class C {
public:
    virtual void f() { }
};

typedef
void (*Func)();

typedef
struct {
    Func *vt;
} _class_A;

void f( _class_A * ) { }

Func VTclass_A[] = { (void (*)( )) f };

int i, j;
_class_A a, *pa = &a;
C c; C *pc = &c;
void (*pf)(class_A *) = f;

void main()
{
```

⁵Estes dados foram obtidos em uma estação SPARC com Unix.


```

a.vt = VTclass_A;

for (i = 0; i < 1000; i++ )
  for (j = 0; j < 1000; j++ ) {
    // ; /* 0.21 */
    // f(pa); /* 0.37 - 0.21 = 0.16 */
    // pf(pa); /* 0.41 - 0.21 = 0.20 */
    // ( ( void (*) (_class_A *) ) pa->vt[0]) (pa); /* 0.46 - 0.21 = 0.25 */
    // pc->f(); /* 0.59 - 0.21 = 0.38 */
  }
}

```

A tabela de tempos é sumarizada em

f (pa)	1
(*pf) (pa)	1.25
pa->vt [0] (pa)	1.56
pb->f ()	2.4

Observe que o envio de mensagem `pb->f()` é implementado pela própria linguagem C++ e utiliza um mecanismo mais lento do que a implementação descrita nesta seção.

Um sumário da transformação Krakatoa \rightarrow C é:

- Cada objeto possui um campo `vt` que aponta para uma tabela (vetor) de ponteiros onde cada ponteiro aponta para um dos métodos da classe do objeto.
- Todos os objetos de uma mesma classe apontam para a mesma tabela de métodos.
- Um envio de mensagem `a.m()`, onde o tipo de `a` é `A`, é transformado em `_a->vt[0](_a)`
O método `m` é invocado através de um dos elementos da tabela (neste caso, elemento da posição 0). O objeto é sempre o primeiro parâmetro.
- As classes são transformadas em estruturas contendo as variáveis de instância e mais um primeiro campo `vt` que é um ponteiro para um vetor de funções (métodos).
- Os métodos são transformados em funções adicionando-se como primeiro parâmetro um ponteiro chamado `this` para objetos da classe.
- Chamadas a métodos da superclasse (como em `super.put(1)`) são transformadas em chamadas estáticas.