Geração de Código em C para Cianeto

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
e-mail: josedeoliveiraguimaraes@gmail.com

18 de agosto de 2019

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

```
class Program {

   func run {
      var Int i, b;
      var Boolean primo;
      var String msg;
      Out.println: "Olá, este é o meu primeiro programa";
      Out.println: "Digite um número: ";
      b = In.readInt;
         // 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";
      }
      Out.println: 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 Cianeto.

```
/* 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

// crie funções readInt e readString
int readInt() {
    int n;
    char __s[512];
    gets(__s);
    sscanf(__s, "%d", &_n);
    return n;
}

char *readString() {
   char s[512];
   gets(s);
   char *ret = malloc(strlen(s) + 1);
   strcpy(ret, s);
   return ret;
}

/* 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 Cianeto. */

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 self
      cujo tipo é a estrutura que representa a classe, neste caso,
      _class_Program.
   */
void _Program_run( _class_Program *self )
{
      // os nomes de variáveis locais são precedidos por _
   int _i;
   int _b;
   boolean _primo;
      // Strings são mapeadas para char * em C
   char *_msg;

      // Out.println: com Strings são mapeados para puts em C
   puts( "Ola, este e o meu primeiro programa" );
   puts( "Digite um numero: ");
      // In.readInt é mapeado para uma chamada da função readInt
   b = readInt();
     // o restante do código é praticamente igual em Cianeto e C, a menos
     // de nomes de identificadores e parenteses
   _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
  };

   /*
      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);
  /* A linha acima poderia ser substituída por
```

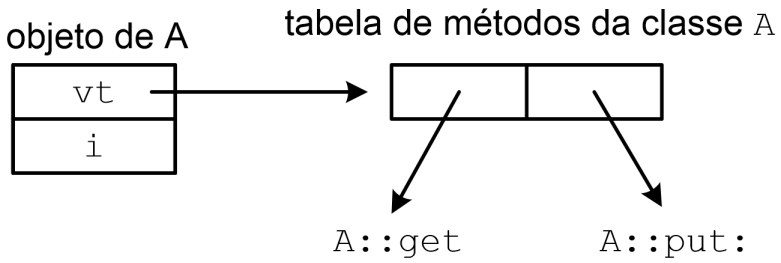

Figura 1: Representação de um objeto da classe `A`

```
        _Program_run(program);
  */
  return 0;
}
```

Os nomes de identificadores do programa em Cianeto, 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 Cianeto 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 mostra[1] um objeto da seguinte classe `A`:

```
class A {
   var Int i
   func get -> Int {
      return self.i;
   }
   func put: Int p_i {
      self.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;
```

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

```
    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 *self ) {
  return self->_A_i;
  }

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

O nome do primeiro parâmetro é sempre `self` 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();  /* Cianeto */
```

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 Cianeto para C. Apresentamos abaixo um programa em Cianeto seguido da sua tradução para C.

```
class A {
   var Int i
   func get -> Int {
      return self.i;
   }
   func put: int p_i {
      self.i = p_i;
   }
}


class Program {
   func run {
      var A a;
      var Int k;

      a = A.new;
      a.put: 5;
      k = a.get;
      Out.print: k;
   }
}
```

Tradução do programa Cianeto acima para C:

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

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

// readInt e readString

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 *self ) {
  return self->_A_i;
  }

void _A_put( _class_A *self, int _p_i ) {
  self->_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 *self )
{
     // 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);
     // Out.print: 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 {
   var Int lastInc
   private func add: int n {
```

```
      self.lastInc = n;
      super.put( super.get + n );
   }
   func print {
      Out.print: self.get;
   }
   override
   public func put: Int p_i {
      if p_i > 0 {
         super.put: p_i;
      }
   }
   func inc {
      self.add: 1;
   }
   func getLastInc -> Int {
      return self.lastInc;
   }
   final func 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 *self, int _n )
{
  self->_B_lastInc = _n;
  _A_put( (_class_A *) self, _A_get( (_class_A *) self ) + _n );
  }

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

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

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

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

void _B_atLast( _class_B *self ) {
}

   // 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 `self`. O método `get` é público.
2. chamada a métodos privados usando `self`, como em "`self.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.

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

Veremos abaixo como gerar código para cada uma destas situações.

1. "`self.get`" é traduzido para
   ```
   (self->vt[0])(self)
   ```
   ou melhor,
   ```
   ( (int (*)(_class_A *)) self->vt[0]) ( (_class_A *) self )
   ```
   Se o método é público, a ligação mensagem/método é dinâmica — é necessário utilizar a tabela de métodos mesmo o receptor da mensagem sendo `self`.

2. A chamada `self.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(self, 1)
   ```
   Não há necessidade de converter `self` pois o seu tipo é `_class_B` e `_B_add` é declarado como
   ```
   void _B_add( _class_B *self, int _n ) { ... }
   ```
   Chamadas a métodos privados nunca precisarão de conversão de tipo para `self`. Recordando, envios de mensagem para `self` 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( self, _p_i )
   ```
   ou
   ```
   _A_put( (_class_A *) self, _p_i )
   ```
   com conversão de tipos.

   Como o destino do envio de mensagem com seletor `put:` não é especificado (como `a` em `a.put: 5`), assume-se que ele seja `self`. Observe que em
   ```
   _A_put( (_class_A *) self, _p_i )
   ```
   é necessário converter um ponteiro para `_class_B`, que é `self`, em um ponteiro para `_class_A`. 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 )
   ```

4. 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`.

5. 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.

6. A declaração de `_class_B` é

   ```
   typedef
     struct _St_B {
   ```

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

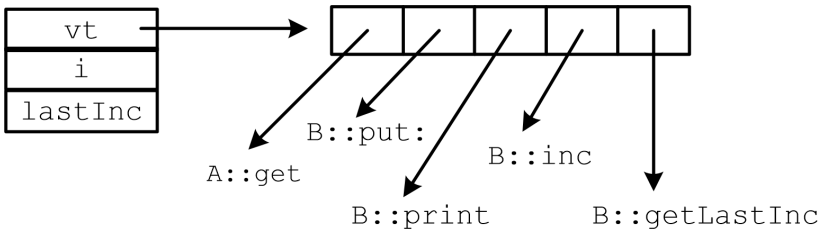

Figura 2: Objeto e tabela de métodos de `B`

```
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`).

7. 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`).

8. 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.

Pode-se enviar uma mensagem para uma variável de instância de uma classe. Se `B` tivesse uma variável de instância `bb` do tipo `B`, poderíamos ter um método

```
func printAlterEgo {
    Out.println: self.bb.print;
}
```

A tradução do envio de mensagem "`self.bb.print`" para C seria

```
(self->bb->vt[2])( self->_bb )
```

ou melhor,

```
( (void (*)(_class_B *)) self->bb->vt[2]) ( self->_bb )
```

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 {
   func f: A a {
      a.put: 5;
   }
}
```

Este método é transformado em

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

É fácil ver que o envio de mensagem "`x.f: t`" do código

```
var X x = X.new;
t = A.new;
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 = B.new;
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 do método `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
  };
```

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

Os envios de mensagem `In.readInt` e `In.readString` são transformados em chamadas para as funções `readInt` e `readString`, geradas logo no início do arquivo:

[4] `a` é o parâmetro formal de `f`.

```
int readInt() {
    int n;
    char __s[512];
    gets(__s);
    sscanf(__s, "%d", &_n);
    return n;
}

char *readString() {
   char s[512];
   gets(s);
   char *ret = malloc(strlen(s) + 1);
   strcpy(ret, s);
   return ret;
}
```

O envio de mensagem `Out.print: e1, e2, ... en` é equivalente a

```
Out.print: e1;
Out.print: e2;
...
Out.print: en;
```

E o envio de mensagem `Out.println: e1, e2, ... en` é equivalente a

```
Out.print: e1;
Out.print: e2;
...
Out.println: en;
```

O comando `Out.print: 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

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

O comando `Out.println: expr` é equivalente a

```
Out.print: expr; Out.print: "\n";
```

Se preferir, você poderá usar um único `printf`:

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

Se `b` for do tipo `String`, o código gerado pode ser

```
printf("%s\n", _b);
```

ou

```
puts(_b);
```

Métodos "shared" devem ser transformados em funções em C que não tomam `self` como primeiro parâmetro. Variáveis "shared" devem ser transformadas em variáveis globais. Um método shared

como nome `m` de uma classe de `A`, seja ele público ou privado, é traduzido para uma função com nome `_shared_A_m`. Uma variável shared com nome `x` de uma classe `A` é traduzida para uma variável global `_shared_A_x`.

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

```
class A {
   shared var Int n;
   shared func get -> Int {
      return A.n;
   }
   shared public func set: int n {
      A.n = n;
   }
}

class Program {
   func run {
      A.set: 0;
      Out.print: 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 _shared_A_n;
int _shared_A_get() {
   return _shared_A_n;
}
void _shared_A_set(int n) {
   _shared_A_n = n;
}

Func VTclass_A[] = {
};

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

```
void _Program_run( _class_Program *self )
{
   _shared_A_set(0);
   printf("%d", _shared_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;
- `nil` em Cianeto 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 Cianeto, 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 Cianeto, 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() { }
  };
```

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

```
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()
{
  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 Cianeto $\longrightarrow$ 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

Listing 1: Exemplo com a função `maximo`

```
#include <stdio.h>

void maximo() {
    puts("Olá ! Eu sou o máximo");
}

void main() {
    void (*f)();
    f = maximo;
    (*f)();         /* chama maximo */
    f();            /* chama maximo */
}
```

```
_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 `self` para objetos da classe.
- chamadas a métodos da superclasse (como em `super.put: 1`) são transformadas em chamadas estáticas.

A Ponteiros para Função

Em C, podemos declarar um ponteiro para função com a sintaxe

```
void (*f)();
```

neste caso, `f` é um ponteiro para uma função sem parâmetros e que retorna void. `f` pode apontar para uma função compatível:

```
f = maximo;
```

`maximo` é uma função declarada como

```
void maximo() {
   puts("Olá ! Eu sou o máximo");
}
```

`maximo` pode ser chamada a partir de `f` usando-se quaisquer das sintaxes abaixo.

```
(*f)();      /* chama maximo */
f();         /* chama maximo */
```

Veja o programa completo na listagem 1.

Podemos definir um tipo em C através de `typedef`:

```
typedef
   int Number[10];
```

Agora `Number` é sinônimo de vetores inteiros:

```
Number v;  //   v é um vetor de inteiros de 10 posições
```

Da mesma forma, podemos definir `Func` como um ponteiro para funções:

```
typedef
   void (*Func)();
Func f;
f = maximo;
f();   // chama maximo
```

Podemos definir também um vetor de ponteiros para funções:

```
Func v[] = {
   maximo,
   minimo
};
```

`minimo` é definida como

```
void minimo() {
   puts("Oi. Sou o mínimo");
}
```

Agora, `v` é um vetor de ponteiros para funções. Ou melhor, `v` é um vetor de ponteiros para funções que não têm parâmetros nem retornam nada. Então `v[0]` é um ponteiro para uma função:

```
v[0]();   //   chama maximo
v[1]();   //   chama minimo
(*v[0])();  // chama maximo
(*v[1])();  // chama minimo
```

Até agora vimos apenas funções sem parâmetros e sem tipo de retorno. E se tivermos uma função como `addOne`?

```
int addOne(int i) {
   return i + 1;
}
```

Vejamos:

```
void (*f)();
f = addOne;   // oops ...
f();  // oops ...
```

Na atribuição, há um erro de tipos. Estamos atribuindo uma função com um parâmetro e valor de retorno para um ponteiro para uma função sem parâmetros e sem valor de retorno. Temos que usar uma conversão de tipos (*cast*):

```
 f = (void (*)()  )  addOne;
```

Listing 2: Exemplo com a função `addOne`

```
#include <stdio.h>

int addOne(int i) {
   return i + 1;
}

void main() {
   void (*f)();
   int n;
   f = (void (*)()  ) addOne;
   n = ((int (*)(int)  ) f)(1);
   printf("%d\n", n);
}
```

O tipo "`void (*)()`" lê-se "ponteiro para uma função sem parâmetro retornando void". Na chamada de `f`, há outro erro. Estamos chamando a função sem passar o parâmetro. `f` aponta para `addOne` que espera um parâmetro. Temos que converter `f` para o tipo de `addOne` antes de chamar esta função:

```
n = ((int (*)(int)  ) f)(1);
```

O tipo "`int (*)(int)`" é um ponteiro para uma função que tem um inteiro como parâmetro e retorna valor inteiro. O código completo deste exemplo está na Listagem 2.

Podemos colocar funções de vários tipos em um único vetor:

```
Func v[] = {
   maximo,
   addOne,
   minimo
};


...
// chama maximo
v[0]();
//  chama addOne passando 5 como parâmetro
n = ((int (*)(int)  )  v[1])(5);
// chama minimo
v[2]();
```

Estudando detalhadamente o exemplo acima, descobrimos que há um erro de tipos na inicialização de `v`. Cada elemento de `v` deve ser do tipo `Func`, função sem parâmetros retornando `void`. Mas `addOne` possui um parâmetro e retorna `int`. Então devemos usar uma conversão de tipos:

```
Func v[] = {
   maximo,
   (int (*)(int) ) addOne,
   minimo
};
```

É so.