Geração de Código em C para Simples

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

March 19, 2002

1 Implementação de Simples

Esta seção descreve a tradução dos programas de Simples para C. Os nomes de alguns identificadores do programa em Simples, como classes e métodos, terão seus nomes alterados na tradução. Os nomes de variáveis locais e de instância são mantidos. Contudo, se o tipo da variável pa no programa em Simples 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 um objeto da classe `A` da Figura 2.[1]

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

Cada classe possui um vetor de ponteiros onde cada entrada aponta para um dos métodos 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 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:

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

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

```
class A
  private:
    var i : integer;
  public:
    proc get() : integer;
      begin
      return self.i;
      end
    proc put( p_i : integer );
      begin
      self.i = p_i;
      end
end
```

Figure 2: Uma classe em Simples

```
typedef
  struct St_A {
    Func *vt;
    int i;
    } class_A;
```

O tipo Func é definido como

```
typedef
  void (*Func)();
```

Isto é, um ponteiro para uma função.

Cada método de A 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->i;
  }

void A_put( class_A *self, int p_i ) {
  self->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.

Agora, a tabela de métodos 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.

Como dissemos anteriormente, cada objeto de A aponta para um vetor de métodos 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 = A.new();  /* Simples */
```

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 métodos

```
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.

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);
```

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 Simples para C. O programa Simples da Figura 3 é traduzido para o programa C mostrado nas Figuras 4 e 5. 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.

Considere agora a classe B da Figura 6. Esta classe possui vários aspectos ainda não examinados:

1. o método print envia a mensagem get para self.

2. o método put de B chama o método put de A através de super.put(p_i);[2]

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

```
class A
  private:
    var i : integer;
  public:
    proc get() : integer;
      begin
      return self.i;
      end
    proc put( p_i : integer );
      begin
      self.i = p_i;
      end
end


class Program
  public:
    proc run()
      var a : A;
          k : integer;
      begin
      a = A.new();
      a.put(5);
      k = a.get();
      write(k);
      end
end
```

Figure 3: Um programa em Simples

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

typedef
  void (*Func)();

typedef
  struct St_A {
    Func *vt;
    int i;
    } class_A;

class_A *new_A();

int A_get( class_A *self ) {
  return self->i;
  }

void A_put( class_A *self, int p_i ) {
  self->i = p_i;
  }

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;
  }
```

Figure 4: Tradução do programa da Figura 3 para C

```
typedef
  struct St_Program {
    Func *vt;
    } class_Program;

class_Program *new_Program();

void Program_run( class_Program *self )
{
  class_A *a;
  int k;

  a = new_A();
  ( (void (*)(class_A *, int)) a->vt[1] )(a, 5);
  k = ( (int (*)(class_A *)) a->vt[0] )(a);
  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;
 }

void 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);
  }
```

Figure 5: Continuação da tradução do programa da Figura 3 para C

```
class B subclassOf A
  public:
    proc print();
      begin
      write( self.get() );
      end
    proc put( p_i : integer );
      begin
      if p_i > 0
      then
        super.put(p_i);
      endif
      end
end
```

Figure 6: Herança de A por B com acréscimo e redefinição de métodos

3. o acréscimo de métodos nas subclasses (print);

4. a redefinição de métodos nas subclasses (put).

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 )
   ```

2. 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 da mensagem put não é especificado (com em a.put(5)), assume-se que ele seja self.

3. O acréscimo de métodos em subclasses faz com que a tabela de métodos aumente proporcionalmente. Assim, a tabela de métodos para B possui uma entrada a mais do que a de A, para print.

4. 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 7.

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

Figure 7: Objeto e tabela de métodos de B

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:
    proc f( a : A );
      begin
      a.put(5);
      end
end
```

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 a chamada

```
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

[4] a é o parâmetro formal de f.

```
Func VTclass_B [] = {
  A_get,
  B_print,
  B_put
  };
```

a execução da função f chamaria B_print.

A transformação da classe B para linguagem C é mostrada na Figura 8.

Observe que a estrutura class_B contém todas as variáveis de instâncias herdadas de A — neste caso, variável i. Se B definisse variável de instância

```
var j : integer;
```

a declaração de class_B seria

```
typedef
  struct St_B {
    Func *vt;
    int i;
    int j;
    } class_B;
```

As variáveis da superclasse aparecem antes das variáveis da subclasse.

Observe também na chamada super.put(p_i), transformada 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.

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]

```
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, *pc = &c;
```

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

```
typedef
  struct St_B {
    Func *vt;
    int i;
    } class_B;

class_B *new_B();

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);
  }

Func VTclass_B[] = {
  (void (*) () ) A_get,
  (void (*) () ) B_put,
  (void (*) () ) B_print
  };

class_B *new _B()
{
  class_B *t;

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

Figure 8: Tradução da classe B para a linguagem C

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

Figure 9: Tabela comparativa dos tempos de chamada de função

```
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 na Figura 9. 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.

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