UFV
Universidade Federal de Viçosa

INF 112 - Programação II

Escopo, Cuidados da Memória e Static

Introdução

- Abstração
  - Simplificação de um problema difícil
  - É o ato de representar as características essenciais sem incluir os detalhes por trás
- Ocultação de dados
  - Informações desnecessárias devem ser escondidas do mundo externo (usuários)

Lembrando

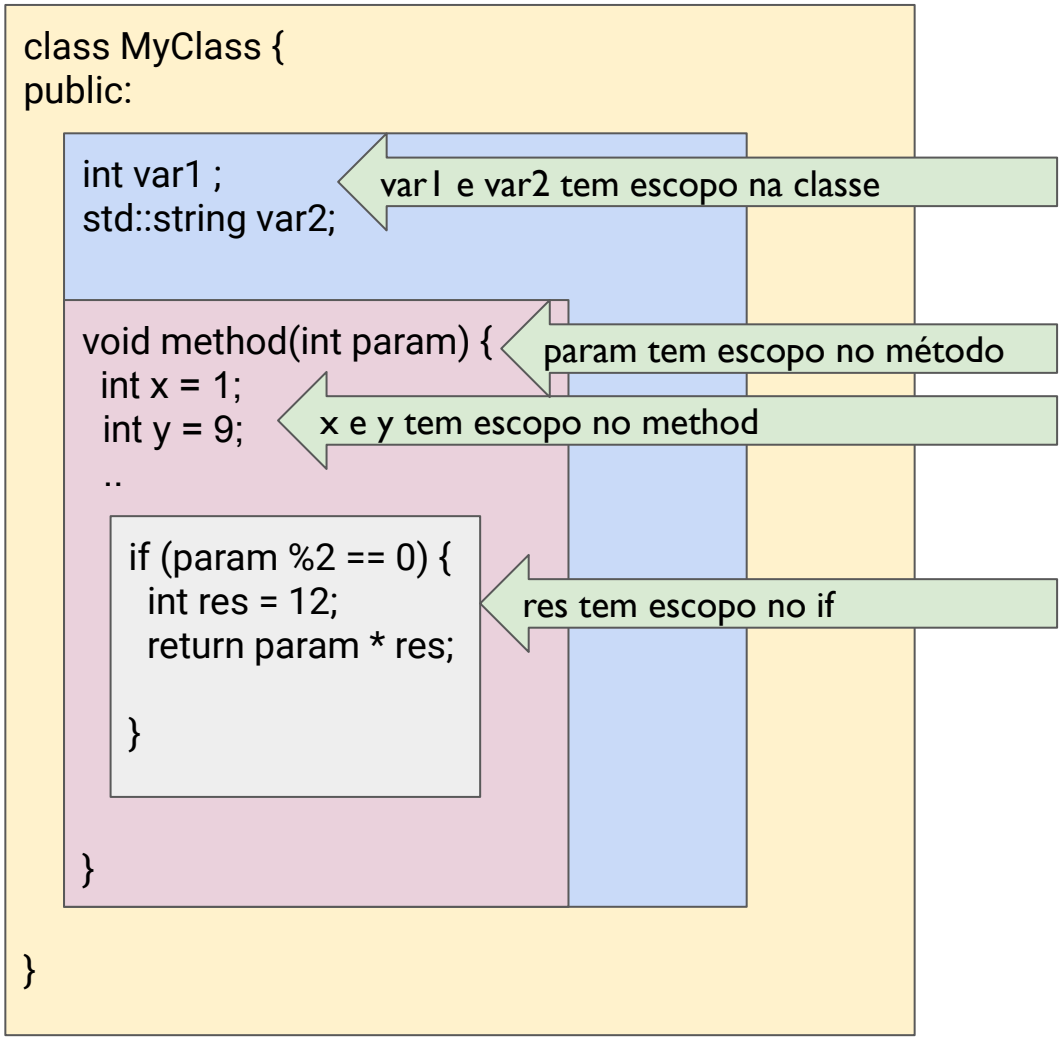
- Classes
  - Definem o comportamento dos objetos
  - Atributos, memória
  - Métodos, como os objetos são alterado
- Objetos
  - Instâncias na memória
  - Um estado
- Classes/Struct
  - Na classe tudo é **private** inicialmente

Escopo e Visibilidade

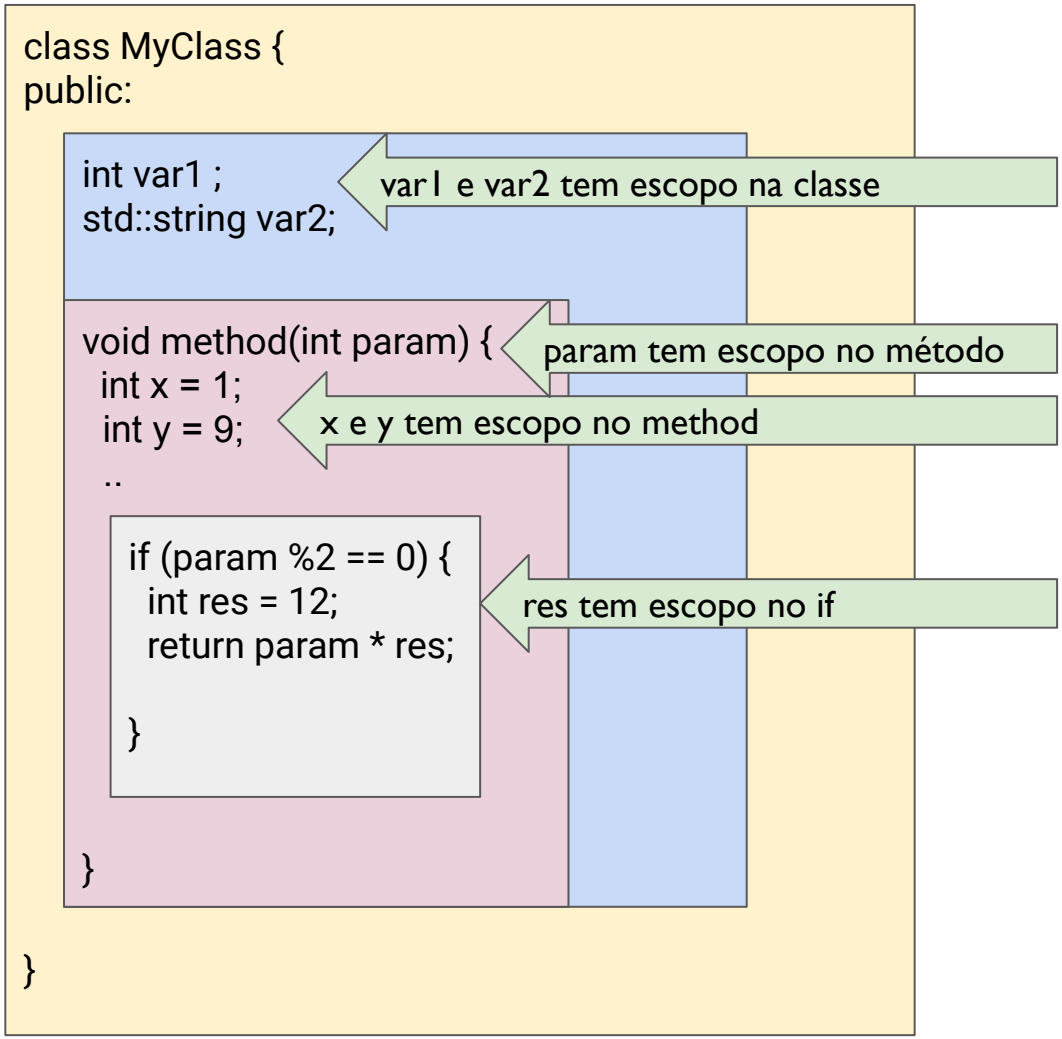
Ocultação 1: Escopo

- O escopo de uma variável é a região de um programa dentro do qual a variável pode ser referenciada via de seu nome
- O escopo define quando o sistema aloca e libera memória para armazenar a variável
- Memória alocada no heap existem além do escopo da mesma

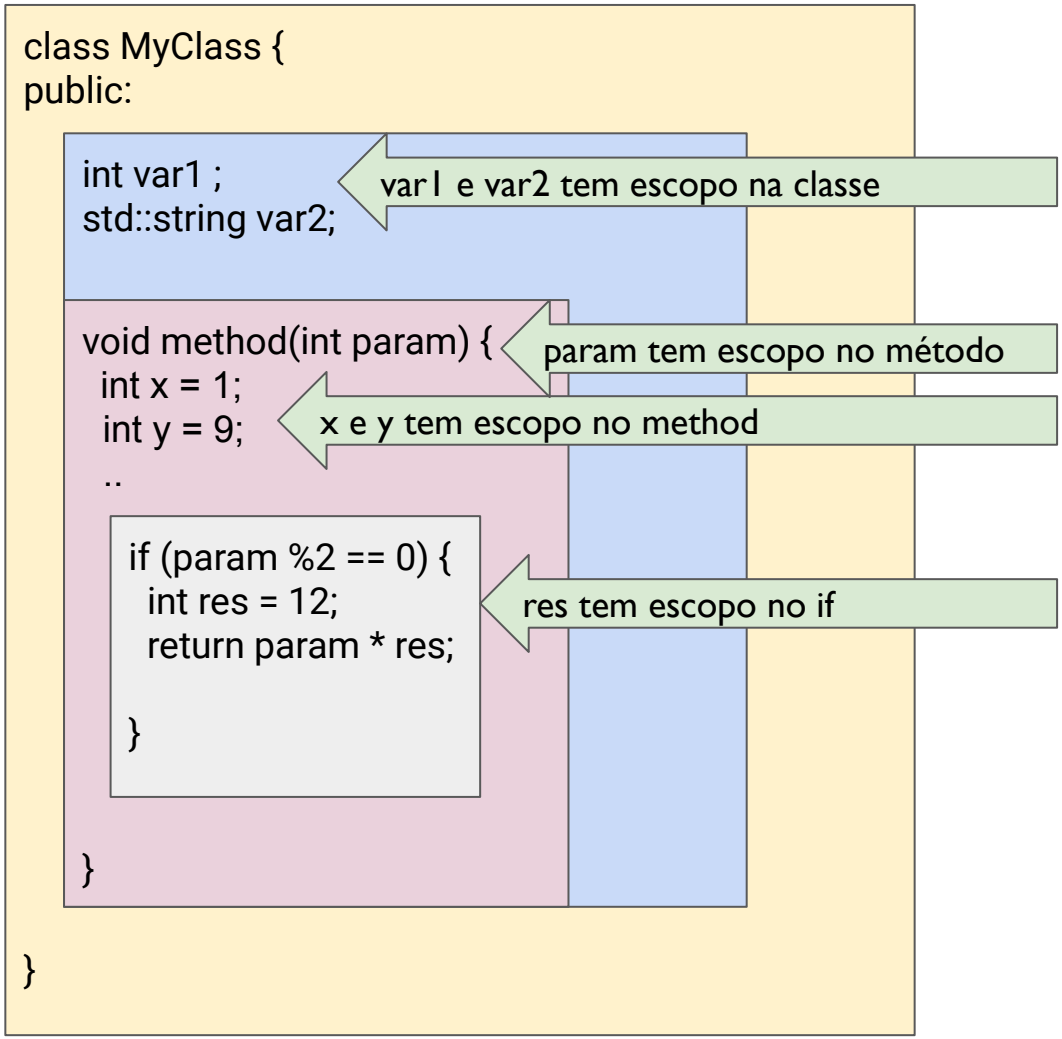
Escopo


class MyClass {
public:
int var1 ;
std::string var2;
var1 e var2 tem escopo na classe
void method(int param) {
param tem escopo no método
int x = 1;
int y = 9;
x e y tem escopo no method
..
if (param %2 == 0) {
int res = 12;
return param * res;
}
res tem escopo no if
}
}

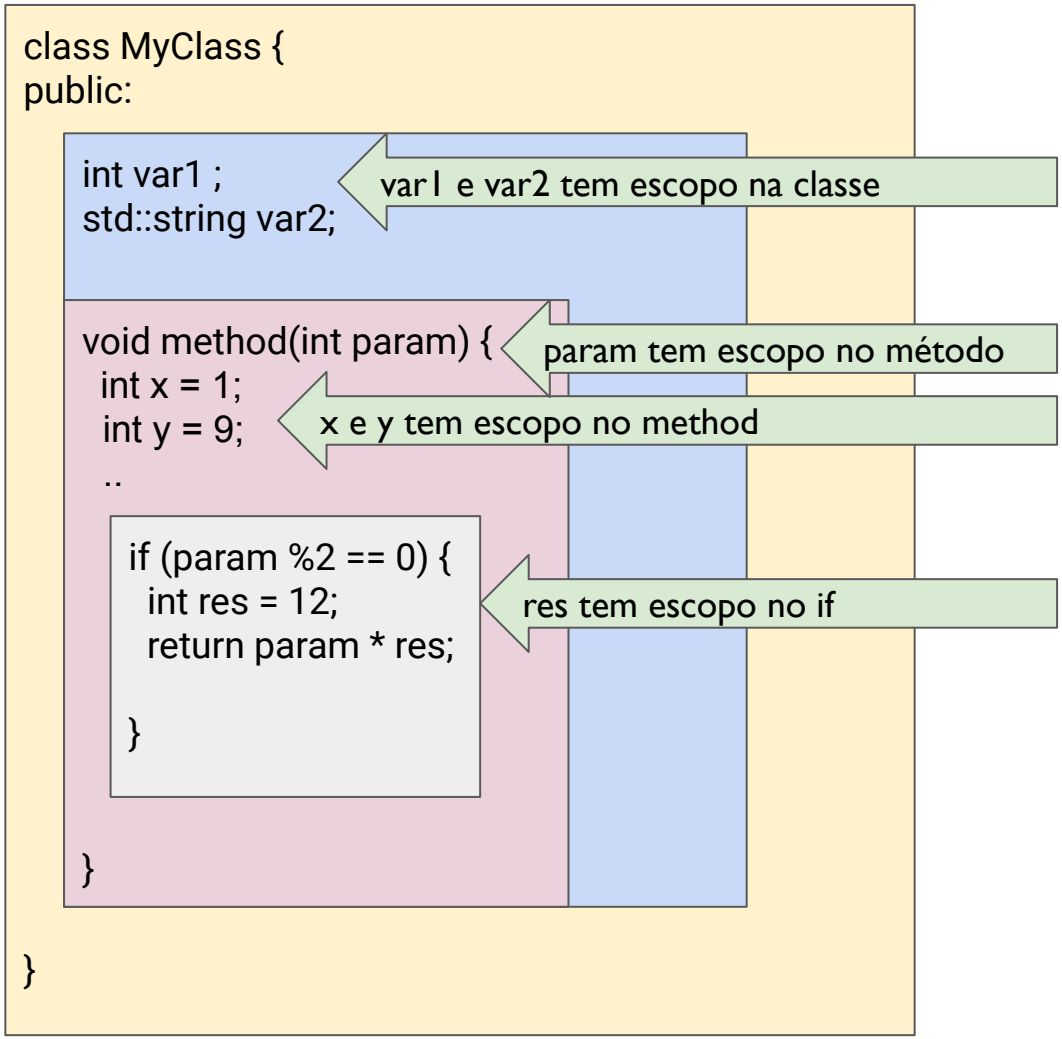
Quando o método termina?



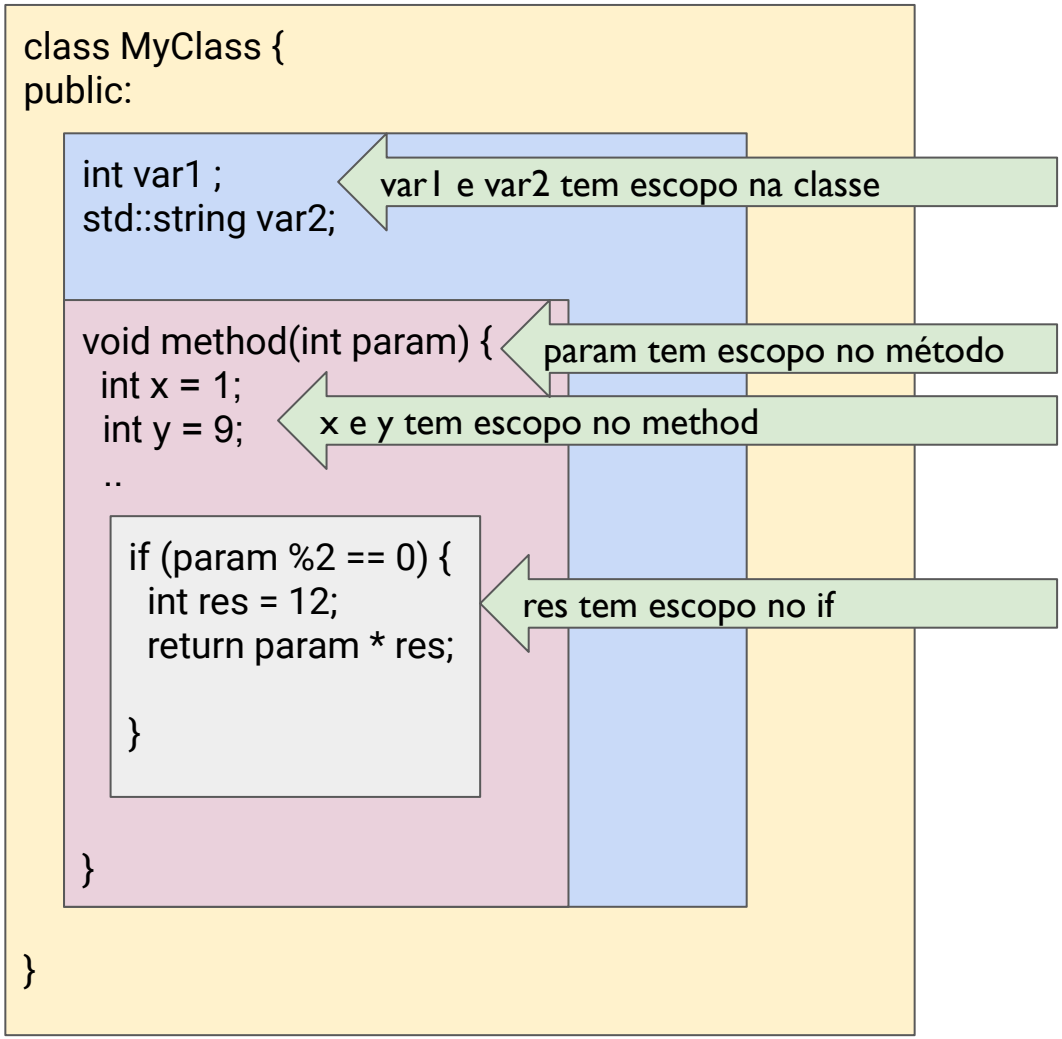
x, y, res, param I "somem" da pilha



Quando o objeto é destruído?



var1 e var2 somem



Escopo e Estado

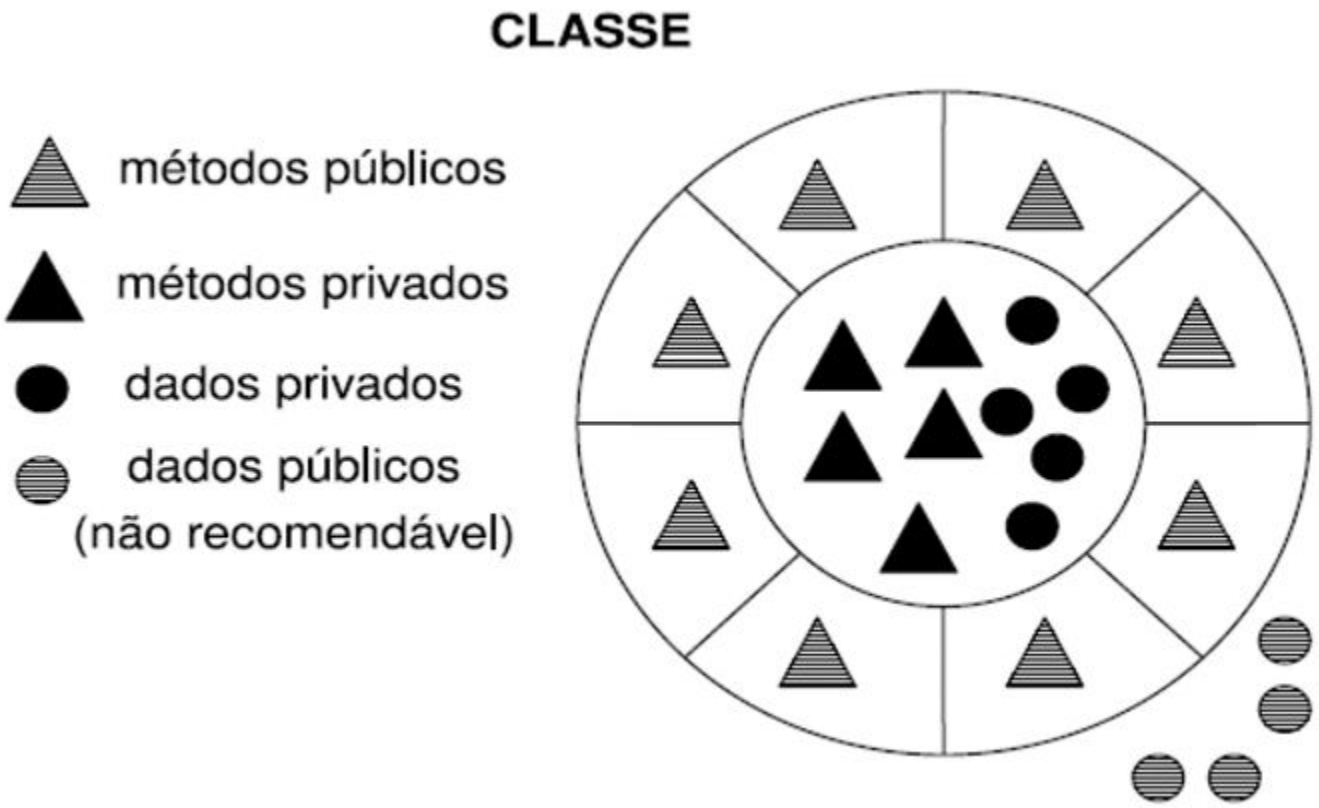
- O escopo define o **estado**
  - Estado do método
    - Valores das variáveis do método
  - Estado do objeto
    - Valores dos atributos
- Programação OO
  - Métodos

    +
  - Estado do objeto

Ocultação 2: Encapsulamento

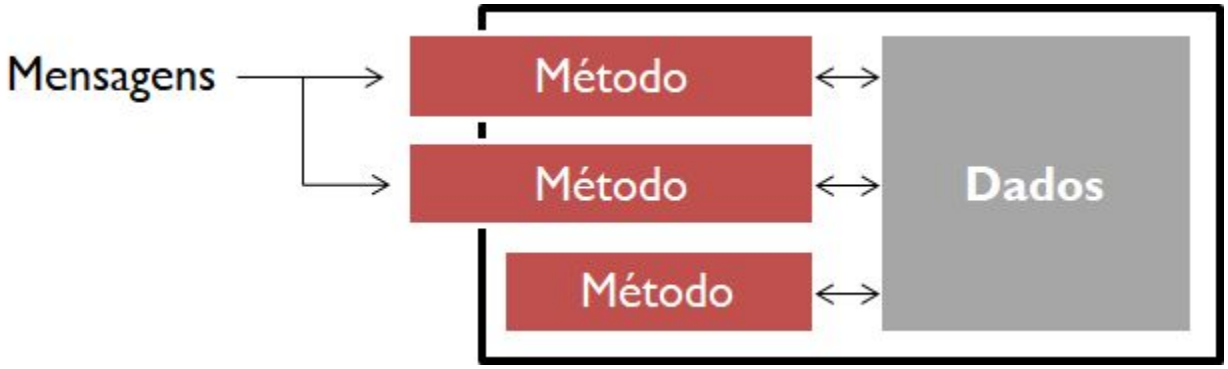
- Encapsulamento
  - Mecanismo que coloca juntos os dados e suas funções associadas, mantendo-os controlados o acesso
- Proporciona abstração
  - Separa a visão externa da visão interna
  - Protege a integridade dos dados do Objeto

Encapsulamento



Chamada de Métodos

- Informações encapsuladas em uma Classe
  - Estado (dados)
  - Comportamento (métodos)



Encapsulamento

Benefícios

- Desenvolvimento
  - Melhor compreensão de cada classe
  - Facilita a realização de testes
- Manutenção/Evolução
  - Impacto reduzido ao modificar uma classe
  - Interface deve ser o mais constante possível

Encapsulamento

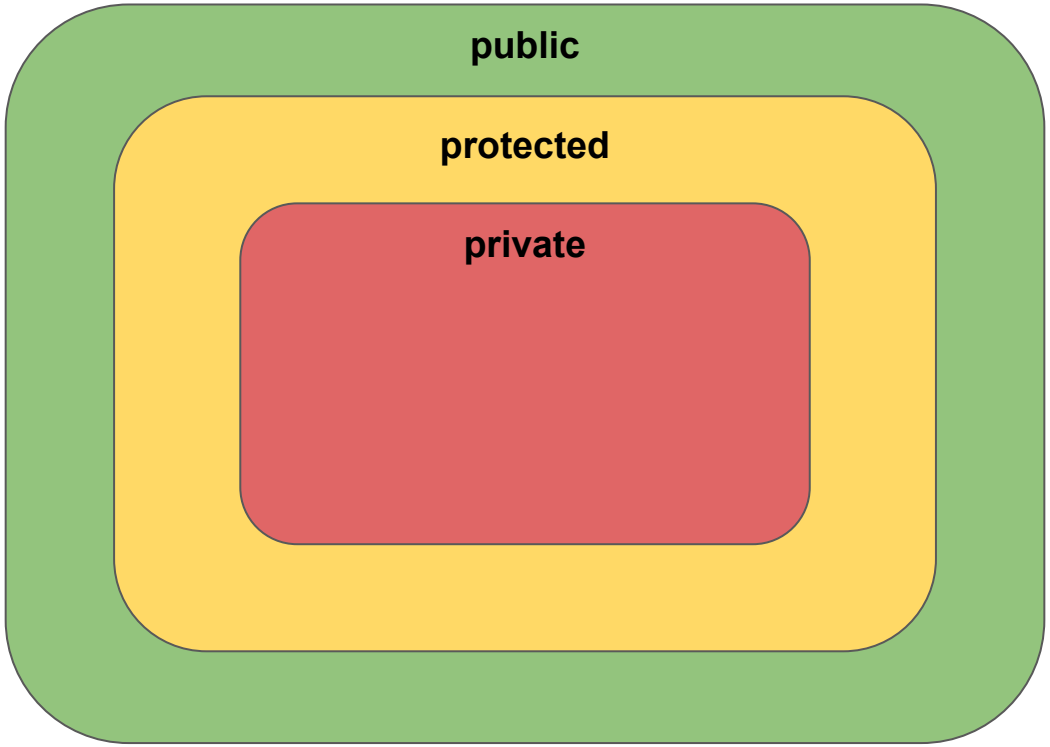
- Encapsulamento ocorre nas classes
- O comportamento e a interface de uma classe são definidos pelos seus membros
  - Atributos
  - Métodos
- Fazem uso dos modificadores de acesso

Encapsulamento
C++

- Modificadores de acesso
  - Public
  - Protected
  - Private
- Membros declarados após o modificador

Encapsulamento

Modificadores de acesso



Encapsulamento

Modificadores de acesso - Public

- Permite que o membro público possa ser acessado de qualquer outra parte do código
- Mais liberal dos modificadores
  - Fazem parte (definem) o contrato da classe
  - Deve ser usado com responsabilidade
  - Não é recomendado
    - Por quê?

Encapsulamento

Modificadores de acesso - Public

- Qualquer outra classe acessa x, y
- Ferimos o encapsulamento
- Atributos public são raros, mas existem
  - Tenho certeza que mudar o mesmo não muda em nada a lógica de estado do objeto

```
class Ponto {
public:
  int x;
  int y;
};
```

Encapsulamento

Lei

- No sentido de não quebrar a encapsulação, é muito importante que os membros de uma classe (atributos e métodos) sejam visíveis apenas onde estritamente necessário
  - A lei é: **"Não posso quebrar o que não posso acessar"**

Encapsulamento

Lei

- Por isso, é comum usarmos "**private**" como especificador de controle de acesso para atributos de uma classe
- Já que é freqüente querermos que métodos de uma classe sejam chamados por objetos de outras classes, não é raro usarmos "**public**" como especificador

Encapsulamento

Modificadores de acesso - Private

- Permite que o membro privado possa ser acessado por métodos da mesma classe
- O mais restritivo dos modificadores
  - Deve ser empregado sempre que possível
  - Utilizar métodos auxiliares de acesso
- Quando não há declaração explícita
  - Nível padrão (implícito)

Encapsulamento

Modificadores de acesso - Private

```
class Ponto {
private:
  int _x;
  int _y;
public:
  Ponto(int x, int y):
    _x(x), _y(y) {}
};
```

```
int main() {
  Ponto p(7, 10);
  p._x = 9;
}
```

Encapsulamento

Acessando e modificando atributos

- Evitar a manipulação direta de atributos
  - Acesso deve ser o mais restrito possível
  - De preferência todos devem ser private
- Sempre utilizar métodos auxiliares
  - Melhor controle das alterações
  - Acesso centralizado

Encapsulamento

Getters e Setters

- Convenção de nomenclatura dos métodos
- Get
  - Os métodos que permitem apenas o acesso de consulta (obter) devem possuir o prefixo get
- Set
  - Os métodos que permitem a alteração (definir) devem possuir o prefixo set

Encapsulamento

Getters e Setters

- Atributos **private** devem possuir get
  - set é opcional.
  - set CPF é raro.
- Nomenclatura alternativa
  - Atributos booleanos devem utilizar o prefixo "is" ao invés do prefixo get
- Melhora a legibilidade e entendimento

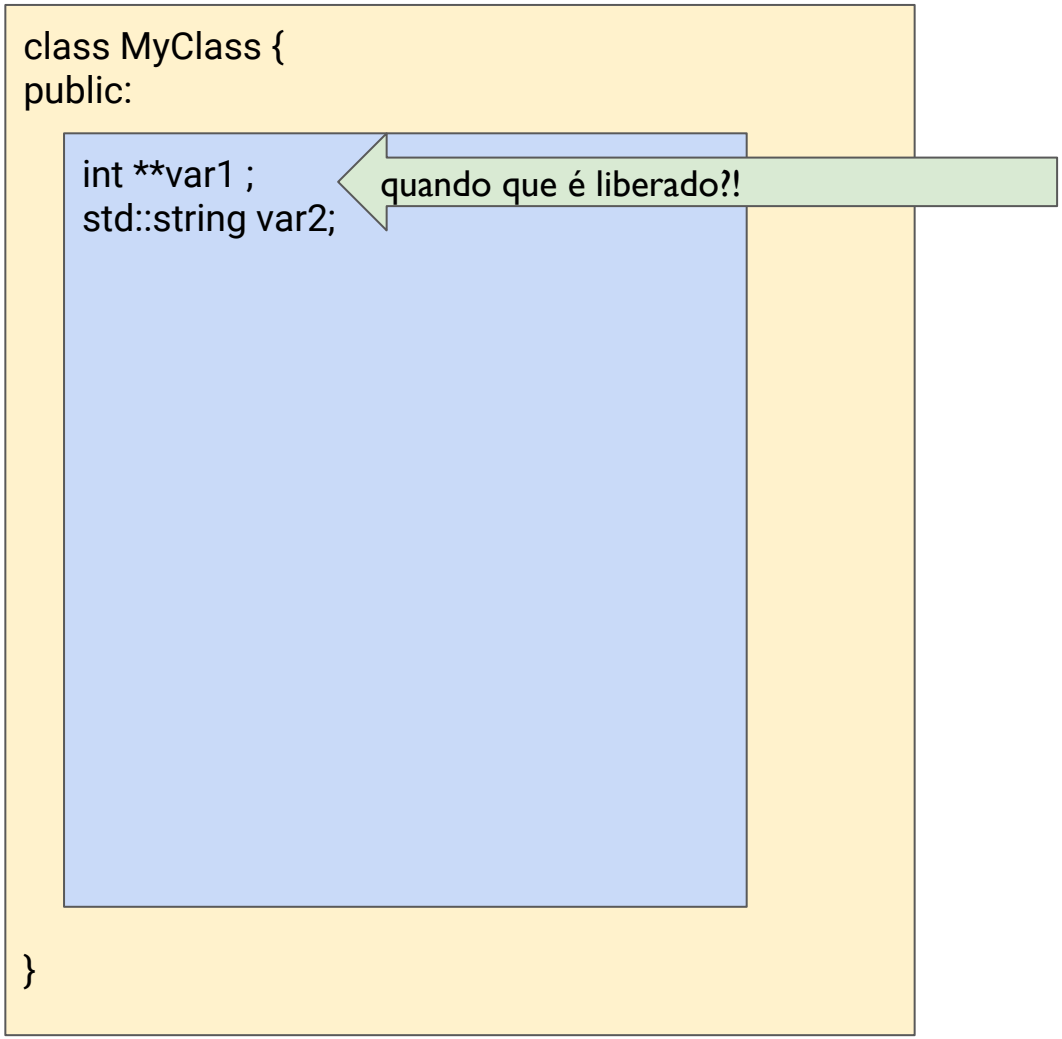
Encapsulamento

Modificadores de acesso - Protected

- Permite que o membro possa ser acessado apenas por outras classes que
  - Fazem parte da hierarquia (derivadas)
    - Próximas aulas
  - Classes "amigas"
    - Algo bem específico em C++

Destrutores

Ponteiros e Escopo



Destrutores

Relembrando….

- Destrutores são chamados tanto para:
  - Objetos no heap
    - Depois de um delete
  - Objetos no stack
    - Depois que a função termina

Cuide da memória que você alocou!

Problema

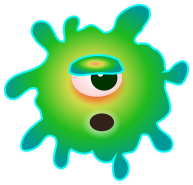
Vírus que se reproduz

Vamos mudar o programa do Vírus para:

1. Garantir um limite de reprodução
   - Nunca se reproduz além do mesmo
2. Retornar uma cópia de si mesmo
   - Sempre que se reproduzir
3. Guardar ponteiros para as cópias
   - Assim rastrear todos os vírus
     - Alguns vírus têm mutações sazonais

Exemplo

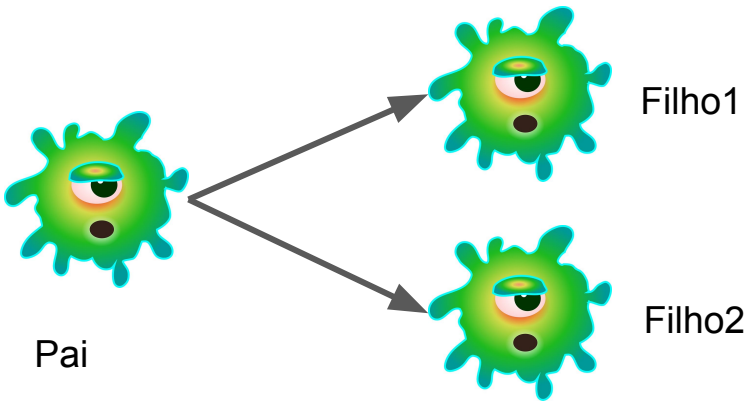
```
_forca=0.2
```



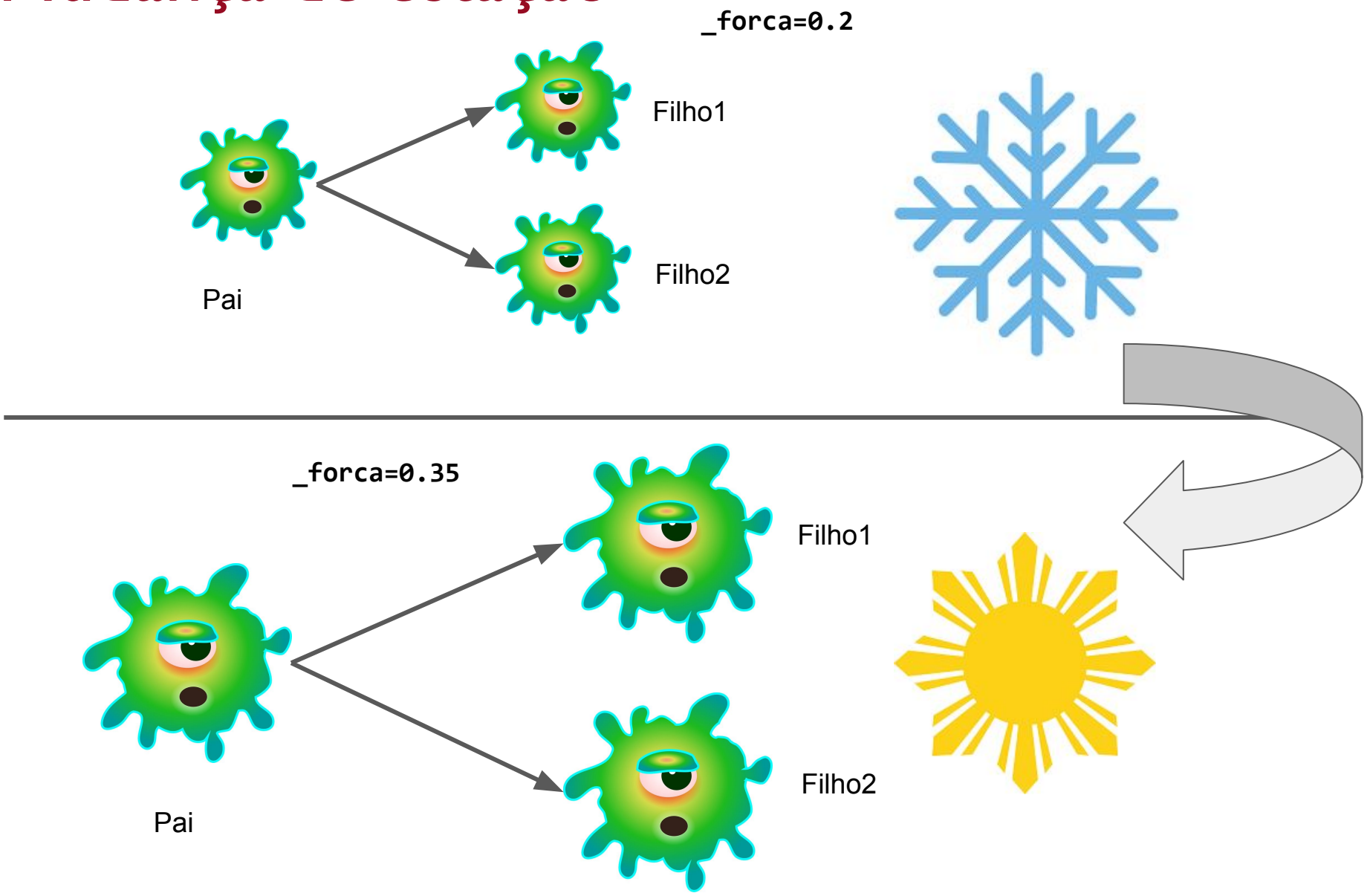
Pai

Exemplo

```
_forca=0.2
```



Mudança de estação



virus.h

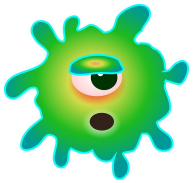
```
#ifndef INF112_VIRUS_H
#define INF112_VIRUS_H

#include <string>

class Virus {
private:
  // Guarda os filhos alocados. Todos no heap!
  Virus **_filhos;
  // Número de filhos atual
  int _numero_filhos;
  // . . . Mesmos atributos da aula anterior ... //
public:
  // Construtor. A capacidade de reprodução define o num filhos
  Virus(std::string nome, double forca, int capacidade_reproducao);
  // Destrutor para desalocar os filhos
  ~Virus();
  // . . . Mesmos métodos de antes ... //
  // Reproduz o vírus. Nosso foco!
  Virus *reproduzir();
};
#endif
```

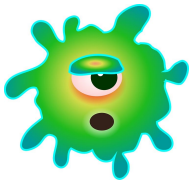
Construtor

```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



Construtor

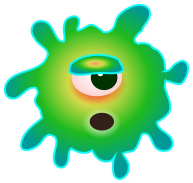
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



_nome = "Bob"

Construtor

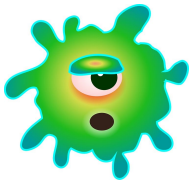
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



```
_nome = "Bob"
_forca = 0.2
```

Construtor

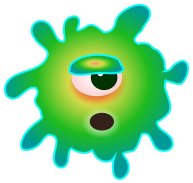
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
```

Construtor

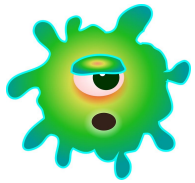
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 0
```

Construtor

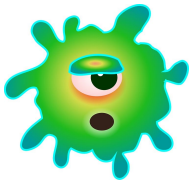
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  // vetor de tamanho capacidade_reproducao para os filhos (no heap)
  _filhos = new Virus*[capacidade_reproducao]();
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 0
_filhos =
```

Reprodução

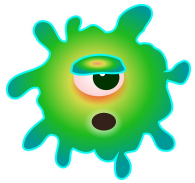
```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 0
_filhos =
```

Reprodução

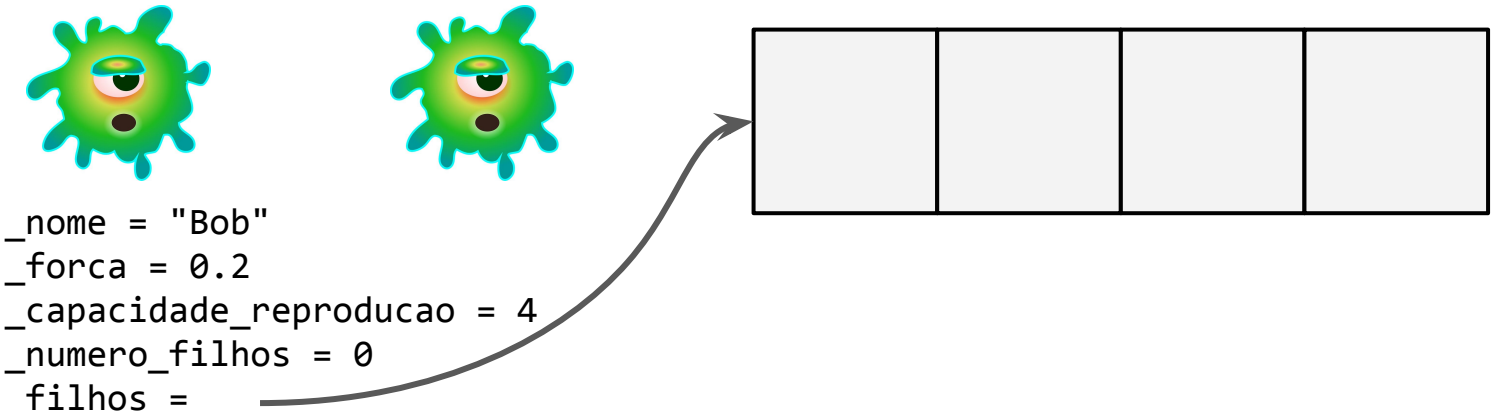
```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 0
_filhos =
```

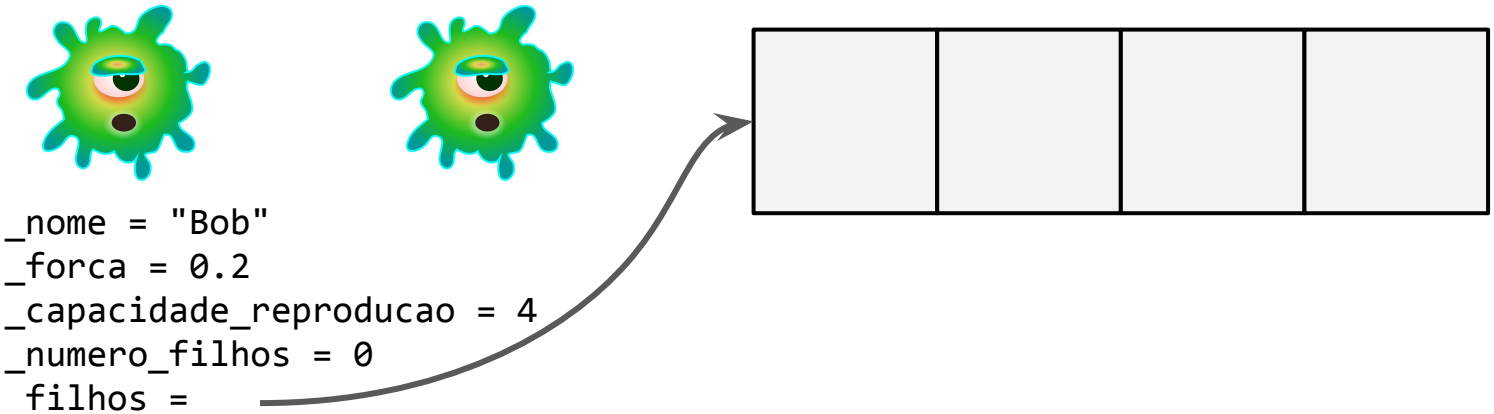
Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```



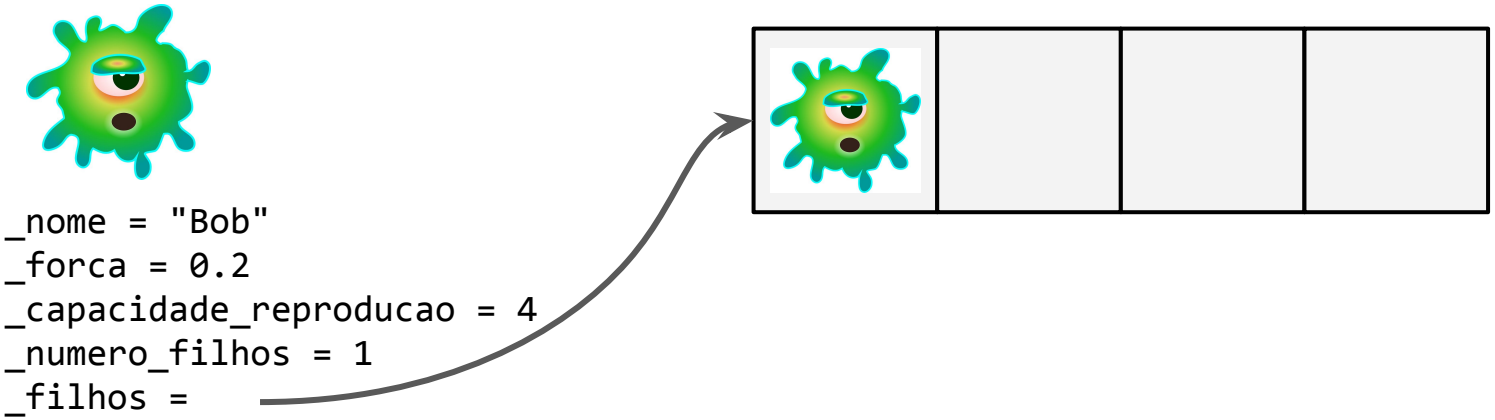
Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```



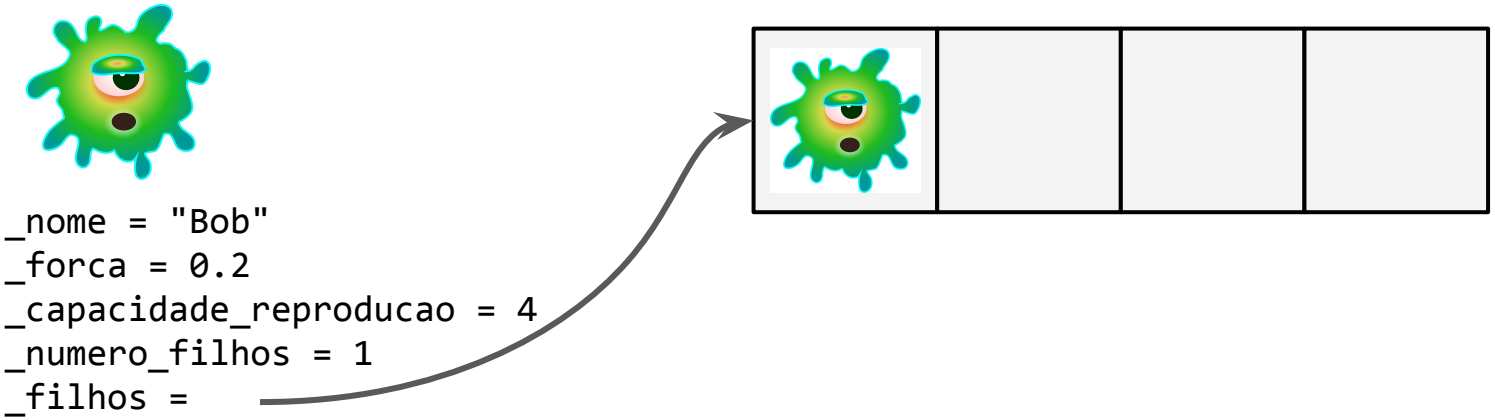
Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```



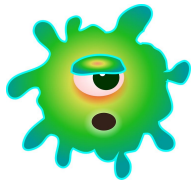
Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

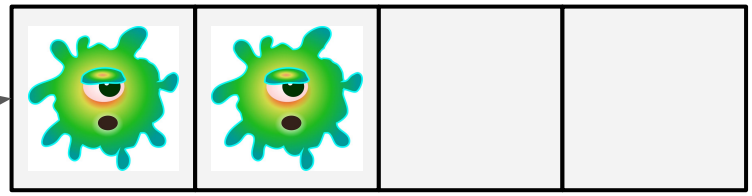


Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

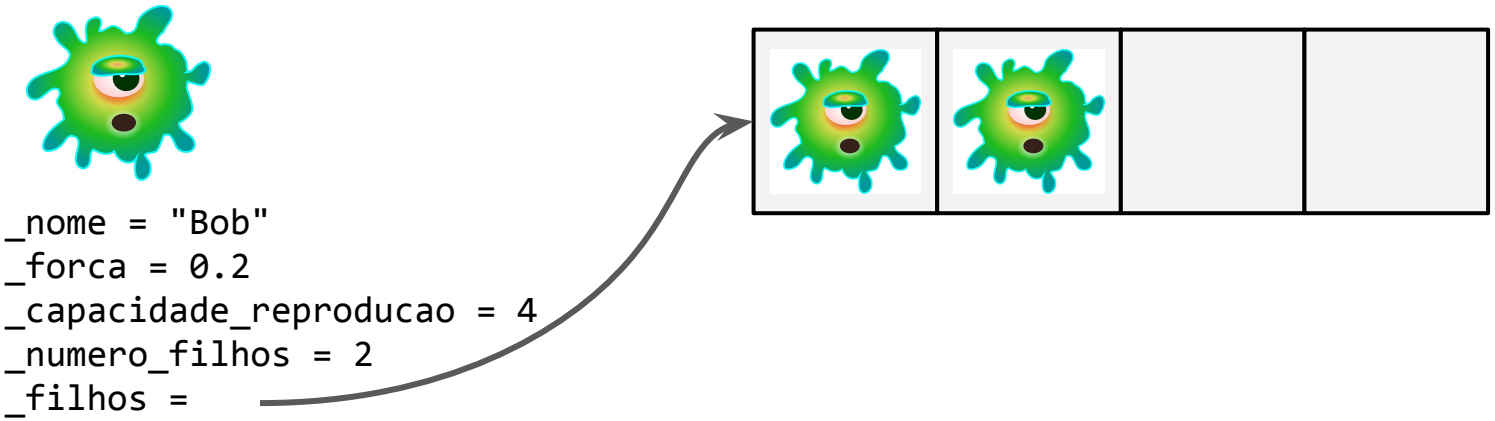


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 1
_filhos =
```



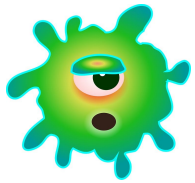
Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

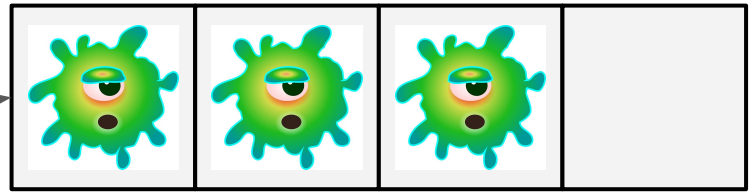


Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

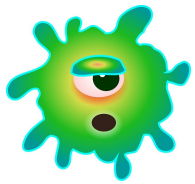


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 3
_filhos =
```

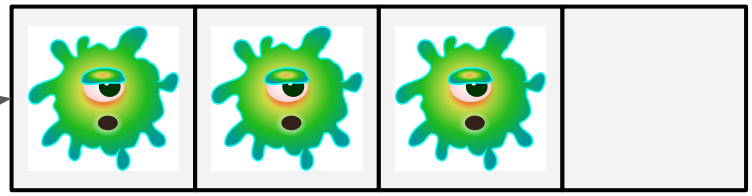


Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

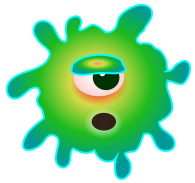


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 3
_filhos =
```

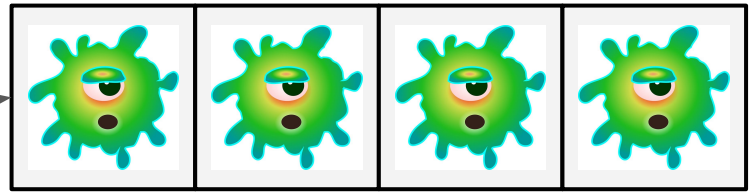


Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

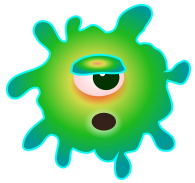


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 4
_filhos =
```

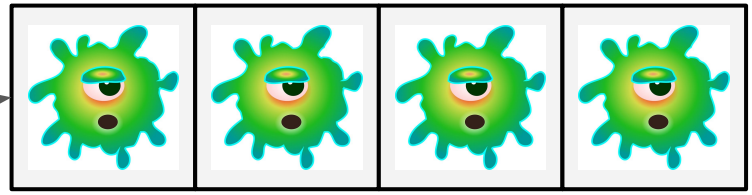


Reprodução

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

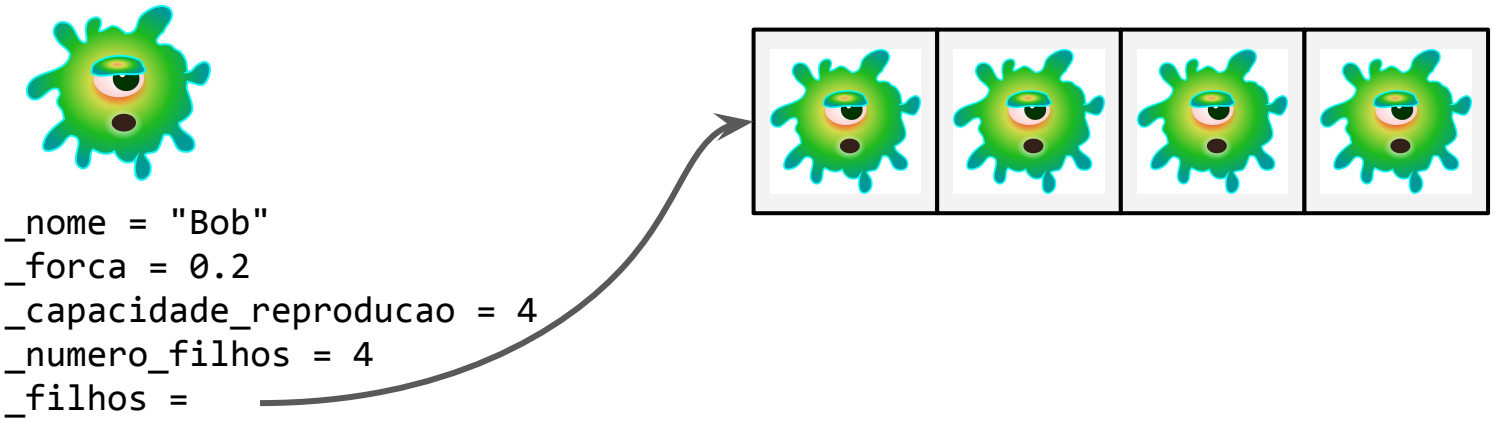


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 4
_filhos =
```



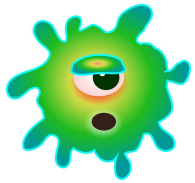
Retornamos null indicando que não dá mais

```
Virus *Virus::reproduzir() {
  if (this->_numero_filhos == this->_capacidade_reproducao) {
    return nullptr;
  }
  // Aloca uma um novo filho.
  Virus *novo_virus = new Virus(_nome, _forca, _capacidade_reproducao);
  _filhos[_numero_filhos] = novo_virus;  // Guarda copia em um vetor
  _numero_filhos += 1;                   // Aumenta o número de filhos
  return _filhos[_numero_filhos - 1];    // Retorna ponteiro para copia
}
```

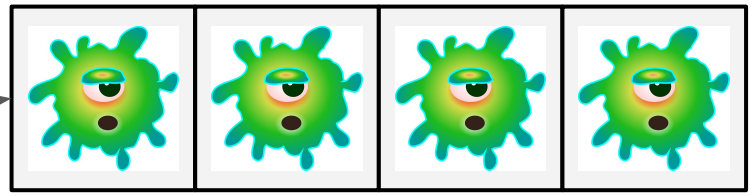


Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
          delete _filhos[i];
     delete[] _filhos;
  }
}
```

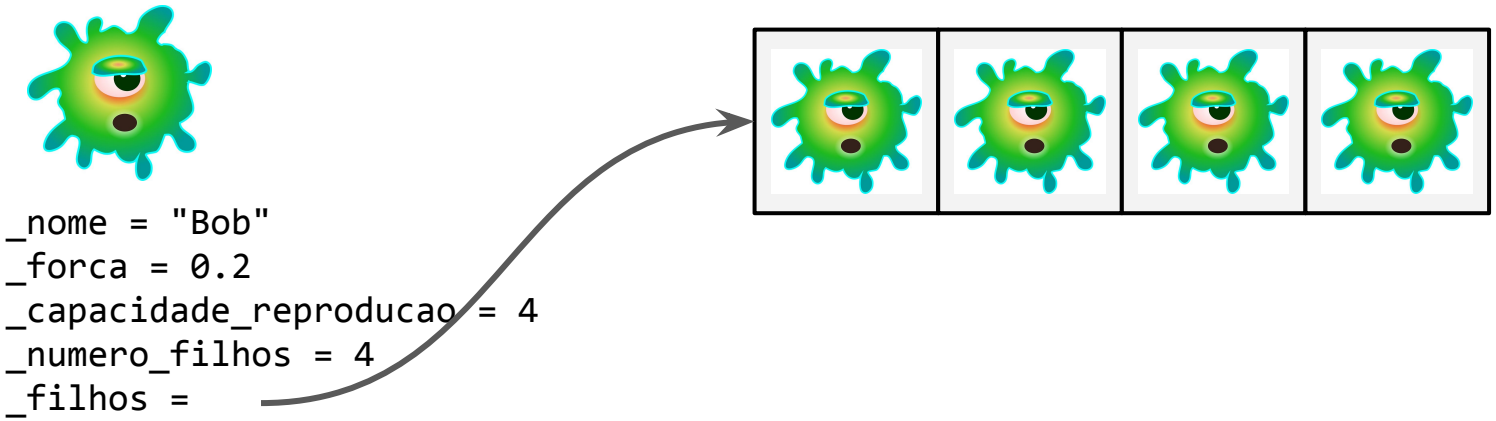


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 4
_filhos =
```



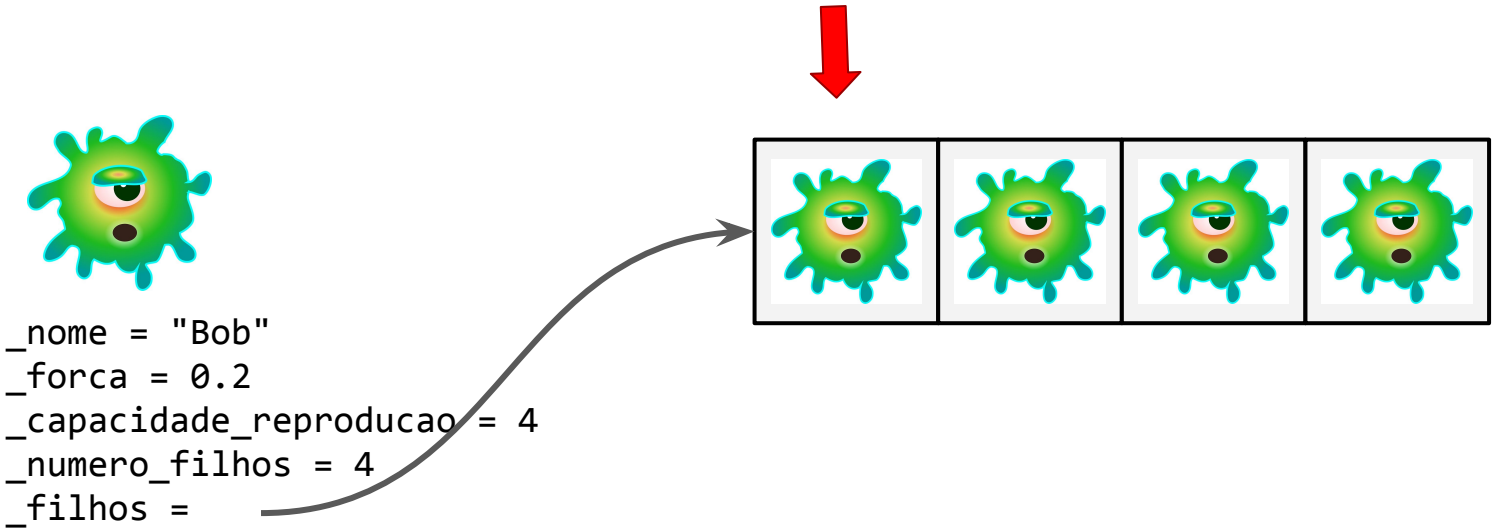
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
    for (int i = 0; i < _numero_filhos; i++)
      if (_filhos[i] != nullptr)
        delete _filhos[i];
    delete[] _filhos;
  }
}
```



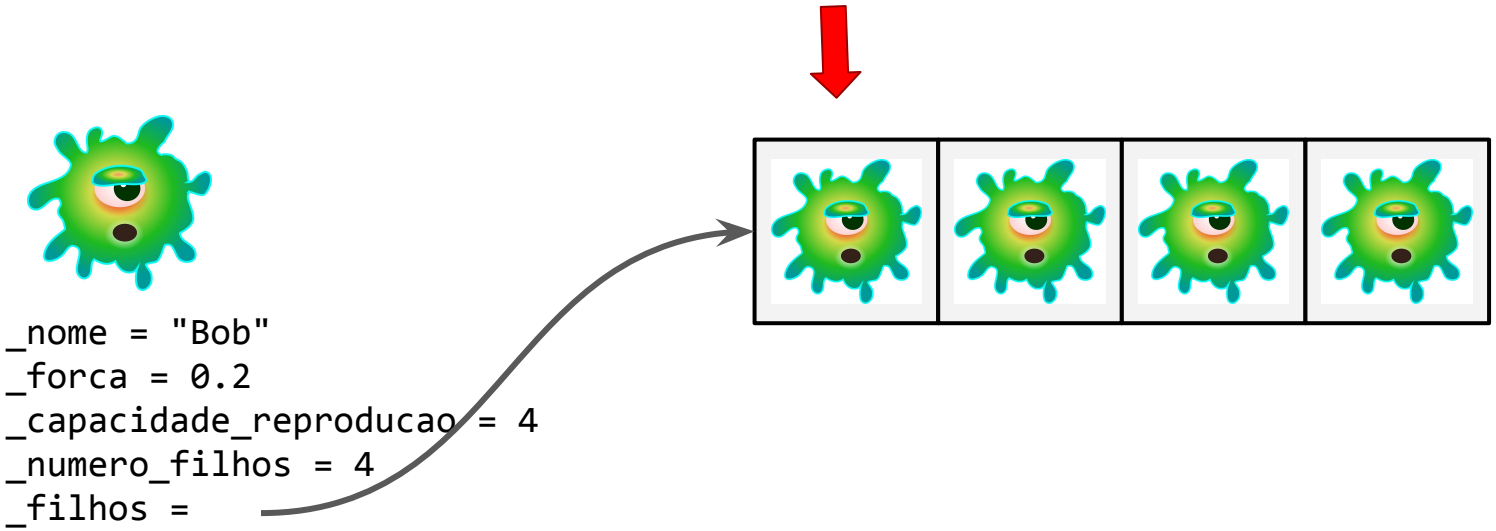
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
        if (_filhos[i] != nullptr)
            delete _filhos[i];
     delete[] _filhos;
  }
}
```



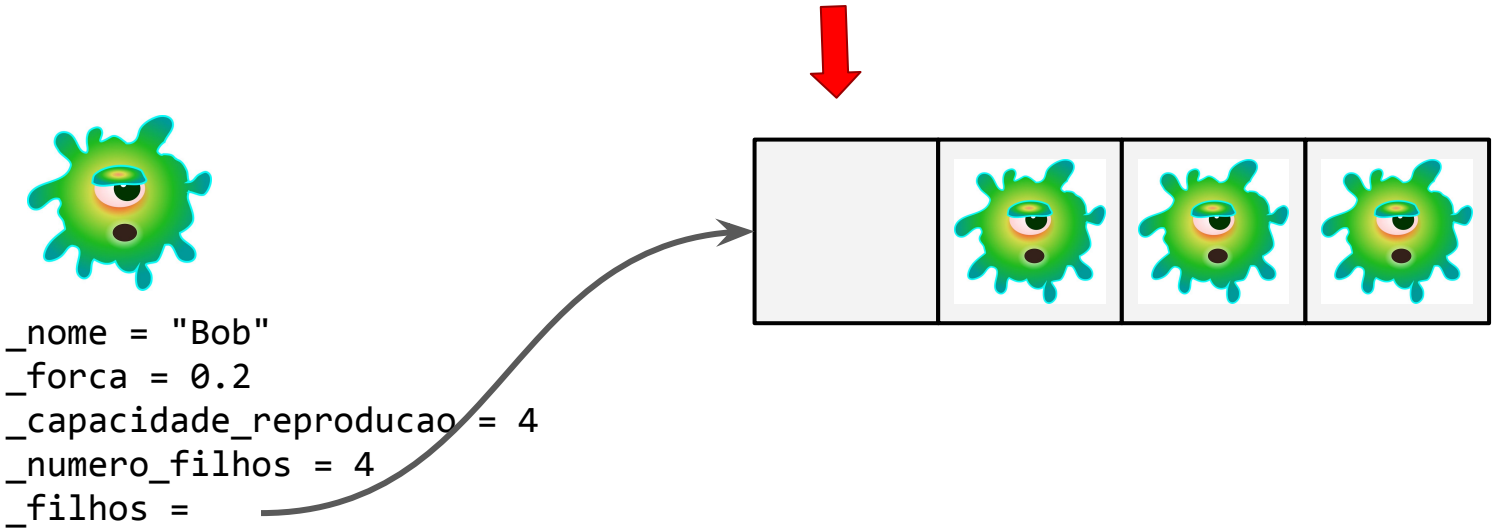
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



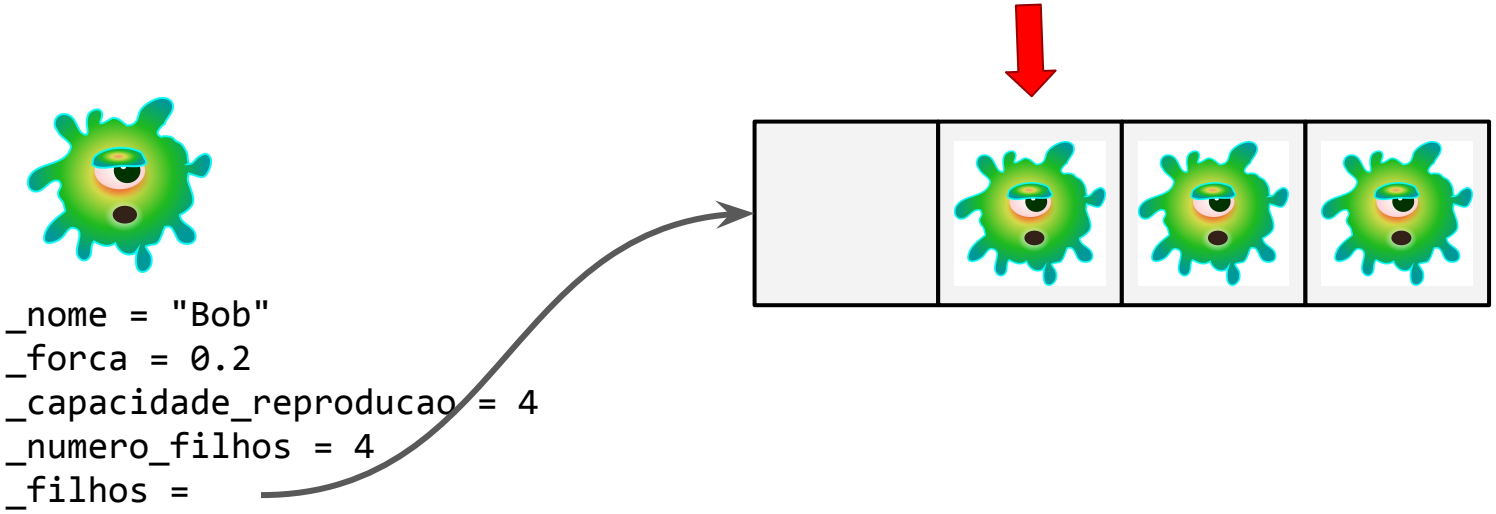
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



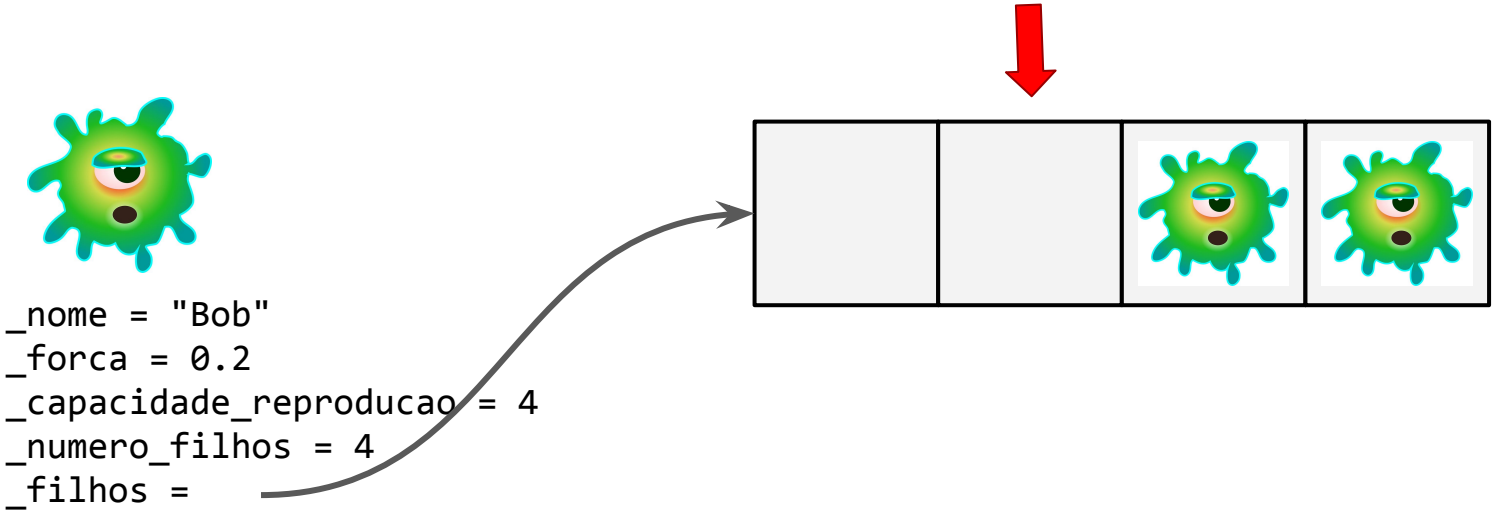
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



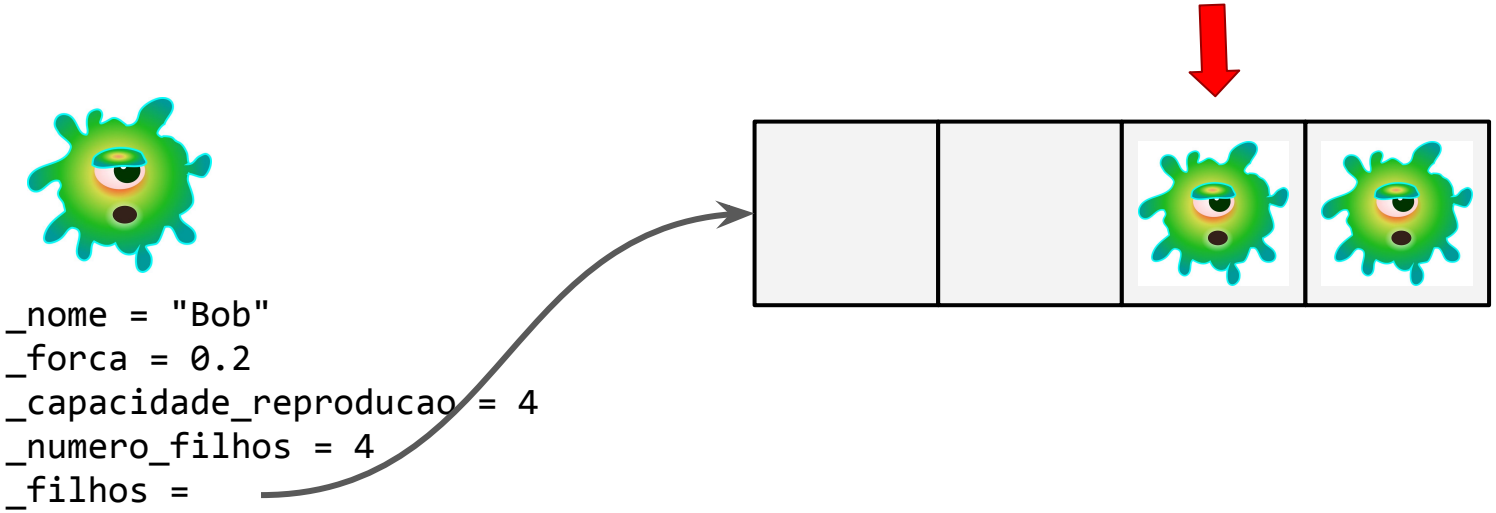
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



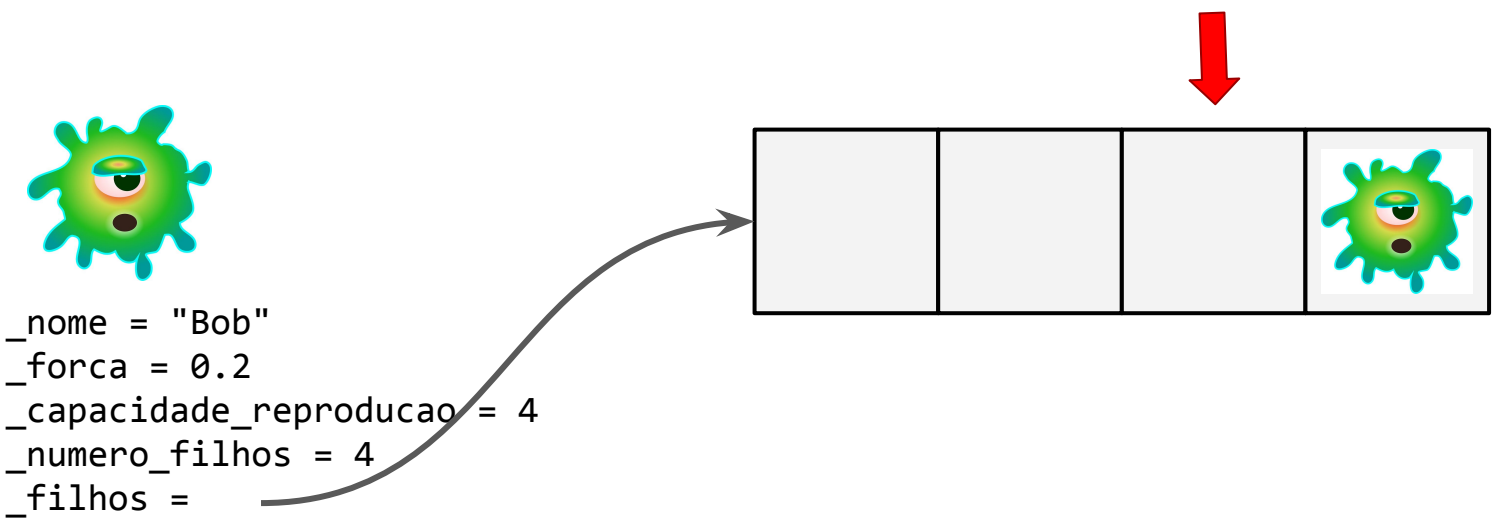
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



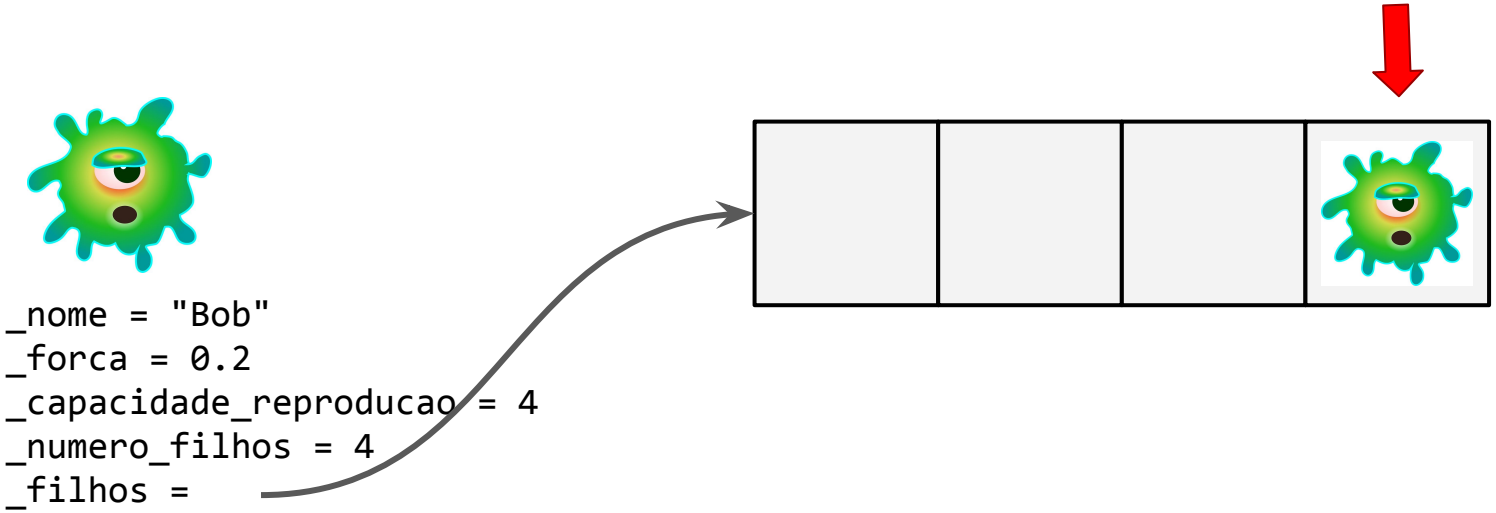
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



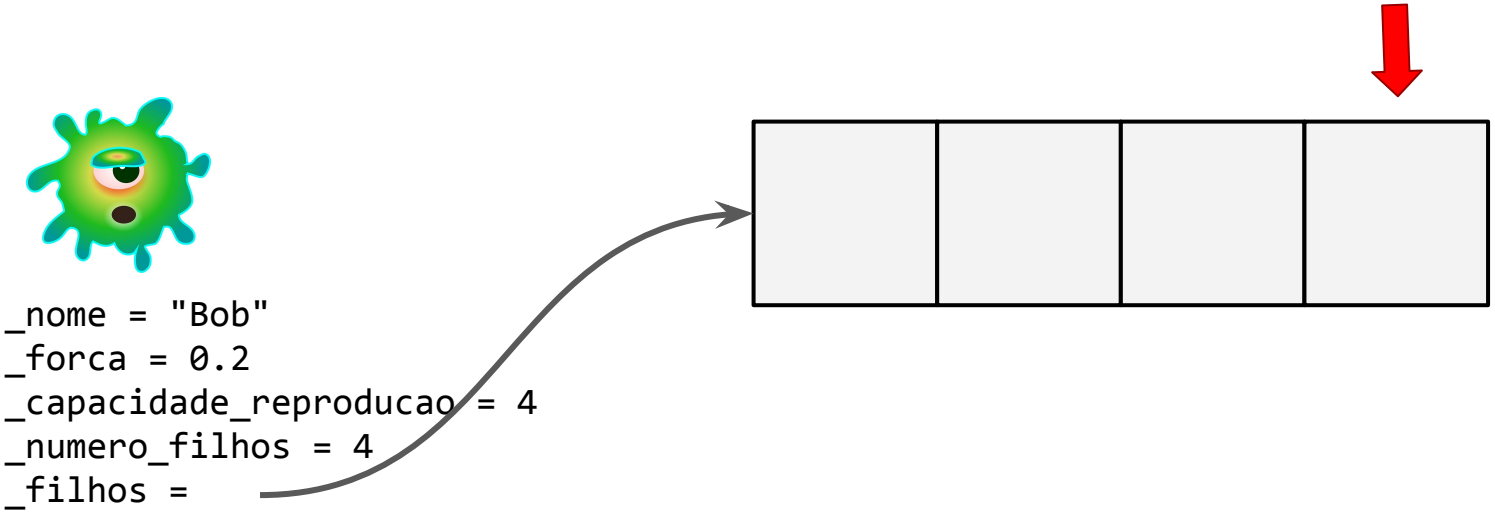
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



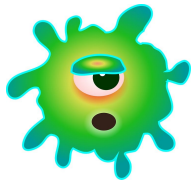
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
        delete _filhos[i];
     delete[] _filhos;
  }
}
```



Destrutor

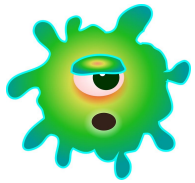
```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
          delete _filhos[i];
     delete[] _filhos;
  }
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 4
_filhos =
```

Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
          delete _filhos[i];
     delete[] _filhos;
  }
}
```

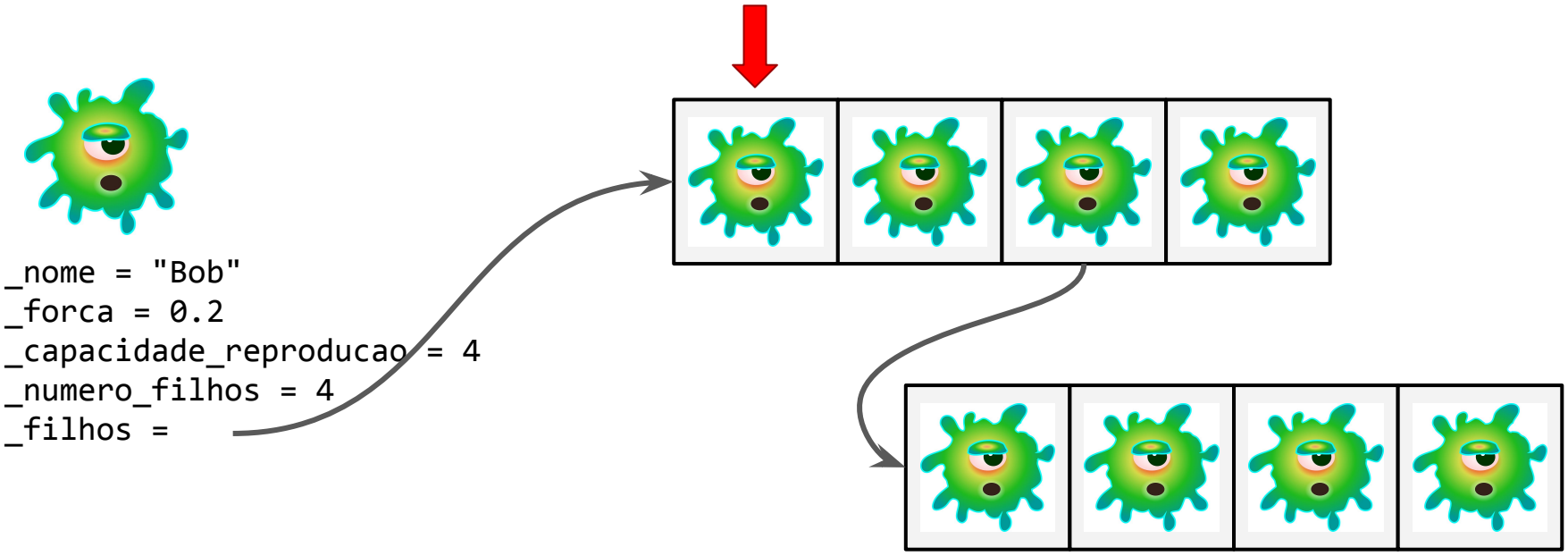


```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 4
_filhos =
```

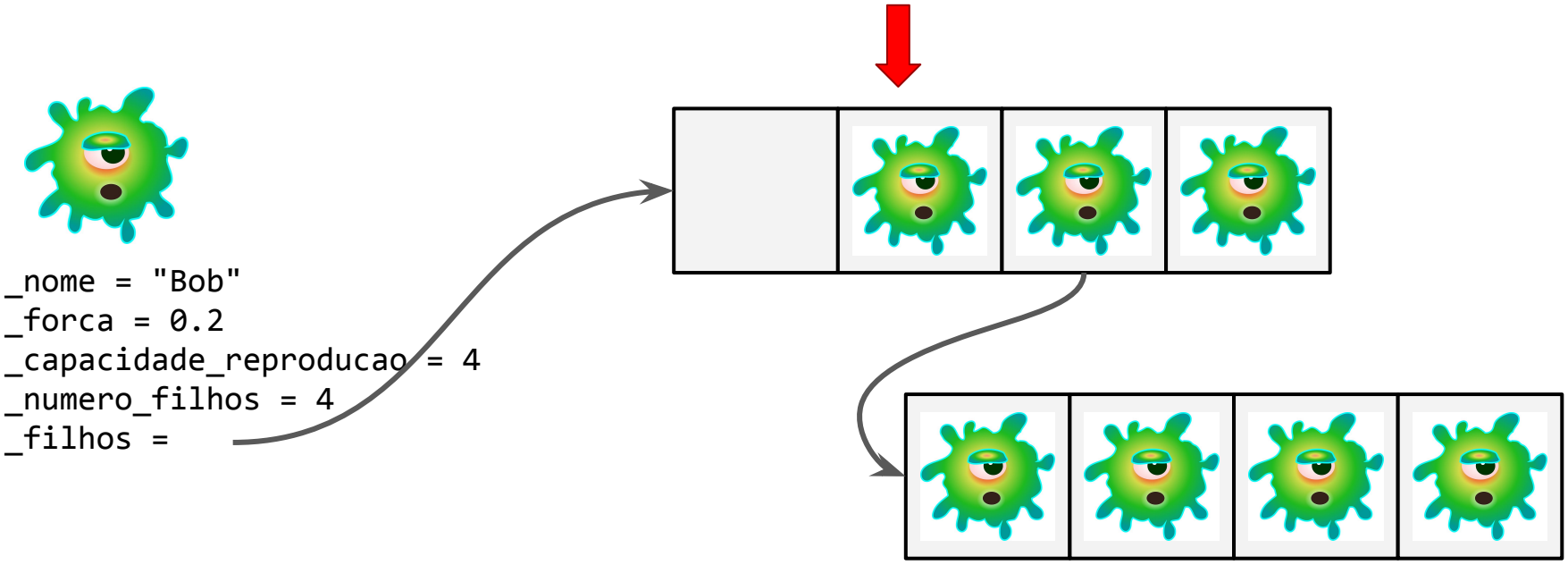
Destrutor

```
Virus::~Virus() {
  if (_filhos != nullptr) {
     for (int i = 0; i < _numero_filhos; i++)
       if (_filhos[i] != nullptr)
          delete _filhos[i];
     delete[] _filhos;
  }
}
```

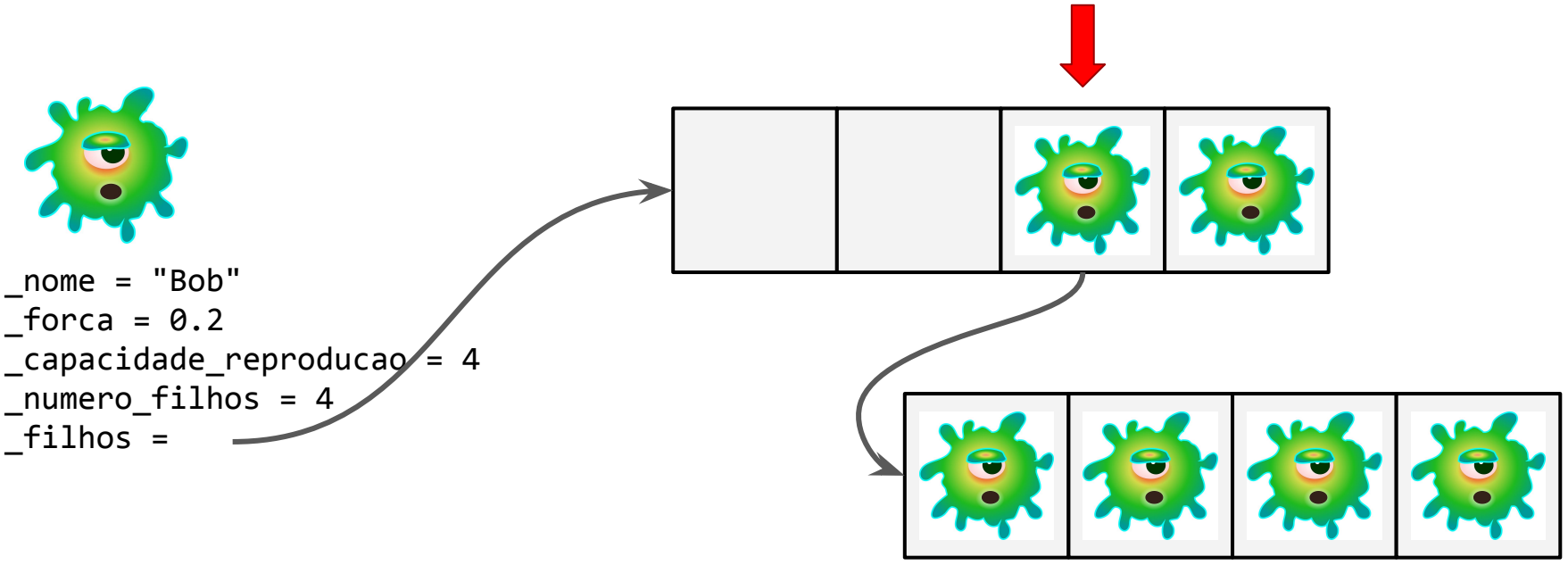
Quando temos uma árvore



Quando temos uma árvore

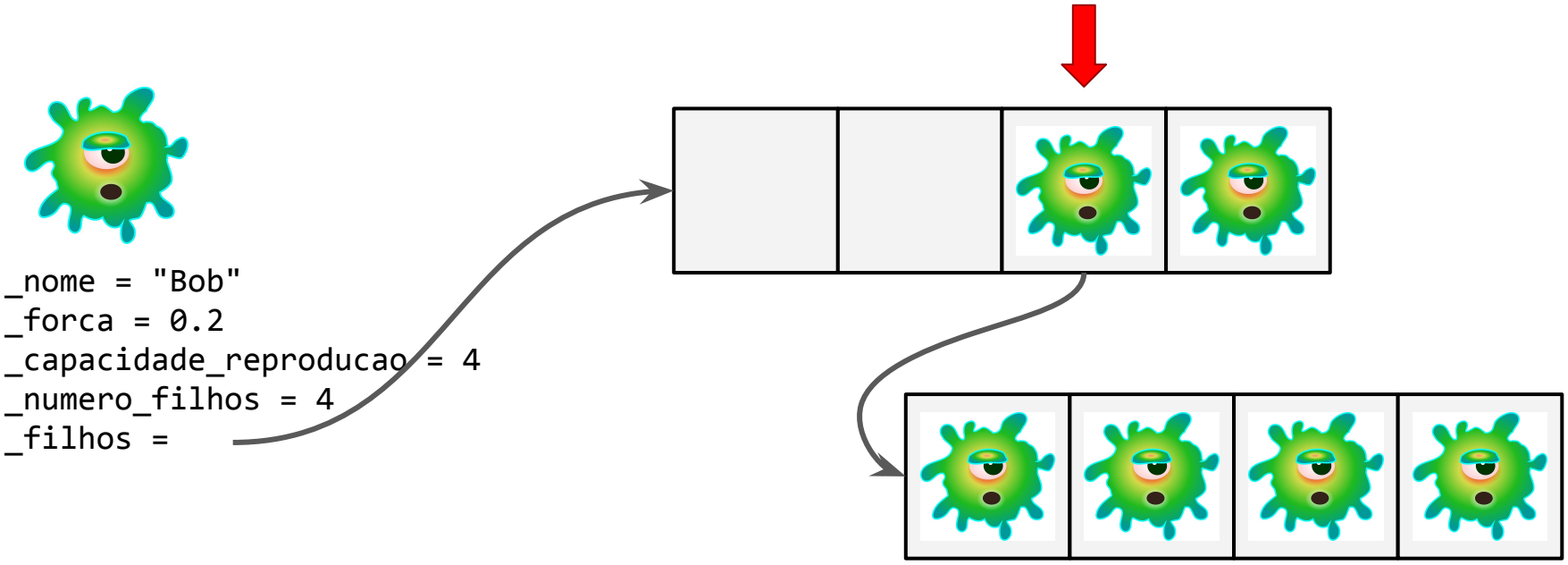


Quando temos uma árvore



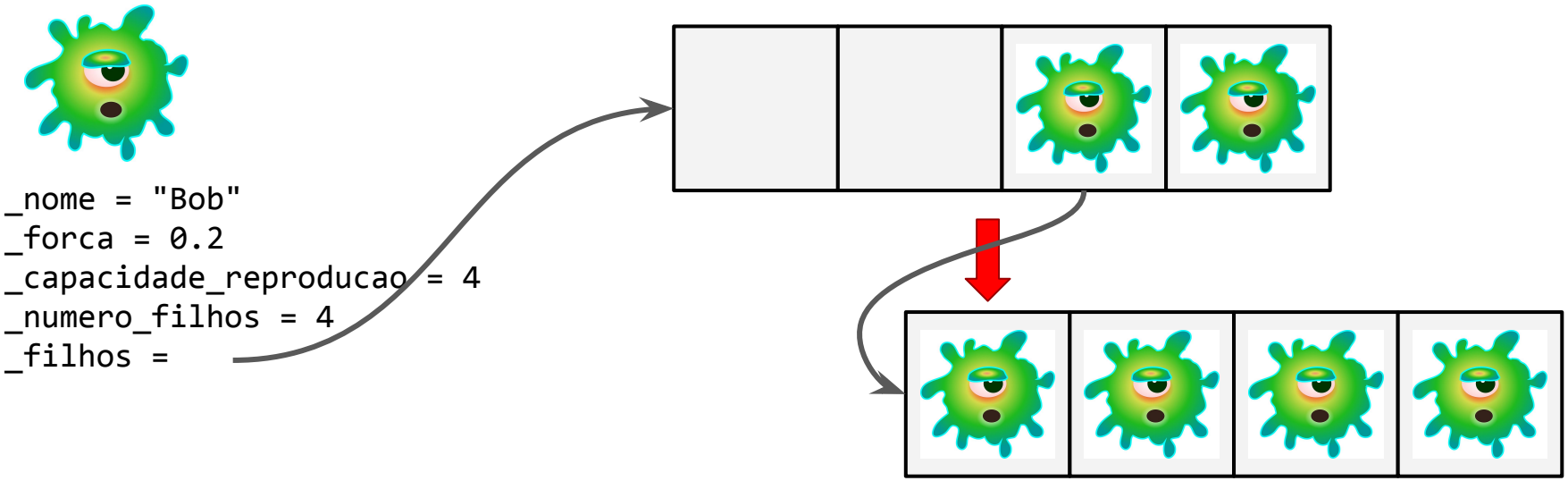
O destrutor do vírus atual...

Também destrói sua lista de filhos



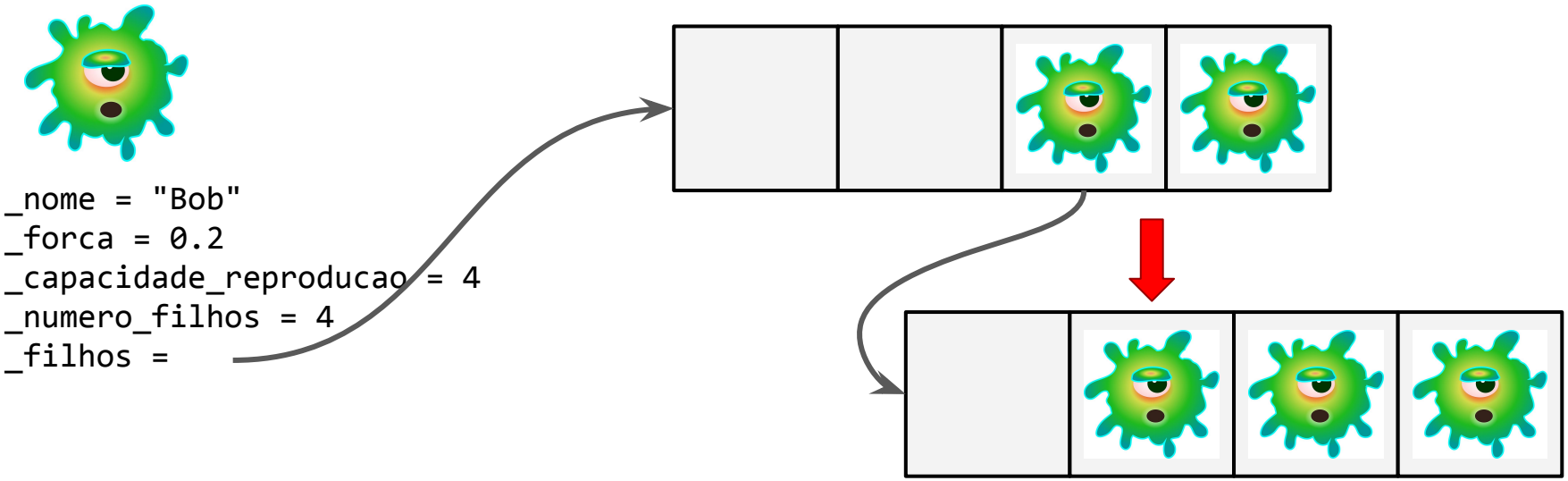
O destrutor do vírus atual...

Também destrói sua lista de filhos



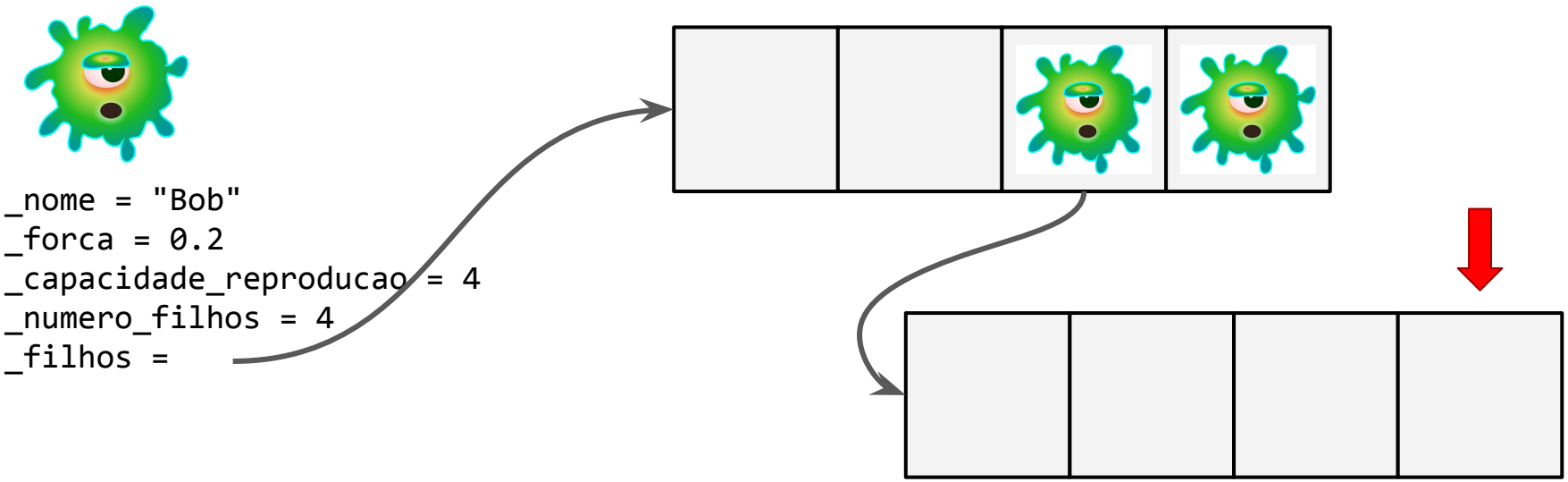
O destrutor do vírus atual...

Também destrói sua lista de filhos



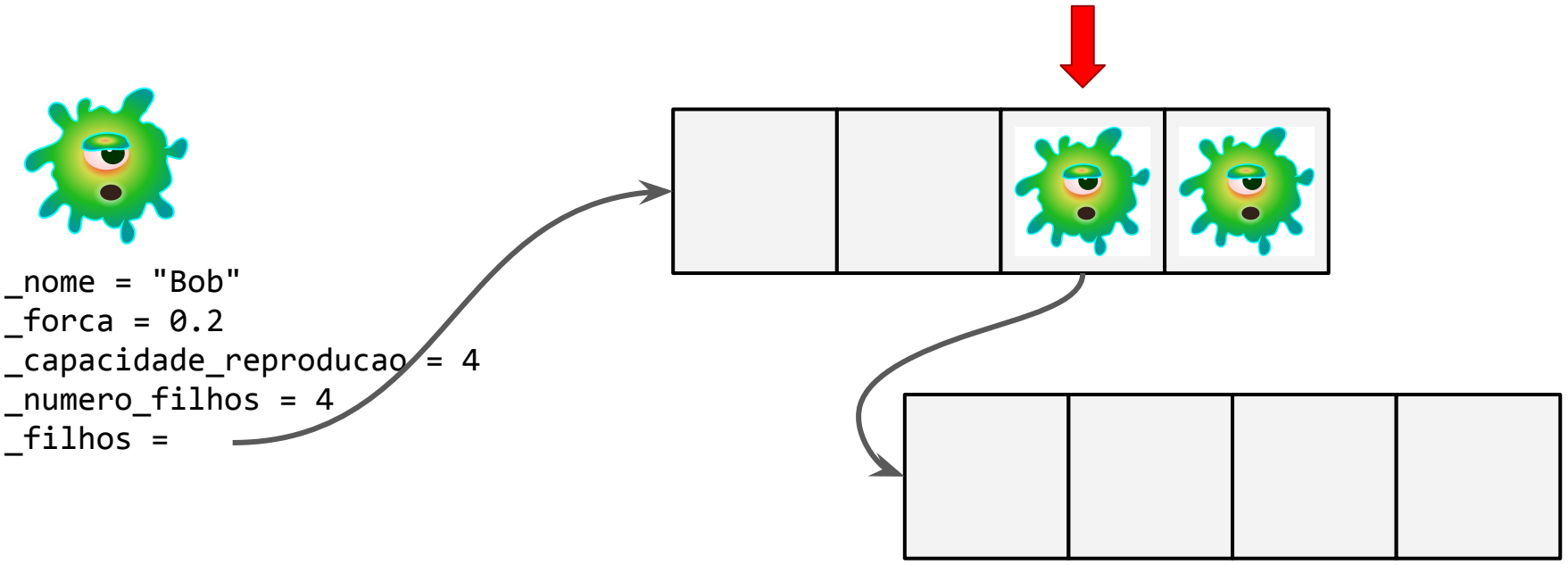
O destrutor do vírus atual...

Também destrói sua lista de filhos



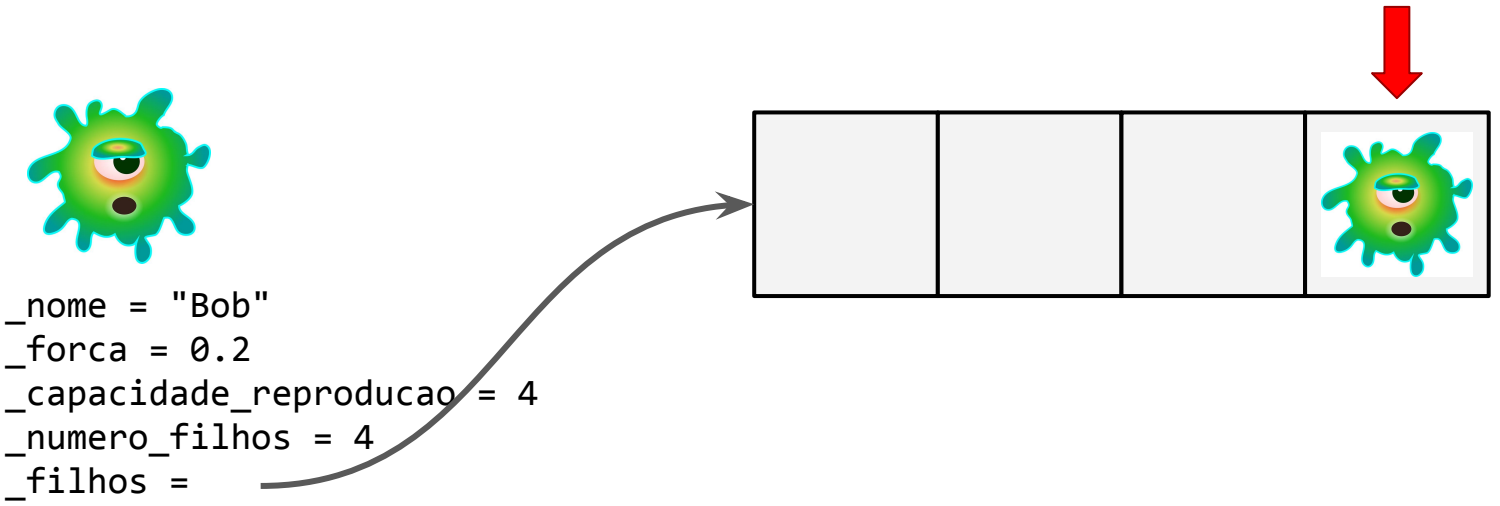
O destrutor do vírus atual...

Também destrói sua lista de filhos



O destrutor do vírus atual...

Também destrói sua lista de filhos



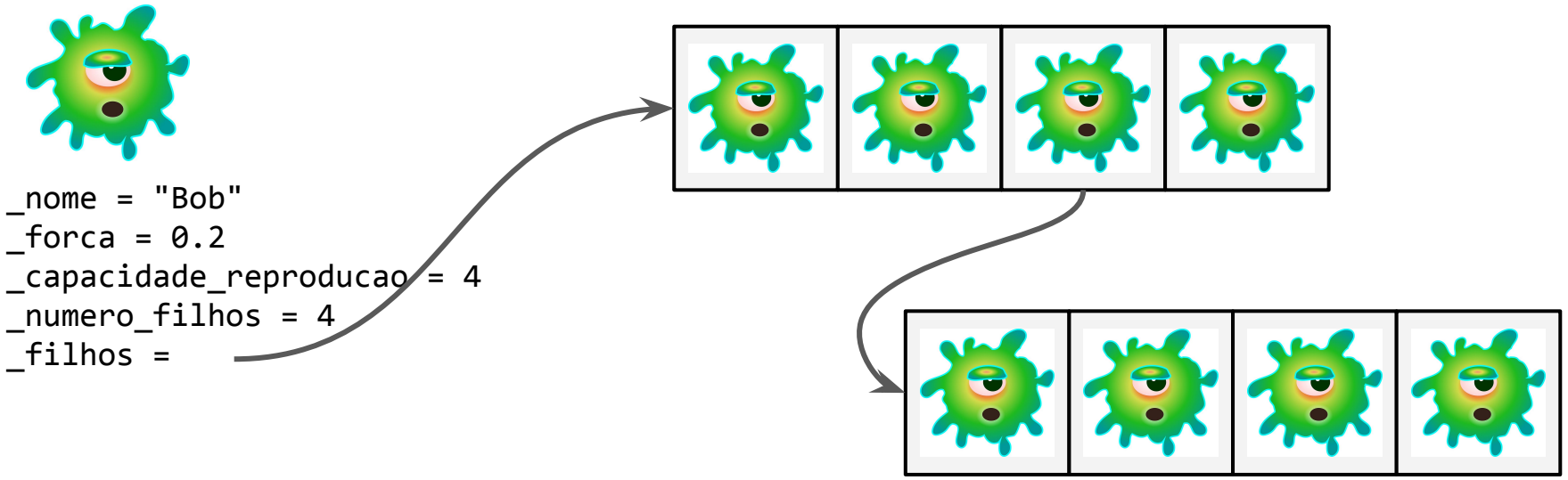
Variáveis Estáticas

Problema

Contar a quantidade total de infecções de todos os vírus

Total é 9 no exemplo abaixo

Bob + 8 descendentes



Classes

Membros estáticos

- Membros de instância
  - Espaço de memória alocado para cada Objeto
  - Chamados somente através do Objeto
- **Membros de classe (estáticos)**
  - Espaço de memória único para todos Objetos
  - Podem ser chamados mesmo sem um Objeto

Problema

Alterando o .h

```
#ifndef PDS2_VIRUS_H
#define PDS2_VIRUS_H

class Virus {
private:
  // Conta quantas infeccções todos os virus já causaram.
  static int _infeccoes_totais;
  // . . .
public:
  // . . .
  // Retorna quantas infeccções todos os virus já causaram.
  static int get_infeccoes_totais();
};


#endif
```

Problema

Alterando o .h

```
#ifndef INF112_VIRUS_H
#define INF112_VIRUS_H

class Virus {
private:
  // Conta quantas infeccções todos os virus já causaram.
  static int _infeccoes_totais;
  // . . .
public:
  // . . .
  // Retorna quantas infeccções todos os virus já causaram.
  static int get_infeccoes_totais();
};


#endif
```

Problema

Alterando o .cpp

```
#include "virus.h"

int Virus::_infeccoes_totais = 0;

Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}

// . . . todo o resto
```

Problema

Note as diferenças

```
#include "virus.h"

int Virus::_infeccoes_totais = 0;

Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}

// . . . todo o resto
```

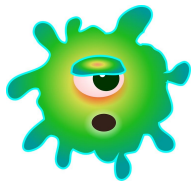
Note que iniciamos fora do construto. Compartilhado!

Incrementamos ao criar novos virus

Problema

Passo a passo

```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



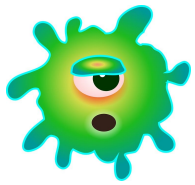
```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 1
_filhos =
```

Virus::_infeccoes_totals = 1;

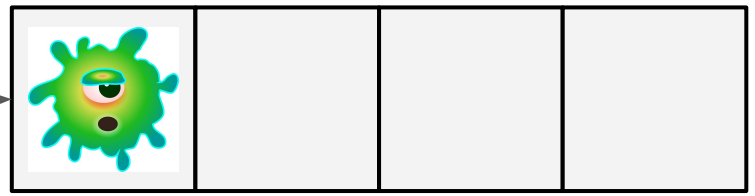
Problema

Lembrando que o reproduzir chama o construtor

```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



```
_nome = "Bob"
_forca = 0.2
_capacidade_reproducao = 4
_numero_filhos = 1
_filhos =
```

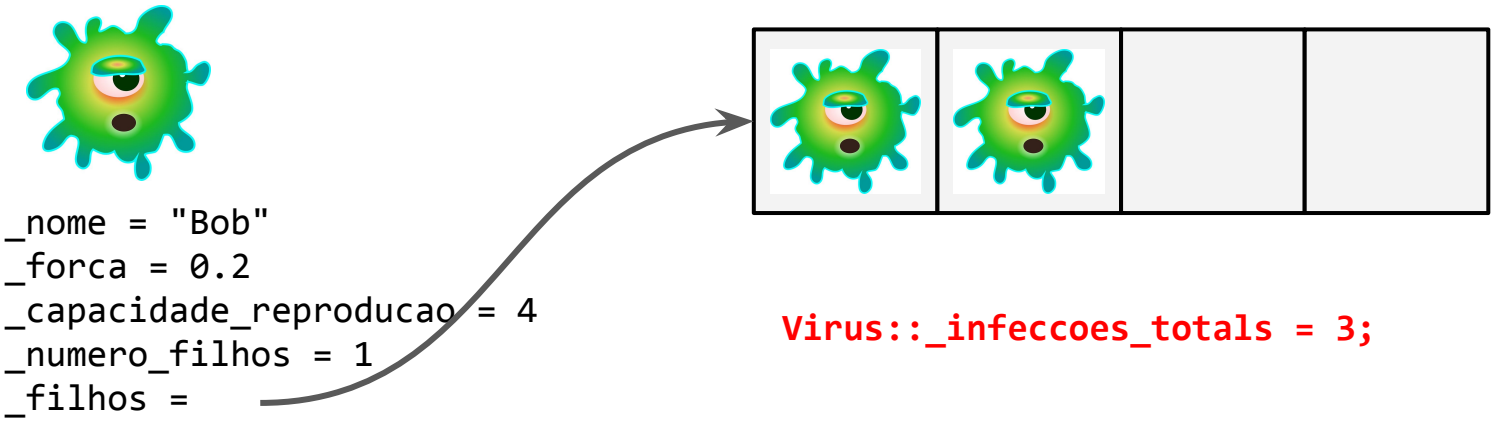


Virus::_infeccoes_totals = 2;

Problema

Cada novo objeto incrementa o contador

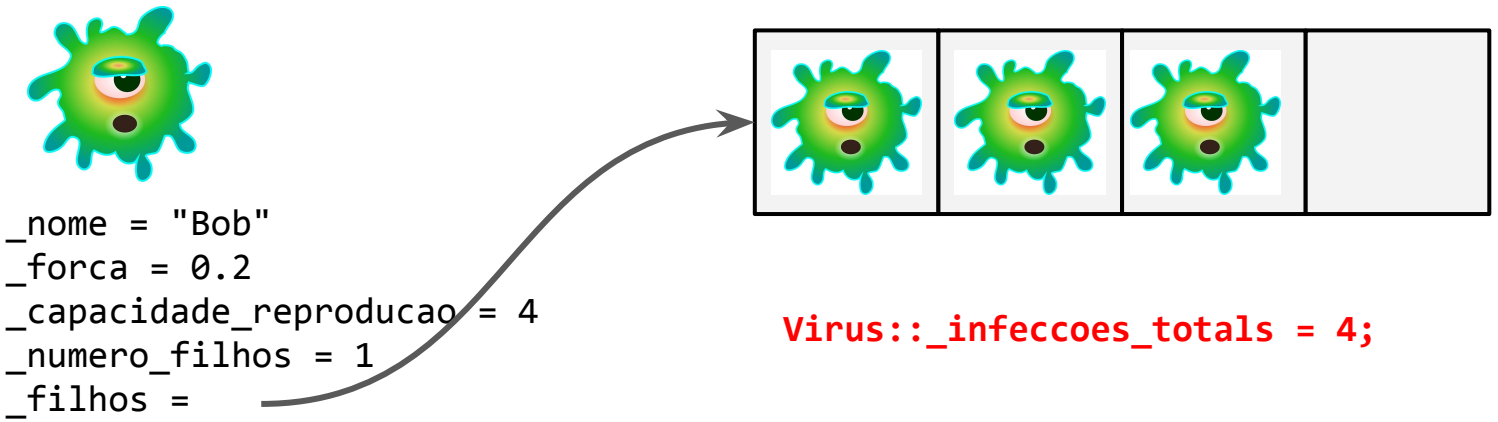
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



Problema

Assim contamos o número total de vírus alocados

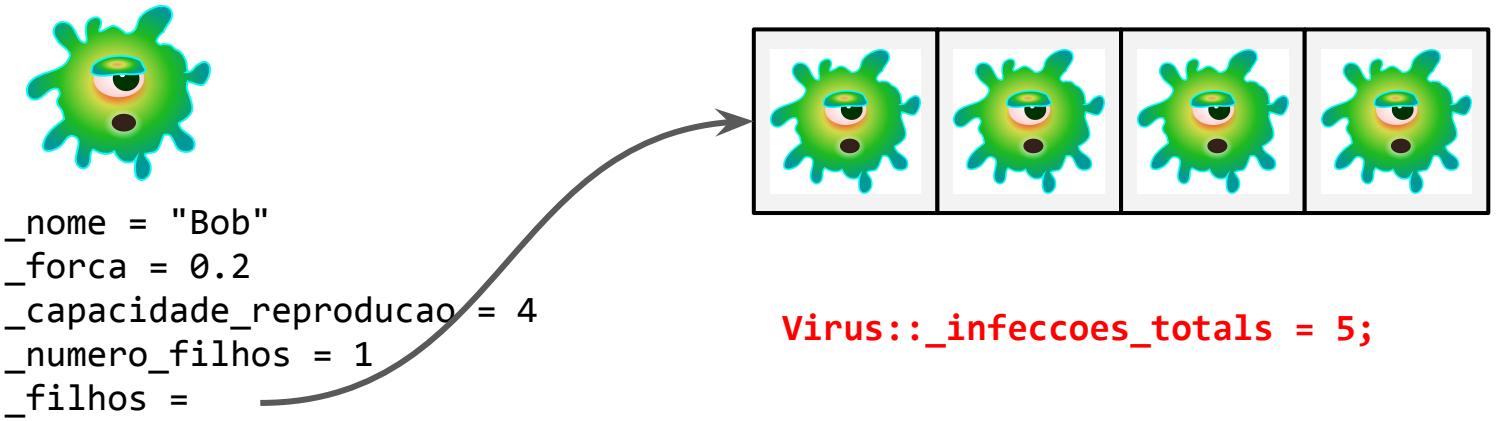
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



Problema

Assim contamos o número total de vírus alocados

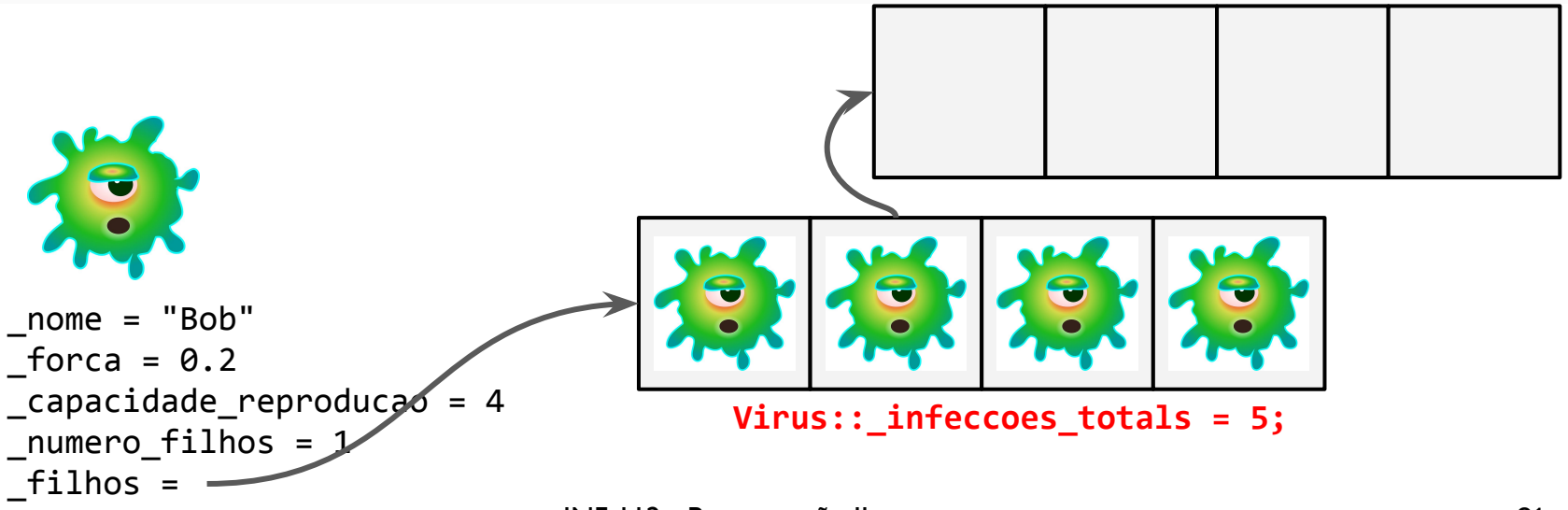
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



Supondo que um dos filhos se reproduza

Como fica o contador?

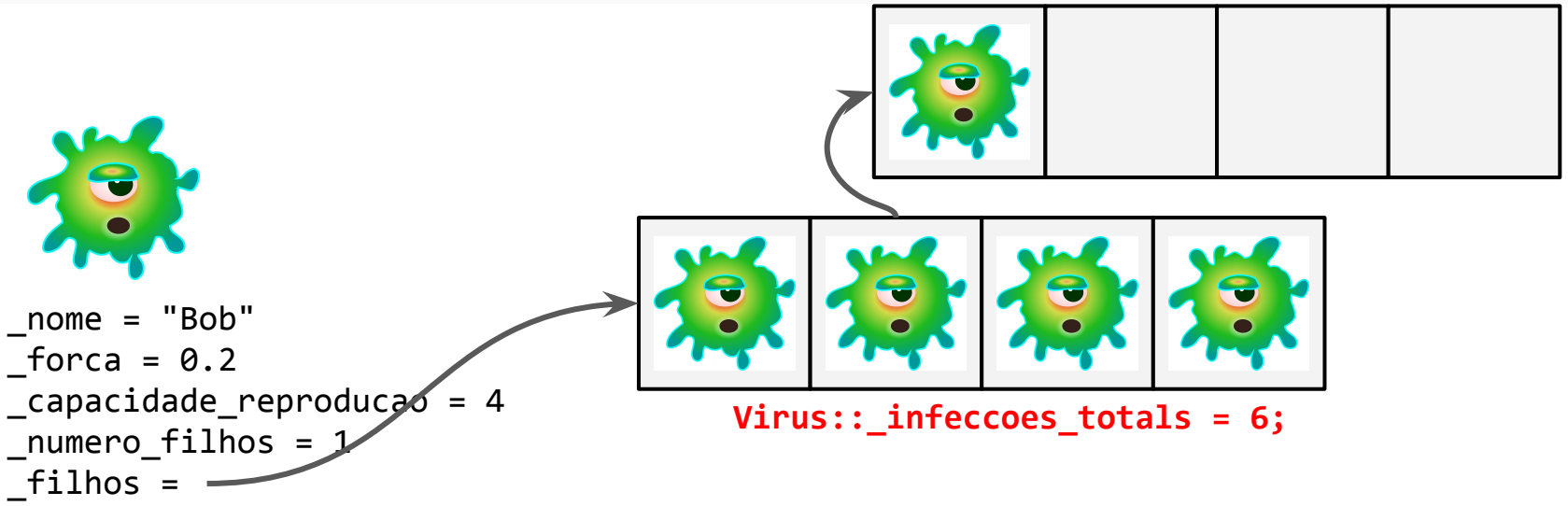
```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



Supondo que um dos filhos se reproduza

Continua ok! **Todos** os objetos **da classe** compartilham **static**

```
Virus::Virus(std::string nome, double forca, int capacidade_reproducao) {
  _nome = nome;
  _forca = forca;
  _capacidade_reproducao = capacidade_reproducao;
  _numero_filhos = 0;
  _filhos = new Virus[capacidade_reproducao]();
  _infeccoes_totais++;
}
```



Acessando o atributo estático

- O atributo pertence e todos os objetos
- Diferentes formas de acesso
- Todas são equivalentes
  - Acessa o mesmo local da memória

Acessando o atributo estático

- Pela classe Virus::

```
int Virus::get_infeccoes_totais() {
  return Virus::_infeccoes_totais;
}
```

- Pelo objeto (com e sem this)

```
int Virus::get_infeccoes_totais() {
  return _infeccoes_totais;
}
```

```
int Virus::get_infeccoes_totais() {
  return this->_infeccoes_totais;
}
```