Encapsulação

Gonzalo Travieso

2020

1 Um exemplo

Para motivar conceitos que serão apresentados em breve, vamos considerar o problema de implementar um biblioteca simples para lidar com números racionais.

Podemos representar números racionais guardando os valores de numerador e denominador desse número:

```
struct Racional {
    int numerador;
    int denominador;
};
```

Esta representação é adequada, mas se desejamos trabalhar com números racionais usando essa representação, devemos considerar que:

- Nem todos os pares de valores de numerador e denominador são válidos; mais explicitamente, o denominador não pode ser zero.
- Para cada número racional, existem diversas representações. Por exemplo, $1/2 = 2/4 = 3/6 = 1024/2048 = -23/-46$. Como os números têm representação limitada no computador, o mais conveniente é usarmos sempre a fração com menores valores possíveis para numerador e denominador, isto é, garantir que não há divisores comuns entre o numerador e o denominador. Também por questão de consistência, queremos representar $-1/2$ e $1/-2$ da mesma forma; para isso, usamos sempre a forma em que o denominador é positivo.

Um número racional representado dessa forma será denominado *normalizado*. Queremos então garantir que os nossos números racionais estejam adequadamente normalizados, depois que for colocado um valor neles ou depois de realizar operações sobre eles. Para facilitar isso, vamos definir uma função para a normalização:

```
// Garante que:
// - r.denominador > 0
// - std::gcd(r.numerador, r.denominador) == 1
void normaliza(Racional &r) {
  // Denominador zero é inválido
  if (r.denominador == 0) {
    std::cerr << "Denominador nulo nao permitido!\n";
```

```
    exit(1);
  }

  // Garante que denominador é positivo
  if (r.denominador < 0) {
    r.numerador = -r.numerador;
    r.denominador = -r.denominador;
  }

  // Elimina fator comum
  auto fator_comum = std::gcd(r.numerador, r.denominador);
  r.numerador /= fator_comum;
  r.denominador /= fator_comum;
}
```

Precisamos também conseguir criar um novo racional devidamente normalizado a partir de valores de numerador e denominador:

```
Racional novo_racional(int numerador, int denominador) {
  Racional resultado{numerador, denominador};
  normaliza(resultado);
  return resultado;
}
```

Podemos então definir um conjunto de operações sobre números racionais:

```
// Calcula a + b
Racional soma(Racional const &a, Racional const &b) {
  // x/y + w/z => (x*z + w*y)/(y*z)
  auto den = a.denominador * b.denominador;
  auto num = a.numerador * b.denominador + b.numerador * a.denominador;

  Racional resultado = novo_racional(num, den);
  return resultado;
}

// Calcula a - b
Racional substracao(Racional const &a, Racional const &b) {
  // x/y - w/z => (x*z - w*y)/(y*z)
  auto den = a.denominador * b.denominador;
  auto num = a.numerador * b.denominador - b.numerador * a.denominador;

  return novo_racional(num, den);
}

// Calcula a * b
Racional produto(Racional const &a, Racional const &b) {
  // x/y * w/z => (x*w)/(y*z)
  auto num = a.numerador * b.numerador;
  auto den = a.denominador * b.denominador;
```

```
  return novo_racional(num, den);
}

// Calcula a / b
Racional divisao(Racional const &a, Racional const &b) {
  // x/y * w/z => (x*w)/(y*z)
  auto num = a.numerador * b.denominador;
  auto den = a.denominador * b.numerador;

  return novo_racional(num, den);
}

// Calcula -a
Racional menos(Racional a) {
    auto num = -a.numerador;
    auto den =  a.denominador;
    return novo_racional(num, den);
}

// Verifica a == b
bool igual(Racional const &a, Racional const &b) {
  return (a.numerador == b.numerador) && (a.denominador == b.denominador);
}
```

Note o seguinte:

- Usamos `novo_racional` sempre que queremos criar um novo valor de número racional a partir de numerador e denominador.
- `novo_racional` chama internamente `normaliza`, garantindo que a representação esteja da forma desejada.
- `igual` pode apenas comparar os valores de numerador e denominador, já que estamos sempre mantendo os números normalizados. Se não fizéssemos isso, seria necessário eliminar fatores comuns antes da comparação. Isto é, o código de `are_equal` tem a pré-condição de que os argumentos são normalizados.
- A normalização dos valores também é importante nas operações aritméticas, visto que algumas delas podem gerar *overflow* se os valores não estiverem normalizados, sendo que a mesma operação poderia funcionar para números normalizados.

2 Encapsulação

Como apresentado acima, o código tem algums problemas, como:

1. O acesso irrestrito aos membros `numerador` e `denominador` da `struct` pode causar problemas.
2. Não existe ligação sintática (na linguagem) entre o tipo de dados `Racional` e as funções que implementam suas operações básicas.

Esse problemas, discutidos melhor a seguir, podem ser resolvidos com o uso de encapsulação e classes.

2.1 Problemas com o acesso irrestrito

O principal problema do código é que se podem acessar os campos `numerador` e `denominador` de um `Racional` em qualquer parte do código, de forma totalmente fora do controle de quem implementou esse tipo de dados. Por exemplo, um código pode fazer:

```
Racional a = novo_racional(2, 3); // a vale 2/3

a.denominador = 0; // E agora?
```

Veja como, apesar do cuidado na implementação em evitar deixar o denominador em zero, isso **não garante** que o denominador de um `Racional` vai realmente ser zero, pois o código que faz uso desse tipo de dados (geralmente denominado *código cliente* do tipo), pode gerar esse problema. Também, apesar de existir uma função para criar racionais adequadamente normalizados, nada impede que o cliente crie um racional não normalizado. Na verdade, é muito natural fazer isso:

```
int a = 1, b = -2;

// Mais tarde no código:
Racional r{a, b}; // Booom!
```

Ou então, veja este outro código:

```
Racional r1 = novo_racional(-2, 3); // r1 = -2/3
Racional r2;
r2.denominador = r1.numerador;
r2.numerador = r1.denominador; // r2 = 1/r1 ?????
Racional r3 = novo_racional(3, -2);
if (!igual(r2, r3)) {
    std::cerr << "BOOOOOOOMMMMMMMMMMM!\n";
}
```

Ao tentar calcular o inverso de um racional, este código quebrou a regra de que o denominador deve ser maior que zero, invalidando a normalização e fazendo com que a função `igual` não funcione mais!

Já que o programador que implementa um tipo de dado e suas operações não tem controle sobre como o usuário vai fazer uso desses dados, não se pode garantir que o código apresentado funcione corretamente.

O problema pode ser expresso mais formalmente usando o conceito de *invariantes*. Do mesmo modo que uma repetição tem um invariante, que é um predicado sempre válido em cada iteração da repetição, um tipo de dados também tem um invariante, que neste caso garante que a representação do dado está adequada. Para o nosso exemplo de racionais, o invariante é aquele indicado em comentários na declaração do tipo:

- O denominador é sempre maior do que zero.

- O numerador e o denominador não têm nenhum divisor comum diferente de 1.

Estas duas condições precisam ser garantidas continuamente para um `Racional`, mas isso não é possível com o código apresentado.

2.2 Problemas com a desconexão entre tipo e funções

As funções apresentadas acima são todas operações básicas relacionadas ao tipo `Racional`, permitindo operações aritméticas, comparação e ajuste de valor. No entanto, não existe nenhuma indicação no código que informe isso. Temos que deduzir o fato analisando o que as funções fazem e lendo os comentários. Isso não é uma situação ideal, pois o nosso código deve ser capaz, na medida do possível, de transmitir as intenções do programador.

Como exemplo do problema, considere a seguinte função:

```
// Computa 1 / (1 + r)
Racional transforma(Racional r) {
    Racional um = novo_racional(1, 1);
    return divisao(um, soma(um, r));
}
```

Note como essa função possui uma *assinatura* (tipos dos argumentos e valor de retorno) idêntica à da função `menos` acima. Mas enquanto `menos` é claramente uma operação básica em racionais (troca de sinal), `transforma` possivelmente é apenas uma função qualquer necessária a um código específico do usuário. Como distinguir esses dois casos?

3 Classes

Os dois problemas apresentados acima podem ser solucionados através do conceito de *encapsulação*, com o uso de *classes*. Os dois princípios básicos são:

- Definimos um tipo de dados e suas operações **simultaneamente**.
- A forma de representação dos dados não é importante para o usuário dos dados (o cliente), e deve permanecer escondida. Os dados só podem ser usados através das **operações** definidas para eles.

O primeiro princípio resolve a desvinculação entre tipos e funções e o segundo princípio garante que não seja possível ao cliente acessar o tipo de dados de forma indisciplinada.

3.1 Definindo uma classe

Para realizar isso, precisamos:

- Usar `class` ao invés de `struct`[1].
- Declarar quais são as funções que fazem parte das operações do tipo.

[1] Mas veja relação entre `struct` e `class` a ser apresentada mais tarde.

- Declarar o que estará disponível ao cliente, diferenciando do que é apenas detalhe de implementação e não pode ser acessado pelo cliente.

Vejamos o caso específico de `Racional`:

```
// VERSÃO 1!
class Racional {
  int _numerador;
  int _denominador;
  void _normaliza();
public:
  inicializa(int = 0, int = 1);
  std::tuple<int, int> numerador_denominador() const;
  Racional soma(Racional const &) const;
  Racional subtracao(Racional const &) const;
  Racional produto(Racional const &) const;
  Racional divisao(Racional const &) const;
  Racional menos() const;
  bool igual(Racional const &) const;
};
```

Essa declaração pode ser entendida da seguinte forma:

- O `class Racional` diz que `Racional` é um novo tipo de dados estruturado. Como usamos `class` ao invés de `struct`, indicamos que estamos preocupados com encapsulação.
- Os membros `_numerador` e `_denominador` são similares aos correspondentes membros da `struct` anterior, mas são considerados *parte da implementação* e **não acessíveis aos clientes**. Estes membros são ditos **privados** da classe.
- As funções `_normaliza`, `soma`, `subtracao`, `produto`, `divisao`, `menos`, `igual`, `numerador_denominador`, e `inicializa`, por serem declaradas dentro da classe, são consideradas operações sobre o tipo de dados. Essas operações têm acesso aos membros privados.
- O rótulo `public:` indica que todas as declarações que o seguem são acessíveis aos clientes. Neste caso, os clientes podem chamar qualquer uma das funções que seguem o rótulo, mas não os membros `_numerador` e `_denominador`, nem a função `_normaliza`.
- Membros como `_numerador` e `_denominador` são usados apenas para guardar dados, e portanto são chamados *membros tipo dados*, ou *campos*.
- Membros como `soma`, `inicializa` ou `_normaliza` são denominados *membros tipo função* ou *métodos*.
- O uso de nomes começando com `_` para membros privados, como `_numerador` ou `_normaliza` é uma convenção minha para deixar claro quais membros são acessíveis aos clientes, não é algo requerido pela linguagem. Outros programadores têm outras convenções.

- O método `numerador_denominador` é necessário neste caso para acessar o valor do numerador e denominador, visto que os membros correspondentes `_numerador` e `_denominador` são privados. Estes métodos permite leitura mas não escrita, portanto o cliente não pode alterar o numerador e o denominador de forma indisciplinada.
- Em relação às funções associadas com a `struct` apresentadas anteriormente, os métodos definidos aqui têm um parâmetro a menos. A razão será explicada em breve.
- A razão para a presença de um `const` após algums dos métodos e não outros tambem será explicada em breve.

Na declaração do tipo acima, fornecemos apenas *protótipos* dos métodos. Precisamos então fornecer também as definições. Isto é feito como no exemplo abaixo:

```
Racional Racional::divisao(Racional const &b) const {
  // x/y * w/z => (x*w)/(y*z)
  auto num = _numerador * b._denominador;
  auto den = _denominador * b._numerador;

  Racional res;
  res.inicializa(num, den)
  return res;
}
```

Para entender essa definição:

- O nome da função agora é `Racional::divisao`, o que significa que estamos definindo o método `divisao` da classe `Racional`.
- Existem dois tipos de acesso a membro nesse código:
  - Acessos como `b._numerador`, similar ao que realizamos no código de `struct` anterior. Neste caso, queremos o membro `_numerador` do objeto `b`.
  - Acessos como `_numerador`, sem um operador de acesso a membro `.` anterior. Esta é a convenção para dizer que estamos acessando o membro `_numerador` de um objeto especial, que é aquele sobre o qual a chamada do método `divisao` foi feita, como ficará claro a seguir.
- O `const` após a lista de parâmetros da função indica que este método não altera o objeto sobre o qual ele é chamado (neste caso, nem `_numerador` nem `_denominador` têm seu valor alterado).

Para contrastar, vejamos como fica a definição do método `inicializa`:

```
void Racional::inicializa(int numerador, int denominador) {
    _numerador = numerador;
    _denominador = denominador;
    _normaliza();
}
```

Neste código, alteramos o valor de `_numerador` e `_denominador`, e portanto o método não é `const`. Também, ao chamar o método `_normaliza`, ele será chamado sobre o mesmo objeto para o qual `inicializa` foi chamado.

Os outros métodos devem ser alterados de forma similar.

Vejamos agora alguns exemplos de uso.

```
Racional a, b, c;
a.inicializa(2, 5);
b.inicializa(3, 1);
c = a.divisao(b);
```

Note a nova sintaxe de chamada de método: à esquerda do método, precisamos colocar um objeto do tipo `Racional`, em seguida o nome do método seguido por seus argumentos. Então, quando o compilador vê a chamada

```
a.inicializa(2, 5);
```

ele verifica que `a` é do tipo `Racional`, e que portanto ele precisa chamar o método `Racional::inicializa(int, int)`, passando 2 e 5 como valores iniciais para os parâmetros `numerador` e `denominador` desse método. Como o objeto à esquerda do método é o objeto `a`, então durante a execução dessa chamada de `inicializa`, `_numerador` se refere a `a._numerador` e `_denominador` se refere a `a._denominador`. Também a chamada `_normaliza()` dentro do método `inicializa` será entendida como `a._normaliza()`, e os membros `_numerador` e `_denominador` nomeados no código de `_normaliza` se referirão aos correspondentes campos do objeto `a`. Ocorre algo similar na chamada `a.divisao(b)`, e aqui podemos ver o significado do `const` em `divisao`: ele indica que essa chamada não irá alterar o valor do objeto `a`; obviamente o objeto `b` também não é alterado, pois ele é passado por referência constante.

3.2 O que conseguimos

Com o uso de encapsulação, garantimos que o código cliente não pode alterar diretamente os membros que realizam a implementação do tipo. Por exemplo, um código como o que segue é inválido:

```
Racional r;
r._numerador = 3;   // ERRO! Tentativa de acesso a membro privado
r._denominador = 7; // ERRO! Tentativa de acesso a membro privado
```

Portanto, o ajuste de valor de um `Racional` tem que passar pelo método `inicializa`, que garante que o valor esteja representado do modo que desejamos (com denominador positivo e sem fatores comuns), isto é, mantendo o **invariante** do tipo. Como todas as operações que implementamos também garantem isso, então sabemos que não é possível existir um `Racional` que não mantenha o invariante desejado.[2] Isso significa que, ao realizarmos as diversas operações (`soma`, `produto`, etc.), sabemos que os parâmetros usados mantêm o invariante, e podemos levar isso em consideração no código, por exemplo como fizemos no código de `igual`.

Garantir a manutenção do invariante do tipo é a principal razão para o uso de encapsulação.

[2] Isso ainda não é totalmente verdade, pois temos um problema restante a resolver. Veja na próxima seção.

3.3 O problema da inicialização e construtores

Infelizmente, o código anterior ainda não garante o invariante do tipo `Racional` em todos os casos. Por exemplo, no código seguinte:

```
Racional a, b, c;
a.inicializa(1, 2);
c = a.produto(b);
```

Como não foi feito um `inicializa` para o objeto `b`, ele pode ter qualquer valor em seus membros `_numerador` e `_denominador`, por exemplo, um `_denominador` zero, o que causa problemas depois nos cálculos seguintes.

3.3.1 Construtores

Para realmente garantirmos que um objeto de um certo tipo sempre terá seu invariante válido, **devemos garantir esse invariante já na criação do objeto**! Para isso usamos o que é denominado um **construtor**. Um construtor é um método da classe que é automaticamente chamado para um objeto da classe quanto ele é criado. Para definir um construtor, basta definir um método **com o mesmo nome da classe** e **sem valor de retorno** (isto é, não se especifica nem `void`). Para a nossa classe `Racional` fazemos:

```
// VERSÃO 2!
class Racional {
  int _numerador;
  int _denominador;
  void _normaliza();
public:
  Racional(int = 0, int = 1);
  std::tuple<int, int> numerador_denominador() const;
  Racional soma(Racional const &) const;
  Racional subtracao(Racional const &) const;
  Racional produto(Racional const &) const;
  Racional divisao(Racional const &) const;
  Racional menos() const;
  bool igual(Racional const &) const;
};
```

E então:

```
Racional::Racional(int numerador, int denominador) {
    _numerador = numerador;
    _denominador = denominador;
    _normaliza();
}
```

Estabelecemos valores *default* para o numerador e o denominador, de modo que se eles não forem especificados, o racional é inicializado com 0 e se for especificado apenas um valor n, o racional é inicializado como $n/1$. Os parâmetros para o construtor são passados na hora em que um objeto é criado:

```
Racional a{2, 3}; // Cria a._numerador = 2; a._denominador = 3;
Racional b{9}; // Cria b._numerador = 9; b._denominador = 1;
Racional c; // Cria c._numerador = 0; c._denominador = 1;
c = Racional{7, 2}; // Agora c._numerador = 7; c._denominador = 2;
```

Repare que eu retirei o método `inicializa` da interface, visto que podemos usar o construtor para fazer a mesma função de um modo diferente, como se mostra na última linha do exemplo acima.

No caso geral, o construtor deve receber os parâmetros necessários para calcular os valores iniciais dos membros de dados do objeto. Isso não implica necessáriamente, como neste caso, receber um valor para cada um dos membros.

Se a inicialização *default* dos membros já é suficiente para garantir o invariante do objeto, então não é necessário fornecer um construtor, apesar de que você pode querer fornecer um para permitir inicialização com valores diferentes dos *default* ao mesmo tempo garantindo o invariante do tipo.

3.3.2 Inicialização de membros

No código do construtor apresentado acima, estamos **atribuindo** os valores passados de numerador e denominador para os membros correspondente. Uma outra opção é fazermos a **inicialização** desses membros com os valores passados. Isso pode ser feito através da sintaxe de **lista de inicialização de membros**, exemplificada no código abaixo:

```
Racional::Racional(int numerador, int denominador)
    : _numerador(numerador), _denominador(denominador)
{
    _normaliza();
}
```

Aqui os membros são inicializados com os valores fornecidos, e então o código dentro das chaves é executado. A ordem dos membros na lista de inicialização de membros deve ser **a mesma da ordem de declaração** dos membros na classe.

Neste exemplo, não há muita diferença entre usar inicialização ou atribuição como no código anterior, mas a inicialização é necessária em alguns casos, como membros constantes ou referências, ou passagem de parâmetros para construtores de membros.

3.4 Métodos *inline*

O uso de encapsulação tende a necessitar um conjunto de métodos triviais (de código bem simples). Podemos nestes casos realizar a implementação dos métodos junto com a sua declaração, o que indicará ao compilador que esses métodos podem ser expandidos *inline* no ponto de chamada, ao invés de gerar uma chamada de função e retorno. Isto só deve ser feito para códigos simples. No nosso exemplo, podemos fazer:

```
// VERSÃO 3!
class Racional {
  int _numerador;
  int _denominador;
```

```
  void _normaliza();

public:
  Racional(int numerador = 0, int denominador = 1)
      : _numerador(numerador), _denominador(denominador) {
    _normaliza();
  }

  std::tuple<int, int> numerador_denominador() const {
    return {_numerador, _denominador};
  }

  Racional soma(Racional const &) const;
  Racional subtracao(Racional const &) const;
  Racional produto(Racional const &) const;
  Racional divisao(Racional const &) const;
  Racional menos() const { return {-_numerador, _denominador}; }
  bool igual(Racional const &b) const {
    return (_numerador == b._numerador) && (_denominador == b._denominador);
  }
};
```

O restante dos métodos fica implementado como anteriormente.

3.5 Funções amigas

O exemplo anterior apresenta a forma mais tradicional de uso de encapsulação. Entretanto, para o nosso caso especial, ela tem um problema que vamos discutir agora.

Quando pensamos em uma operação aritmética sobre números, por exemplo uma soma $a + b$, uma característica dessa operação é que os argumentos (operandos) a e b são *simétricos*, isto é, atribuimos a mesma importância para os dois. Mesmo quando a operação em si não é simétrica, como na subtração $a - b$, não damos uma interpretação diferente para um operando ou outro. Já no nosso código isso não acontece. Se vamos somar dois racionais `a` e `b`, fazemos `a.soma(b)`, o que tem a seguinte interpretação: veja qual o tipo do objeto `a`, na classe desse objeto, procure um método chamado `soma` que aceite como parâmetro um objeto do tipo `b` e se existir, chame esse método sobre o objeto `a` passando como parâmetro o objeto `b`. Veja como temos uma assimetria: o método a executar é procurado em `a`, mas não em `b`. Veremos mais para frente um dos impactos negativos práticos dessa assimetria.

Podemos resolver esse problema usando o conceito de *funções amigas*: ao invés de declarar as operações como *métodos* da classe, as declaramos como *funções* normais, mas que terão acesso aos membros privados da classe (porisso são denominadas *amigas*). No nosso exemplo:

```
// VERSÃO 4. Ainda não é a final!
class Racional {
  int _numerador;
  int _denominador;
  void _normaliza();
```

```
public:
  Racional(int numerador = 0, int denominador = 1)
      : _numerador(numerador), _denominador(denominador) {
    _normaliza();
  }

  std::tuple<int, int> numerador_denominador() const {
    return {_numerador, _denominador};
  }

  friend Racional soma(Racional const &, Racional const &);
  friend Racional subtracao(Racional const &, Racional const &);
  friend Racional produto(Racional const &, Racional const &);
  friend Racional divisao(Racional const &, Racional const &);
  friend Racional menos(Racional const &a) {
    return {-a._numerador, a._denominador};
  }
  friend bool igual(Racional const &a, Racional const &b) {
    return (a._numerador == b._numerador) && (a._denominador == b._denominador);
  }
};
};
```

Uma função é amiga se:

- Tem antes de seu protótipo a palavra-chave `friend`.
- É declarada dentro da declaração da classe.

Como as funções amigas não são métodos, elas precisam receber explicitamente todos os parâmetros (um método recebe acesso implícito ao objeto sobre o qual é chamado), e portanto precisamos acrescentar um parâmetro em cada função. Também não faz sentido falar em `const` para essas funções, visto que elas não são métodos atuando sobre um objeto (o caráter constante ou não é determinado pelo tipo do parâmetro).

A implementação das funções é como a de qualquer outra função. Por exemplo, a função `produto` seria:

```
Racional produto(Racional const &a, Racional const &b) {
  // x/y * w/z => (x*w)/(y*z)
  auto num = a._numerador * b._numerador;
  auto den = a._denominador * b._denominador;

  return {num, den};
}
```

A palavra-chave `friend` não aparece aqui, pois a função já foi declarada amiga quando seu protótipo foi apresentado. Por ser amiga, códigos que acessam membros privados, como `a._numerador` e `a._normalize()` são permitidos.

Com essas definições, podemos agora fazer operações aritméticas da seguinte forma:

```
Racional a{2, 3}, b{1, 7}, c;
c = soma(a, produto(a, b));
```

A vantagem, como discutido acima, é que agora os parâmetros são simétricos, como `a` e `b` na chamada `produto` acima. Veremos a importância disso (além da estética) mais tarde.

IMPORTANTE: O uso de `friend` tem a intenção de permitir uma sintaxe mais apropriada a certas operações sobre o tipo que estamos definindo. Portanto somente podem ser declaradas como `friend` funções que são **operações básicas** do tipo de dados sendo definido. Em qualquer hipótese, é um erro incluir uma função como amiga de uma classe apenas para possibilitar acesso aos membros privados. *Isto seria contrário a todo o objetivo de se usar encapsulação.* Se você precisa da encapsulação você não deve criar um buraco nela declarado funções que não são operações básicas do tipo como `friend`; se você não precisa de encapsulação, então para que usá-la? Uma `struct` basta.

4 Quando usar encapsulação

Vimos nos exemplos acima como usar encapsulação em C++ e suas vantagens. Para deixar isso mais claro, podemos estabelecer a seguinte regra para a decisão entre o uso de `struct` e `class`:

> Use encapsulação (e portanto `class`), se o tipo que você está definindo tem um invariante não-trivial que precisa ser mantido. Se o tipo não tem invariante ou o invariante é trivial, use `struct`.

Dizemos que um *invariante* é *trivial* se ele é garantido pelos tipos de dados dos membros. Por exemplo, vamos supor que queremos representar pontos no espaço bidimensional sem limites. Neste caso, podemos representar o pontos pelas suas coordenadas x e y, e definir uma `struct`:

```
struct Ponto {
    double x, y;
}
```

O invariante seria que `x` e `y` têm que representar o número de unidades de distância da origem, com o sinal apropriado; como não temos limites na região do espaço, qualquer valor é válido, desde que seja um número real. Mas isso é garantido por termos definido `x` e `y` como `double`, que representa um subconjunto dos números reais.[3] Mas note que isso só é válido porque o ponto pode estar em qualquer lugar do plano. Se em nosso problema o ponto tem que ficar em uma região restrita, digamos dentro de um quadrado $[0,1]\times[0,1]$, então necessitamos encapsulação para garantir esse invariante.

[3] Na verdade é mais complicado do que isso, pois um `double` pode guardar valores que não são reais, como `NaN` ou `inf`. Estamos desconsiderando essas complicações aqui.