Gerenciamento de Recursos

Gonzalo Travieso

2019

1 Apresentação do problema

Ocorre frequentemente que objetos de certos tipos necessitem armazenar recursos do sistema, que eles precisam gerenciar. Isso pode gerar problemas no código, se não forem tomados alguns cuidados na implementação da classe do objeto.

Para exemplificar os problemas, vamos considerar um exemplo artificial onde eles são facilmente mostrados. Suponha que numa empresa temos um sistema que registra alguns dados sobre os seus colaboradores, como nome, função e salário. Mas alguns colaboradores têm adicionalmente um cargo comissionado (por exemplo, um chefe de grupo), que representa um valor adicional ao seu salário básico. Vamos tentar uma primeira implementação:

```
#include <string>

class Colaborador {
    struct Comissionado {
        std::string cargo;
        double comissao;
    };

    std::string _nome;
    std::string _funcao;
    double _salario;
    Comissionado *_com_ptr{nullptr};

public:
    Colaborador() = default;

    Colaborador(std::string nome, std::string funcao, double salario)
        : _nome(nome), _funcao(funcao), _salario(salario) {}

    std::string nome() const { return _nome; }
    std::string funcao() const {
        auto res{_funcao};
        if (_com_ptr) {
            res += " [" + _com_ptr->cargo + "]";
        }
        return res;
```

```
    }
    double a_pagar() const {
        auto res{_salario};
        if (_com_ptr) {
            res += _com_ptr->comissao;
        }
        return res;
    }
    void deixa_comissao() {
        if (_com_ptr) {
            delete _com_ptr;
        }
        _com_ptr = nullptr;
    }
    void recebe_comissao(std::string cargo, double comissao) {
        deixa_comissao();
        _com_ptr = new Comissionado{cargo, comissao};
    }
};
```

Neste código, o fato de uma pessoa poder ter ou não um cargo comissionado é representado pelo uso de um ponteiro para uma `struct` com os dados do cargo. Se a pessoa não tem um cargo desses, o ponteiro será nulo. Se ela tem um desses cargos, ele aponta para uma `struct` com as informações do cargo. As informações adicionais da comissão são então usadas, quando apropriado, nos locais necessários.

Um elemento novo nesse código é a linha:

```
Colaborador() = default;
```

O significado disso é o seguinte: como definimos um construtor que recebe três parâmetros (nome, função e salário), seria impossível criar um objeto sem fornecer esses valores. No entanto, muitas vezes é conveniente declarar uma variável sem fornecer valores especiais, e deixar cada membro da classe ser inicializado com seus valores *default* (no caso, para uma cadeia é a cadeia vazia e para um double é zero). A linha acima indica que queremos fazer isso. Isto é, ela permite códigos da forma:

```
Colaborador desconhecido;
```

que ocasiona a geração de um objeto com `_nome=`, `_funcao=`, `_salario=0.0` e `_com_ptr` nulo.

O problema parece resolvido, mas se começarmos a analisar como objetos desse tipo podem ser usados, vamos encontrar diversos problemas, que serão discutidos a seguir com uma forma de solucioná-los.

2 O problema do término de escopo

Vejamos o que acontece no seguinte código:

```
int main(int, char *[])
{
```

```
    Colaborador marcio("Marcio Ferreira", "vendedor", 2500);
    Colaborador joana("Joana Ponte", "vendedor", 2500);
    joana.recebe_comissao("chefe de vendedores", 500);

    std::cout << marcio.nome() << " recebe "
              << marcio.a_pagar() << std::endl;
    std::cout << joana.nome()  << " recebe "
              << joana.a_pagar()  << std::endl;
}
```

Temos duas variáveis do tipo `Colaborador` declaradas: `marcio` e `joana`. Como essas variáveis deixam o escopo no final da execução de `main`, temos que analisar o que acontece com esses objetos. No caso de `marcio`, não temos problema, pois a memória usada por seus campos `_nome`, `_funcao_` e `_salario` serão liberadas automaticamente, e `_com_ptr` não está apontando para nada. No entanto, no caso de `joana` temos um problema: seus campos `_nome`, `_funcao_` e `_salario` terão a memória adequadamente liberada, mas a `struct` alocada dinamicamente e que tem seu ponteiro em `_com_ptr` não será liberada (como ela foi alocada com `new`, só será liberada quando fizermos `delete`). Neste código simples não é um problema, pois o programa irá terminar em seguida, mas em códigos mais complexos que lidem com um grande número de objetos pode ocorrer problema no gerenciamento de memória. Precisamos uma forma de indicar que, quando um objeto deixa de existir, se ele tiver uma `struct` alocada sendo apontada ela deve ser liberada.

2.1 Destruidores

Para lidar com esse tipo de problema, onde precisamos executar algum código quando um objeto deixa de existir, o C++ permite a definição de *destruidores*.

Um destruidor é um método para o qual o compilador gera automaticamente uma chamada sempre que o objeto for deixar de existir, quer dizer, se o objeto foi declarado como uma variável simples, o destruidor será chamado quando a variável sair do escopo; se o objeto foi criado com `new`, o destruidor será chamado quando for realizado `delete` sobre um ponteiro que aponta para esse objeto.

Para definir um destruidor, devemos definir um método com um nome que é o nome da classe *precedido* por `~`. Na nossa classe, basta acrescentar:

```
class Colaborador() {
    // O mesmo de antes..
    ~Colaborador() {
        delete _com_ptr;
    }
};
```

A instrução para o compilador dada aqui é: Quando um objeto da classe `Colaborador` deixar de existir, você deve fazer `delete` sobre o objeto apontado pelo campo `_com_ptr`. Se o `_com_ptr` é nulo, então o `delete` não faz nada, conforme o que desejamos.

3 O problema da cópia

Agora vamos supor que temos uma outra Joana Ponte, que também é vendedora, mas não é a chefe. Podemos querer fazer o seguinte:

```
int main(int, char *[])
{
    Colaborador joana("Joana Ponte", "vendedor", 2500);
    joana.recebe_comissao("chefe de vendedores", 500);

    // ... Mais adiante no código:
    Colaborador joana2{joana};
    joana2.deixa_comissao();

    std::cout << joana.nome()  << " recebe "
              << joana.a_pagar()  << std::endl;
}
```

Vamos seguir com cuidado o que aconteceu neste código:

1. Criamos o objeto `joana`.
2. Adicionamos uma `struct` de `Comissionado`, alocada dinamicamente, em `joana._com_ptr`.
3. Criamos `joana2` como *uma cópia* de `joana`. Como não instruimos o compilador sobre como copiar objetos do tipo `Colaborador`, ele irá usar o processo *default* de cópia, que consiste em copiar membro a membro os valores de `joana` em `joana2`. Isto significa que `joana._com_ptr` foi copiado para `joana2._com_ptr`, e os dois objetos *estão apontando para a mesma `struct` na memória*!
4. Quando chamamos `joana2.deixa_comissao()` para retirar a comissão de chefia inexistente para a segunda Joana, o método chamado realiza um `delete` do objeto `Comissionado` apontado. Como esse é o mesmo objeto apontado por `joana._com_ptr`, as informações de comissão da primeira Joana *também serão perdida*!
5. Quando a seguir executamos `joana.a_pagar()` esse método irá encontrar um `_com_ptr` não nulo, o que significa que ele irá tentar ler os valores da comissão. No entanto, *essas informações já foram apagadas*, gerando um erro de execução!

3.1 Construtor de cópia

Para resolver esse problema, precisamos instruir o compilador sobre como ser deve realizar uma cópia de objetos do tipo `Colaborador`. Isso pode ser feito definindo um **construtor de cópia** para essa classe. Um construtor de cópia é um construtor que aceita uma **referência** para um objeto da mesma classe, e cria um novo objeto idêntico.

```
class Colaborador() {
    // O mesmo de antes..
```

```
    Colaborador(Colaborador const &c)
        : _nome(c._nome), _funcao(c._funcao),
        _salario(c._salario), _com_ptr(nullptr) {
        if (c._com_ptr) {
            _com_ptr = new Comissionado{c._com_ptr->cargo,
                                        c._com_ptr->comissao};
        }
    }
};
```

Para os campos tipo `std::string` e `double`, basta copiar os valores do objeto original, pois isso é suficiente. Para o ponteiro, como queremos uma cópia, devemos ter a mesma comissão do objeto original (caso ela exista), mas não queremos usar o mesmo objeto de comissão, para evitar o problema descrito anterioremente; ao invés, criamos um novo objeto de comissão com valor idêntico ao original e guardamos um ponteiro para esse novo objeto.

4 O problema da atribuição

Uma forma alternativa de implementar o código das "duas Joanas"seria:

```
int main(int, char *[])
{
    Colaborador joana("Joana Ponte", "vendedor", 2500);
    joana.recebe_comissao("chefe de vendedores", 500);

    // ... Mais adiante no código:
    Colaborador joana3("", "", 0.0);
    joana3 = joana;
    joana3.deixa_comissao();

    std::cout << joana.nome()  << " recebe "
              << joana.a_pagar()  << std::endl;
}
```

Neste caso, esta forma de escrever é totalmente artificial, mas ainda assim é útil para indicar um outro problema que precisa ser resolvido. O problema é similar ao anterior. Agora, quando a variável `joana2` é criada, ela tem valores inúteis, que depois são substituídos através da atribuição. Como não indicamos ao compilador como fazer atribuição de objetos do tipo `Colaborador`, ele irá fazer atribuição pelo processo *default* de copiar membro a membro os campos de `joana` para `joana2`, resultando no mesmo problema discutido acima com o ponteiro `_com_ptr`!

5 O operador de atribuição

A solução é explicar ao compilador como fazer atribuição de objetos do tipo `Colaborador` através da sobrecarga do operador de atribuição para esse classe.

O operador de atribuição funciona de forma similar ao construtor de cópia, mas tem as seguintes peculiaridades adicionais que precisam ser consideradas em seu código:

- No construtor de cópia, estamos gerando *um novo objeto*, e portanto não precisamos nos preocupar com o que poderia já existir no objeto anteriormente. Já no operador de atribuição, estamos *modificando um objeto existente*, o que significa que precisamos pensar o que acontece com o que estava guardado nesse objetos anteriormente.
- O operador de atribuição pode ser utilizado em códigos que têm o mesmo significado que o seguinte:

  `a = a;`

  Normalmente você não vai fazer isso diretamente (não tem utilidade), mas isso pode surgir indiretamente através do uso de referências ou ponteiros:

  `*p1 = *p2;`

  Neste caso, tanto `p1` quanto `p2` podem estar apontando para o mesmo objeto.

Levando esses pontos em consideração, podemos implementar o operador de atribuição para a nossa classe de exemplo da seguinte forma:

```
class Colaborador {
    // O mesmo de antes...
    Colaborador& operator=(Colaborador const &c) {
        if (&c != this) {
            _nome = c._nome;
            _funcao = c._funcao;
            _salario = c._salario;
            if (_com_ptr) {
                delete _com_ptr;
                _com_ptr = nullptr;
            }
            if (c._com_ptr) {
                _com_ptr = new Comissionado{c._com_ptr->cargo,
                                            c._com_ptr->comissao};
            }
        }
        return *this;
    }
};
```

Alguns comentários:

- O operador de atribuição *deve* ser implementado como membro.
- O `if (&c != this)` (comparando o endereço do objeto original com o do objeto a ser atribuido) é a forma de verificar que não estamos no caso `a = a`; se estivermos nesse caso, nada precisa ser feito, além disso, a execução do código protegido por esse guarda resultaria em código errado no caso `a`

`= a`, pois o objeto `Comissionado` original seria apagado antes de usarmos o seu valor para copiar!

- O `if (_com_ptr)` é feito para garantir que liberamos um possível objeto `Comissionado` previamente existente (e que não será mais necessário).
- Como todo operador de atribuição, ele deve retornar uma referência para o objeto atribuído.

6 Semântica de movimento

Quando estamos lidando com recursos, devemos considerar alguns casos cuidadosamente. No exemplo anterior, o recurso é uma região de memória alocada dinamicamente; outros exemplos de recursos são arquivos abertos, ou semáforos de exclusão mútua para programação de *threads*. Para alguns recursos, podemos ter um (ou ambos) dos seguintes casos:

1. O recurso não pode ser copiado. Um exemplo que já vimos é o caso do `std::unique_ptr`: não podemos copiá-lo pois queremos apenas um ponteiro se responsabilizando pelo objeto apontado.
2. O recurso pode ser copiado, mas a cópia é muito cara. Por exemplo, no caso de um `std::vector` com muitos elementos.

Para lidar com esses casos em situações onde precisamos passar o conteúdo de um lugar para outro, a linguagem C++ define a denominada **semântica de movimento**, quer dizer, permite especificar como o conteúdo de um objeto pode ser transferido para outro objeto, sem gerar uma cópia.

O compilador C++ usa semântica de movimento para gerar códigos otimizados em diversas situações. Por exemplo:

```
std::vector<int> multiples_of(int m, int n) {
    std::vector<int> res(n);
    for (int i = 0; i < n; ++i) {
        res[i] = i * m;
    }
    return res;
}

int main(int, char *[])
{
    std::vector<int> m3;
    m3 = multiples_of(3, 1000000);
}
```

Neste código, a função `multiples_of` gera um vetor local `res` grande (4 milhões de bytes), que deve ser retornado para ter seu valor colocado em `m3`. Em princípio, uma cópia de `res` deve ser feita para colocar em `m3`, mas isto seria muito custoso. Como o compilador sabe que `res` não será mais usado depois do `return`, ele ao invés de *copiar* os dados para `m3`, apenas *move* o conteúdo, que implica copiar algumas informações de tamanho e um ponteiro de um objeto

para outro. Isto é feito pois a classe `std::vector` define a sua semântica de movimento.

Para definir a semântica de movimento de uma classe, devemos definir duas coisas:

- Como criar um objeto movendo as informações de um outro objeto do mesmo tipo. Isto é definida através do *construtor de movimento*.
- Como transferir as informações de um objeto para outro objeto do mesmo tipo. Isto é definido através do *operador de movimento*.

6.1 Referência de *rvalue*

Para definir esses métodos, precisamos introduzir os conceitos de *lvalue*, *rvalue* e referência de *rvalue*.

Considere o código:

```
a = b;
```

Neste código, os elementos `a` e `b` têm características distintas. Para `b`, podemos usar muitas coisas distintas, como

- uma variável;
- uma expressão;
- uma constante;
- uma constante literal;
- um valor retornado por uma função.

Na verdade, em todos esses casos estamos lidando com casos especiais de valores gerados por diversos tipos de expressões. Já no caso de `a`, estamos muito mais limitados. Não podemos fazer loucuras como

```
2 = 5; // Não funciona. Nem em Brasília.
```

Na verdade, o que aparece à esquerda de uma atribuição deve ser algo que represente uma posição de memória, que irá receber o valor da expressão à esquerda. Por exemplo:

- uma variável;
- uma expressão `*p` onde `p` é um ponteiro;
- uma referência (possivelmente retornada por função ou operador).

De uma forma simplificada, denominamos o tipo de elementos que podem aparecer à esquerda de uma atribuição como *lvalues* (de *left values*) e elementos que apenas podem aparecer à direita de uma atribuição de *rvalues* (*right values*).

As referências que discutimos até agora apenas podem ser feitas para *lvalues*. Por exemplo, o código abaixo não funciona:

```
void poe_5(int &i) { i = 5; }
int main(int, char *[])
{
    poe_5(2); // Não funciona, nem com pensamento positivo.
}
```

Neste caso, 2 é uma constante literal, portanto um *rvalue*, e não podemos fazer uma referência para ele.

Se precisamos uma referência para um *rvalue* (como vamos usar a seguir), então precisamos definir a referência como `&&`, a denominada *referência de rvalue*. Precisamos dela quando queremos evitar fazer uma cópia de um elemento, mas esse elemento é um *rvalue* e não um *lvalue*.

6.2 Construtor de movimento

O construtor de movimento é usado quando vamos construir um novo objeto partindo de um objeto do mesmo tipo que é um *rvalue*, transferindo os recursos do *rvalue* para o novo objeto.

A declaração é similar à do construtor de cópia, mas substituimos a referência a um outro objeto por uma referência a *rvalue* (`&&`).

```
class Colaborador() {
    // O mesmo de antes..
    Colaborador(Colaborador &&c)
        : _nome(c._nome), _funcao(c._funcao),
        _salario(c._salario), _com_ptr(c._com_ptr) {
        c._com_ptr = nullptr;
    }
};
```

Note como neste caso basta simplesmente copiar todos os membros do objeto original para o novo objeto, incluindo o ponteiro. Isso pode ser feito porque no construtor de movimento estamos retirando os recursos do objeto original e colocando-os no novo objeto, e portanto o objeto original não será mais utilizado (sem realizar nova inicialização). Para garantir que o objeto original não tente acessar o recurso transferido, também anulamos seu ponteiro.

6.3 Operador de movimento

O operador de movimento é uma variante do operador de atribuição, que realizada a transferência dos recursos de um objeto para outro, ao invés de realizar uma cópia.

A declaração é similar à do operador de atribuição:

```
class Colaborador {
    // O mesmo de antes...
    Colaborador& operator=(Colaborador &&c) {
        if (&c != this) {
            _nome = c._nome;
            _funcao = c._funcao;
            _salario = c._salario;
```

```
            delete _com_ptr;
            _com_ptr = c._com_ptr;
            c._com_ptr = nullptr;
        }
        return *this;
    }
};
```

Simplesmente verificamos o caso `a = a` (se for esse caso, não precisamos transferir nada), caso contrário copiamos os membros do objeto original, incluindo o ponteiro, mas lembrando de apagar possíveis dados presentes antes no ponteiro do objeto atribuído. Também como no construtor de movimento, marcamos anulamos o ponteiro do objeto original.

7 A regra dos cinco

Acima apresentamos a necessidade de definição de cinco métodos para certos tipos de objetos:

1. O destruidor.
2. O construtor de cópia.
3. O operador de atribuição.
4. O construtor de movimento.
5. O operador de movimento.

Para a correta implementação desse tipo de objetos, deve ser seguida a denominada **regra dos cinco**:

> *Se um dos métodos identificados acima precisa ser implementado para uma dada classe, então **todos** os cinco devem ser implementados.*

Lembrar-se dessa regra pode evitar o surgimento de problemas que são muito difícieis de depurar no seu código.

8 A regra dos zero

Está claro que o implicado pela *regra dos cinco* acima representa uma grande sobrecarga ao programador: cinco métodos de implementação não trivial devem ser desenvolvidos simultaneamente nos casos em que são necessários.

Para evitar esse problema, se estabeleceu a **regra dos zero**:

> *Procure desenvolver as suas classes de tal modo que **nenhum** dos cinco métodos acima precisem ser implementados.*

Para garantir que esses métodos não sejam necessários, existem diversas possibilidades:

1. Se os seus objetos não lidam com recursos ou outras características que exijam a implementação de algum dos métodos, então você não precisa implementar nenhum deles.

2. Se os seus métodos lidam com recursos, então você tem as seguintes opções antes de precisar usar a regra dos cinco:

   - Você pode usar objetos especiais presentes na biblioteca de C++ para lidar com os recursos, transformando a sua classe numa classe simples.
   - Se não houver uma classe auxiliar adequada na biblioteca, você pode desenvolver uma classe especial apenas para lidar com o recurso, e então implementar a sua classe usando essa classe auxiliar, ao invés de misturar na sua classe a lógica da classe com os problemas de gerenciamento do recurso.

No caso da nossa classe `Colaborador` ela lida com recursos (a memória alocada para as informções de comissão), então estamos no segundo caso. Felizmente, temos uma classe da biblioteca que pode ser usada para ajudar a lidar com o recurso: o `std::unique_ptr`. Podemos então refazer o código seguindo a regra dos zero da seguinte forma:

```
#include <memory>
#include <string>

class Colaborador {

  struct Comissionado {
    std::string cargo;
    double comissao;
    Comissionado(std::string c, double v) : cargo(c), comissao(v) {}
  };

  struct PonteiroComissao {
    std::unique_ptr<Comissionado> ptr;

    PonteiroComissao() = default;

    PonteiroComissao(std::string cargo, double comissao) {
      ptr = std::make_unique<Comissionado>(cargo, comissao);
    }

    PonteiroComissao(PonteiroComissao const &pc) {
      ptr = std::make_unique<Comissionado>(pc.ptr->cargo,
                                           pc.ptr->comissao);
    }

    PonteiroComissao(PonteiroComissao &&pc) : ptr(std::move(pc.ptr)) {}

    PonteiroComissao &operator=(PonteiroComissao const &pc) {
      if (&pc != this) {
```

```
        auto tmp{
            std::make_unique<Comissionado>(pc.ptr->cargo,
                                           pc.ptr->comissao)};
        ptr = std::move(tmp);
      }
      return *this;
    }

    PonteiroComissao &operator=(PonteiroComissao &&pc) {
      ptr = std::move(pc.ptr);
      return *this;
    }
  };

  std::string _nome;
  std::string _funcao;
  double _salario;
  PonteiroComissao _com;

public:
  Colaborador() = default;

  Colaborador(std::string nome, std::string funcao, double salario)
      : _nome(nome), _funcao(funcao), _salario(salario) {}

  std::string nome() const { return _nome; }

  std::string funcao() const {
    auto res{_funcao};
    if (_com.ptr) {
      res += " [" + _com.ptr->cargo + "]";
    }
    return res;
  }

  double a_pagar() const {
    auto res{_salario};
    if (_com.ptr) {
      res += _com.ptr->comissao;
    }
    return res;
  }

  void deixa_comissao() { _com.ptr.reset(); }

  void recebe_comissao(std::string cargo, double comissao) {
    _com = PonteiroComissao(cargo, comissao);
  }
};
```

As alterações feitas foram as seguintes:

- Criamos uma nova `struct` adicional, denominada `PonteiroComissao`, para representar um ponteiro para a `struct Comissionado`, ao mesmo tempo lidando da forma necessária com o recurso.
- Dentro dessa classe, usamos um `std::unique_ptr`, que já sabe gerenciar ponteiros que apontam para memória alocada dinamicamente.
- Declaramos agora `_com_` como um `PonteiroComissao`. Essa classe faz o gerenciamento do ponteiro associado com ela automaticamente.
- Infelizmente, não podemos usar diretamente o `std::unique_ptr` na nossa classe, pois ele não permite cópias, o que significa que não poderíamos copiar objetos da nossa classe, como fazemos no código do `main`. Para permitir isso, implementamos `PonteiroComissao`, que fornece os métodos de cópia (construtor de cópia de operador de atribuição) necessários. Assim, quando fazemos uma cópia de um `Colaborador`, o C++ automaticamente chama as operações de cópia de `PonteiroComissao` na hora de copiar o membro `_com`.
- Não precisamos inicializar o ponteiro como nulo, pois a classe `std::unique_ptr` já faz isso se não fornecermos um ponteiro para seu construtor.
- No método `deixa_comissao()` basta chamar o método `reset` do ponteiro único, pois ele já libera o objeto apontado e coloca o ponteiro como nulo.
- No método `recebe_comissao(std::string, double)`, não precisamos nos preocupar com o gerenciamento do recurso (alocação da nova `struct` Comissionado ou liberação de uma possível comissão já existente), pois isso é feito pela classe `PonteiroComissao` automaticamente, com ajuda do `std::unique_ptr`. Basta criar um objeto `PonteiroComissao` com os valores corretos e colocá-lo no campo `_com`.
- Na implementação de `PonteiroComissao`, basta criarmos os construtores, operador de atribuição e operador de movimento. Neste caso, o destruidor não é necessário, pois o destruidor do `std::unique_ptr` usado já lida com a liberação do recurso.
- Ao construir um `PonteiroComissao~com novos valores ou fazer uma cópia, basta criar novo ~std::unique_ptr` para um objeto `Comissionado` com os valores corretos, e colocá-los no campo `ptr`.
- Para mover de um `PonteiroComissao` para outro, basta mover o `std::unique_ptr` entre eles.
- Para fazer a atribuição, depois de verificarmos o caso `a = a`, basta criar um novo `std::unique_ptr` com os valores do objeto original e movê-lo para `ptr`. O operador de movimento de `std::unique_ptr` se encarrega de liberar uma possível memória anteriormente alocada.
- O mesmo vale para o operador de movimento: neste caso, não precisamos de uma cópia, e basta mover o `std::unique_ptr`.

9 RAII

Ligado a esse assunto, existe um dialeto de C++ frequentemente usado, tanto em programas de usuário como em bibliotecas, que recebe o nome RAII, cuja origem vem de *Resource Aquisition Is Initialization*.

A idéia é que todos os recursos a serem usados pelo programa devem ser adquiridos através de uma classe especial para lidar com eles, cujo construtor (durante a inicialização, portanto) adquire o recurso, e cujo destruidor cuida da sua liberação. A classe também se ocupa de copiar ou transferir o recurso conforme apropriado.

Dois exemplos que já vimos são:

- As classes para lidar com arquivos, `std::ifstream` e `std::ofstream`. Na construção, abrimos o arquivo e os destruidores dessas classes cuidam da liberação (fechar) do arquivo. Também, a classe é feita para garantir que não existem dois objetos com o mesmo arquivo aberto (através do controle dos operadores de cópia e atribuição).

- A classe `std::unique_ptr`, que gerencia um elemento alocado na memória através do ponteiro. Ao transferirmos o ponteiro para um `unique_ptr` ele se encarrega de liberar os dados alocados automaticamente. Também impede que outro `unique_ptr` se refira ao mesmo objeto.