INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus Cajazeiras

Engenharia de Controle e Automação
Programação Orientada a Objetos – 2021.2

Professor: Leandro Luttiane da Silva Linhares
E-mail: leandro.luttiane@ifpb.edu.br

Documento Texto

Sobrecarga de Operadores – Parte II

Na primeira parte do documento de texto que trata da **Sobrecarga de Operadores** na linguagem C++, foi apresentado como alguns operadores desta linguagem podem ser sobrecarregados para trabalhar com objetos de tipos definidos pelo usuário. Neste intuito, foi apresentada a sintaxe de declaração das funções operadoras e de que forma elas podem ser criadas: como funções-membro da classe (métodos) ou como funções globais (`friends` ou não). Além disso, foi apresentado um exemplo de classe, `CellNumber`, que implementava a sobrecarga dos operadores de inserção fluxo "`<<`" e de extração de fluxo "`>>`". Isso permitiu que estes operadores sobrecarregados fossem utilizados para a saída e entrada de objetos dessa classe, respectivamente.

Nesta segunda parte de nossos estudos sobre **Sobrecarga de Operadores**, iremos abordar como é realizado a sobrecarga de operadores unários e binários. Adicionalmente, mais um estudo de caso será analisado: a classe `Array`. Juntamente com este documento encontram-se disponibilizados os arquivos `Array.hpp`, `Array.cpp` e `mainArray.cpp`. Mais uma vez, deve-se salientar que o conteúdo apresentado neste documento tem como base o ***Capítulo 11 do livro C++: Como Programar*, de Deitel e Deitel (2006)**.

1. Sobrecarga de Operadores Unários

A sobrecarga de um operador unário em C++ pode ser realizada a partir de um método não estático sem argumentos ou a partir de uma função global com um único argumento. Nesse último caso, o argumento deve ser um objeto da classe ou uma referência a um objeto da classe.

Quando uma função-membro (método) é utilizada para efetuar a sobrecarga de algum operador ela deve ser declarada como sendo automática (não-`static`) para que ela possa acessar os dados não estáticos dos objetos da classe. Vimos em aula anterior que os métodos `static` apenas possuem permissão de acesso aos atributos `static` da classe, uma vez que tanto os métodos quanto os atributos estáticos não necessitam que algum objeto da classe tenha sido instanciado para que eles existam. Métodos e atributos estáticos estão associados à classe e não aos objetos instanciados da classe.

Vamos agora considerar a expressão `!s`, em que `s` é um objeto de uma classe denominada `String`. Nesse caso, o operador "`!`" é sobrecarregado e tem a funcionalidade de retornar `true` caso um objeto da classe `String` esteja vazio. Quando um operador unário como "`!`" é sobrecarregado por meio de um método sem argumentos, o compilador, ao encontrar a expressão

`!s`, gera a chamada `s.operador!()`. Assim, `s` é o objeto de classe pelo qual a função operadora `operator!` da classe `String` está sendo chamada. Este método pode ser declarado na definição da classe da seguinte maneira:

```
class String
{
    public:
        bool operator!() const;
    ...
};
```

A sobrecarga de um operador unário também pode ser realizada por meio de uma função global com um único argumento. Se esta for a solução adotada para sobrecarregar operadores unários como "`!`" há duas opções para criação da função operadora: **i**. com um argumento que seja um objeto da classe (isso requer uma cópia do objeto e as alterações feitas neste objeto não serão aplicadas ao objeto original); **ii**. com um argumento que seja uma referência a um objeto da classe (nenhuma cópia do objeto original é realizada, mas todas as mudanças realizadas pela função operadora no objeto referenciado, estará, na verdade, modificando o objeto original). Independente de qual escolha seja tomada para sobrecarregar um operador unário por meio de uma função global, se `s` for um objeto da classe `String`, então a expressão `!s` será tratada como se em seu lugar tivesse sido escrito o código `operator!(s)` de chamada da função operadora correspondente. Para isso, a função global `operator!` poderia ser declarada da seguinte maneira: `bool operator!(const String &)` ou `bool operator!(String)`.

2. Sobrecarga de Operadores Binários

Um operador binário da linguagem C++ pode ser sobrecarregado por meio de um método não estático com um único argumento ou com uma função global com dois argumentos. Neste último caso, pelo menos um dos argumentos deve ser um objeto de classe ou uma referência a um objeto de classe.

Suponha que em uma classe definida pelo usuário denominada `String`, o operador binário "`<`" é sobrecarregado como uma função-membro não estática com um argumento. Considerando que `x` e `y` são objetos da classe `String`, quando o compilador encontra a expressão `x < y`, ele gera a chamada da função operadora `x.operator<(y)` em seu lugar. Desta forma a função-membro `operator<` poderia ser declarada na definição da classe da seguinte maneira:

```
class String
{
    public:
        bool operator<( const String & ) const;
        ...
};
```

Se o operador binário "<" for sobrecarregado como uma função global, ela deve aceitar dois argumentos, sendo que pelo menos um deles deve ser um objeto de classe ou uma referência a um objeto de classe. Se `x` e `y` forem objetos da classe `String` ou referências a objetos desta classe, então a expressão `x < y` é tratada pelo compilador como se o código `operator<(x,y)` tivesse sido escrito em seu lugar, realizando a chamada da função global `operator<` que pode ser declarada da seguinte forma: `bool operator<(const String &, const String &)`.

3. Classe Array: Estudo de Caso

Este é o momento certo para vocês abrirem os arquivos `Array.hpp`, `Array.cpp` e `mainArray.cpp`, pois a partir de agora iniciaremos a analisar a implementação da classe `Array`. Esta classe é utilizada para armazenar vetores de inteiros, de diferentes tamanhos. Ela faz uso de alocação dinâmica de memória, que fez parte da última atividade realizada na disciplina de **Programação Orientada a Objetos** do **Curso de Engenharia de Controle e Automação**. Primeiramente, vamos verificar o que o programa de exemplo de utilização da classe `Array`, `mainArray.cpp`, realiza.

3.1 Arquivo Principal de Exemplo (`mainArray.cpp`)

No arquivo `mainArray.cpp`, o programa inicia instanciando dois objetos (linhas 15 e 16) da classe `Array`. O objeto `integers1` corresponde a um vetor de sete elementos e `integers2` um vetor de tamanho padrão com 10 elementos, conforme definido pelo construtor padrão no arquivo `Array.hpp` (linha 19). Nas linhas 19-21, do arquivo `mainArray.cpp`, o método `getSize` é chamado para exibir o tamanho de `integers1`. Além disso, o objeto `integers1` é exibido no terminal por meio do operador de inserção de fluxo "<<" sobrecarregado na classe `Array`. A saída de exemplo apresentada pela execução do trecho de código da linha 26, demonstra que os elementos de `integers1` foram inicializados corretamente com zeros pelo construtor.

```
15      Array integers1( 7 ); // Array de sete elementos
16      Array integers2;      // Array de 10 elementos por padrão
17
18      // imprime o tamanho e o conteúdo de integers1
19      cout << "Tamanho do vetor integers1 eh "
20           << integers1.getSize()
21           << "\nVetor apos a inicializacao:\n" << integers1;
22
23      // imprime o tamanho e o conteúdo de integers2
24      cout << "\nTamanho do vetor integers2 eh "
25           << integers2.getSize()
26           << "\nVetor apos a inicializacao:\n" << integers2;
27
28      // insere e imprime integers1 e integers2
29      cout << "\nDigite 17 inteiros:" << endl;
30      cin >> integers1 >> integers2;
```

A linha 29 do arquivo `mainArray.cpp` solicita que o usuário informe 17 inteiros para serem inseridos nos objetos `integers1` e `integers2`. Com este propósito, a linha 30 faz uso do

operador de extração de fluxo “>>” sobrecarregado pela classe `Array` para efetuar a leitura dos valores informados pelo usuário. Os sete primeiros valores fornecidos são armazenados em `integers1` e os 10 valores restantes em `integers2`. As linhas 32-34 exibem em tela os elementos dos dois objetos `Array` por meio do operador de inserção de fluxo “<<” sobrecarregado, confirmando que a entrada de informações foi realizada de maneira correta. A linha de código 39, avalia se os dois objetos, `integers1` e `integers2`, são diferentes, o que é confirmado pela saída do programa. Esta avaliação é efetuada por meio da sobrecarga do operador “`!=`”.

A linha 44 instancia um novo objeto da classe `Array`, o objeto `integers3`. Ele é inicializado como uma cópia do objeto já existente `integers1`. Este procedimento, efetua a chamada do construtor de cópia da classe `Array` (linhas 33-40 do arquivo `Array.cpp`), fazendo com que os elementos contidos em `integers1` sejam copiados para `integers3`. Essa linha de código poderia ser substituída por `integers3 = integers1`. O sinal de igual, nesse caso, NÃO seria o operador de atribuição, pois quando um sinal de igual aparece na declaração de um objeto, ele invoca um método construtor. Essa forma de utilização pode ser aplicada para passar um único argumento de objeto da classe para um construtor.

```
32      cout << "\nApós a entrada de valores, os vetores contêm:\n"
33           << "integers1:\n" << integers1
34           << "integers2:\n" << integers2;
35
36      // utiliza o operador de desigualdade (!=) sobrecarregado
37      cout << "\nAvaliacao: integers1 != integers2" << endl;
38
39      if ( integers1 != integers2 )
40         cout << "integers1 e integers2 nao sao iguais" << endl;
41
42      // cria Array integers3 utilizando integers1 como um
43      // inicializador; imprime tamanho e conteúdo
44      Array integers3( integers1 ); // invoca o construtor de cópia
45
46      cout << "\nTamanho do vetor integers3 eh "
47           << integers3.getSize()
48           << "\nVetor apos a inicializacao:\n" << integers3;
```

A linha 52 testa o operador de atribuição “=” sobrecarregado, efetuando a atribuição de `integers2` para `integers1`. As linhas 54 e 55 exibem em tela estes dois objetos com o objetivo de confirmar que a atribuição foi realizada com sucesso. É importante comentar que `integers1` armazenava originalmente 7 números inteiros e precisou ser redimensionado para armazenar uma cópia dos 10 elementos contidos em `integers2`. Essa operação de redimensionamento de um objeto `Array` é realizada de forma transparente ao código-cliente. Na linha 57, o operador de igualdade “==” sobrecarregado é utilizado para confirmar que os objetos `integers1` e `integers2` tornaram-se idênticos após a atribuição executada na linha 52.

```
51      cout << "\nAtribuindo integers2 para integers1:" << endl;
52      integers1 = integers2; // note que o Array alvo é menor
53
54      cout << "integers1:\n" << integers1
55          << "integers2:\n" << integers2;
56
57      // utiliza operador de igualdade (==) sobrecarregado
58      cout << "\nAvaliacao: integers1 == integers2" << endl;
59
60      if ( integers1 == integers2 )
61          cout << "integers1 e integers2 sao iguais" << endl;
62
63      // utiliza operador de subscrito sobrecarregado para criar rvalue
64      cout << "\nintegers1[5] eh " << integers1[ 5 ];
```

Na linha 64 do arquivo `mainArray.cpp`, o operador de subscrito "`[]`" é sobrecarregado para referenciar o sexto elemento armazenado em `integers1` (`integers1[5]`). Neste caso, o subscrito é utilizado como um *`rvalue`* (valor direito) para imprimir o valor armazenado em `integers1[5]`. Observem que `integers1[5]` encontra-se do lado direito do operador de inserção de fluxo "`<<`" e que o operador de subscrito sobrecarregado nas linhas 107-115 do arquivo `Array.cpp` retorna um número inteiro, um elemento contido no objeto da classe.

A linha 68 de `mainArray.cpp` utiliza `integers1[5]` como um *`lvalue`* (valor esquerdo) de uma instrução de atribuição para atribuir o valor 1000 ao sexto elemento de `integers1`. Em breve, veremos que a função operadora `operator[]`, linhas 122-128 do arquivo `Array.cpp`, depois de confirmar que o subscrito (ou índice) é válido, retorna uma referência de um objeto da classe `Array`. Isto é realizado exatamente para permitir que o operador subscrito sobrecarregado atue como um *`lvalue`* modificável.

A linha de código 73 do arquivo principal tenta atribuir o valor 1000 para `integers1[15]`. Este índice está fora do intervalo dos elementos de `integers1`, uma vez que este vetor contém apenas 10 elementos. Como resultado da execução desta linha de código, a função operadora `operator[]` das linhas 92-100 de `Array.cpp`, verifica que o índice 15 está fora do intervalo, imprime uma mensagem de erro em tela e finaliza o programa. Acessar um elemento fora do intervalo de um vetor é um erro de lógica em tempo de execução, não um erro de compilação.

```
66      // utiliza operador de subscrito sobrecarregado para criar lvalue
67      cout << "\n\nAtribuindo o valor 1000 para integers1[5]" << endl;
68      integers1[ 5 ] = 1000;
69      cout << "integers1:\n" << integers1;
70
71      // tentativa de utilizar subscrito fora do intervalo
72      cout << "\nTentativa de atribuir o valor 1000 para integers1[15]" << endl;
73      integers1[ 15 ] = 1000; // ERRO: fora do intervalo
```

A linguagem C++ oferece bastante flexibilidade quando se deseja sobrecarregar o operador subscrito "`[]`". Quando definimos funções operadoras `operator[]`, os subscritos (índices) não precisam ser obrigatoriamente números inteiros. Eles podem ser, por exemplo, caracteres, strings, números de ponto flutuante e até mesmo objetos de classes definidas pelo usuário.

3.2 Implementação da Classe (`Array.hpp` e `Array.cpp`)

Após analisarmos em detalhes como o programa principal de exemplo de uso da classe `Array` funciona, iremos percorrer o arquivo de definição desta mesma classe. À medida que um método do arquivo de cabeçalho `Array.hpp` for encontrado, a sua implementação no arquivo `Array.cpp` será discutida. Vamos começar analisando os atributos privados da classe nas linhas 41 e 42 do arquivo `Array.hpp`. O membro de dados `size` indica o número de elementos armazenados no objeto `Array`. O outro atributo, `ptr`, corresponde a um ponteiro que aponta para um vetor de inteiros dinamicamente alocado e gerenciado pelo objeto `Array`.

```
40      private:
41          int size; // tamanho do array baseado em ponteiro
42          int *ptr; // ponteiro para o primeiro elemento do array baseado em ponteiro
```

As linhas de código 15 e 16 declaram os operadores de inserção "<<" e extração ">>" de fluxo sobrecarregados como funções globais `friends` da classe `Array`. No momento em que o compilador encontra uma expressão como `cout << objetoArray`, ele efetua a chamada da função global `operator<<` da seguinte maneira: `operador<<(cout, objetoArray)`. Quando uma expressão como `cin >> objetoArray` é encontrada pelo compilador, ele invoca a função operadora global `operator>>` por meio da chamada `operador>>(cin, objetoArray)`.

```
15      friend ostream &operator<<( ostream &, const Array & );
16      friend istream &operator>>( istream &, Array & );
```

Lembrem-se da Parte I deste documento!! As funções operadoras de inserção e de extração de fluxo não podem ser declaradas como funções-membro da classe `Array`, pois os objetos desta classe aparecem sempre à direita dos operadores que se deseja sobrecarregar, "<<" e ">>".

A função `operator<<`, implementada nas linhas 131-148 do arquivo `Array.cpp`, imprime os elementos contidos no vetor de inteiros para qual `ptr` aponta. A função `operator>>`, implementada nas linhas 122-128, captura a entrada de informações diretamente para o vetor de inteiros para o qual o atributo `ptr` aponta. Estas duas funções operadoras retornam uma referência para permitir que as instruções de saída e entrada, respectivamente, possam ser cascateadas como em `cout << objetoArray1 << objetoArray2` ou `cin >> objetoArray1 >> objetoArray2`.

```
122 istream &operator>>( istream &input, Array &a )
123 {
124     for ( int i = 0; i < a.size; i++ )
125         input >> a.ptr[ i ];
126
127     return input; // permite cin >> x >> y;
128 } // fim da função
```

```
131 ostream &operator<<( ostream &output, const Array &a )
132 {
133    int i;
134
135    // gera saída do array baseado em ptr private
136    for ( i = 0; i < a.size; i++ )
137    {
138       output << setw( 12 ) << a.ptr[ i ];
139
140       if ( ( i + 1 ) % 4 == 0 ) // 4 números por linha de saída
141          output << endl;
142    } // fim do for
143
144    if ( i % 4 != 0 ) // termina a última linha de saída
145       output << endl;
146
147    return output; // permite cout << x << y;
148 } // fim da função operator<<
```

As funções `operator<<` e `operator>>` possuem acesso aos dados privados de um objeto `Array` porque essas funções foram declaradas como funções amigas (`friends`) da classe `Array`. Além disso, observe que as funções-membro `getSize` e `operator[]` poderiam ser utilizadas por `operator<<` e `operator>>`. Neste caso, elas não precisariam ser `friends` da classe `Array`. Entretanto, as chamadas de métodos adicionais poderiam aumentar o *overhead* de tempo de execução. Para finalizarmos esta questão, observem que na definição das funções `operator>>` e `operator<<` no arquivo `Array.cpp` (linhas 122 e 131, respectivamente) não foi necessário utilizar o operador de resolução de escopo "`::`" precedido pelo nome da classe `Array`. Isso ocorre porque estas funções operadoras globais não são métodos da classe.

A linha 19 do arquivo `Array.hpp` declara o construtor padrão da classe `Array`, especificando um tamanho padrão de 10 elementos para um objeto dessa classe. Quando o compilador encontra uma declaração de objeto `Array` como a da linha 16 do arquivo `mainArray.cpp`, ele realiza a chamada do construtor padrão da classe. Neste caso, o construtor padrão é chamado com um único argumento igual a 10. Analisando a implementação desse construtor, linhas 22-29 do arquivo `Array.cpp`, este método especial da classe `Array` valida o número de elementos informados, atribui o número de elementos ao atributo `size` e realiza a alocação dinâmica de um vetor de inteiros, que passa a ser apontado pelo ponteiro `ptr`. Em seguida, com o auxílio de uma estrutura de repetição `for`, inicializa todos os elementos do vetor recém alocado com o valor zero. Inicializar todos os atributos de um objeto é considerada uma boa prática de programação, isso garante que os objetos estejam em um estado consistente.

```
22 Array::Array( int arraySize )
23 {
24    size = ( arraySize > 0 ? arraySize : 10 ); // valida arraySize
25    ptr = new int[ size ]; // cria espaço para array baseado em ponteiro
26
27    for ( int i = 0; i < size; i++ )
28       ptr[ i ] = 0; // configura elemento do array baseado em ponteiro
29 } // fim do construtor-padrão de Array
```

A linha 20 do arquivo `Array.hpp` declara um construtor de cópia. Ele realiza a inicialização de um objeto `Array` por meio de uma cópia de um objeto `Array` já existente. Como um objeto da classe `Array` possui um atributo que é um ponteiro, é preciso que essa cópia seja feita com muito cuidado, evitando que objetos `Array` diferentes apontem para a mesma posição de memória alocada dinamicamente. Este seria o problema que ocorreria se o compilador tivesse a permissão de definir um construtor de cópia padrão para a classe `Array`. O objeto inicializado a partir da cópia de um outro objeto teria o seu atributo `ptr` apontando para a mesma posição de memória apontada pelo atributo `ptr` do objeto copiado. Isso representaria uma grande inconsistência.

Os construtores de cópia são sempre chamados quando a cópia de um objeto for necessária: quando um objeto é passado por valor para uma função, ao retornar um objeto por valor a partir de uma função ou ao inicializar um objeto como uma cópia de outro objeto da mesma classe.

O construtor de cópia da classe `Array` (linhas 33-40 do arquivo `Array.cpp`) utiliza um inicializador de membro para copiar o valor do atributo `size` do inicializador para o membro de dados `size` do objeto cópia. Na linha 36, `new` é utilizado para alocar espaço em memória de de forma dinâmica, retornando um ponteiro que é atribuído ao membro de dados (atributo) `ptr`. Em seguida, nas linhas 38-39, o construtor de cópia copia todos os elementos do objeto `Array` inicializador para o novo objeto `Array`.

```
33  Array::Array( const Array &arrayToCopy )
34      : size( arrayToCopy.size )
35  {
36      ptr = new int[ size ]; // cria espaço para array baseado em ponteiro
37
38      for ( int i = 0; i < size; i++ )
39          ptr[ i ] = arrayToCopy.ptr[ i ]; // copia para o objeto
40  } // fim do construtor de cópia do Array
```

Duas observações importantes podem ser feitas em relação ao código do construtor de cópia da classe `Array`. A primeira delas é que um objeto de uma classe pode ter acesso aos dados `private` de qualquer objeto dessa classe. Vejam que conseguimos acessar os atributos `size` (linha 34) e `ptr` (linha 39) do objeto `arrayToCopy` passado por referência. A outra observação é que um construtor de cópia deve sempre receber seu argumento por referência, não por valor. Caso contrário, a chamada do construtor de cópia resultaria em uma recursão infinita, pois receber um objeto por valor requer que o construtor de cópia faça uma cópia do objeto de argumento.

A linha 21 do arquivo `Array.hpp` declara o destrutor da classe. Sempre que um objeto sai de seu escopo, este objeto precisa ser destruído, o que faz com que o destrutor da classe seja invocado. Em sua implementação, linhas 43-46 do arquivo `Array.cpp`, podemos observar a utilização de `delete []` para liberar a memória alocada dinamicamente por `new` nos construtores da classe.

```
43  Array::~Array()
44  {
45      delete [] ptr; // libera espaço do array baseado em ponteiro
46  } // fim do destrutor
```

A linha 25 do arquivo `Array.hpp` declara a função operadora que sobrecarrega o operador de atribuição "=". Ao encontrar a expressão `integers1 = integers2` na linha 52 do arquivo `mainArray.cpp`, o compilador efetua a chamada do método `operator=` por meio de `integers1.operator=(integers2)`. Na implementação da função `operator=`, linhas 56-71 do arquivo `Array.cpp`, é testado se uma auto atribuição está sendo realizada (linha 62). Deve-se ter em mente que `this` corresponde a um ponteiro que permite que qualquer objeto tenha acesso ao seu próprio endereço. O ponteiro `this` é um parâmetro implícito para todas as funções-membro (métodos) de uma classe. Assim, em um método, o ponteiro `this` pode ser utilizado para referenciar o objeto que deu origem à chamada do método. A condição da linha 58 (`&right != this`) verifica se o endereço do objeto passado por referência para `operator=`, operando da direita (`right`), é igual ao endereço do objeto apontado por `this` (lembre-se que um ponteiro armazena um endereço de memória). Se esses endereços forem iguais, então os objetos são os mesmos e uma auto atribuição será tentada.

```
56 const Array &Array::operator=( const Array &right )
57 {
58    if ( &right != this ) // evita auto-atribuição:
59    {
60       // para Arrays de tamanhos diferentes, desaloca array do lado esquerdo
61       // original, então aloca o novo array à esquerda
62       if ( size != right.size )
63       {
64          delete [] ptr; // libera espaço
65          size = right.size; // redimensiona esse objeto
66          ptr = new int[ size ]; // cria espaço para a cópia do array
67       } // fim do if interno
68
69       for ( int i = 0; i < size; i++ )
70          ptr[ i ] = right.ptr[ i ]; // copia o array para o objeto
71    } // fim do if externo
72
73    return *this; // permite x = y = z, por exemplo
74 } // fim da função operator=
```

Caso a operação não seja uma auto atribuição, é verificado se os tamanhos dos vetores dos dois objetos são idênticos (linha 62). Nesse caso não é necessário realocar memória para o objeto `Array` que é operando esquerdo da atribuição. Se os tamanhos dos vetores forem diferentes, então `delete` é utilizado para liberar a memória alocada originalmente para o objeto `Array` de destino (operador do lado esquerdo da atribuição), copia o atributo `size` do objeto `Array` de origem (operando do lado direito da atribuição) para o objeto `Array` de destino, utiliza `new` para alocar memória para o objeto `Array` de destino e atribui o endereço de memória alocado para o seu ponteiro `ptr`. Nas linhas 69-70, os elementos do `Array` de origem são copiados para o `Array` destino, utilizando a estrutura de repetição `for`.

A função operadora `operator=` retorna o objeto atual (`*this`, linha 73), que deu origem à chamada da função (operando esquerdo da atribuição) como uma referência constante. Isso permite que sejam realizadas atribuições em cascata de objetos `Array` como em `x = y = z`.

Normalmente, um construtor de cópia, um destrutor e um operador de atribuição sobrecarregado são fornecidos em grupo para qualquer classe que utiliza memória dinamicamente alocada. Não fornecer um operador de atribuição e um construtor de cópia para uma classe quando os objetos dessa classe contêm ponteiros para memória dinamicamente alocada pode ser considerado um erro de lógica.

A linha 26 do arquivo `Array.hpp` declara a função operadora que sobrecarrega o operador de igualdade "==". Ao encontrar a expressão `integers1 == integers2` na linha 60 do arquivo `mainArray.cpp`, o compilador efetua a chamada `integers1.operador==(integers2)`. Nas linhas 78-88 do arquivo `Array.cpp`, podemos analisar a implementação dessa função operadora. Ela retorna `false` se os atributos dos objetos `Array` não forem iguais. Caso contrário, os elementos dos vetores de inteiros são comparados. Se todos eles foram iguais, `operator==` retorna `true`. O primeiro par de elementos que for diferente, faz com que a função operadora retorne `false`.

```
78  bool Array::operator==( const Array &right ) const
79  {
80      if ( size != right.size )
81          return false; // arrays com diferentes números de elementos
82
83      for ( int i = 0; i < size; i++ )
84          if ( ptr[ i ] != right.ptr[ i ] )
85              return false; // o conteúdo do Array não é igual
86
87      return true; // Arrays são iguais
88  } // fim da função operator==
```

As linhas 29-32 do arquivo `Array.hpp` definem o operador de desigualdade "`!=`" sobrecarregado para a classe `Array`. A função operadora `operator!=` utiliza a função `operator==` para verificar se um objeto `Array` é igual a outro e, então, retorna o oposto desse resultado.

```
28      // operador de desigualdade; retorna o oposto do operador ==
29      bool operator!=( const Array &right ) const
30      {
31          return ! ( *this == right ); // invoca Array::operator==
32      } // fim da função operator!=
```

As linhas 35 e 38 de `Array.hpp` declaram as funções operadoras que sobrecarregam o operador de subscrito. Ao encontrar a expressão `integers1[5]` da linha 64 do arquivo `mainArray.cpp`, o compilador invoca a função-membro sobrecarregada `operator[]` apropriada, gerando a chamada `integers1.operator[](5)`. O compilador cria uma chamada para a versão `const` de `operator[]`, linhas 107-118 do arquivo `Array.cpp` quando o operador de subscrito é utilizado em um objeto `const Array`. Por exemplo, se o objeto `const z` for instanciado com a instrução `const Array z(5)`, então a versão `const de operator[]` é

requerida para executar uma instrução como `cout << z[3] << endl`. Isto é importante, pois um objeto `const` somente pode efetuar chamadas de métodos `const` da classe.

```
107  int Array::operator[]( int subscript ) const
108  {
109      // verifica erro de subscrito fora do intervalo
110      if ( subscript < 0 || subscript >= size )
111      {
112          cerr << "\nError: indice " << subscript
113               << " fora do intervalo" << endl;
114          exit( 1 ); // termina o programa; subscrito fora do intervalo
115      } // fim do if
116
117      return ptr[ subscript ]; // retorna cópia desse elemento
118  } // fim da função operator[]
```

As duas definições de `operator[]` verificam se o subscrito (índice) recebido como argumento está no intervalo válido de acesso aos elementos do vetor de inteiros de um determinado objeto `Array`. Se não estiver, uma mensagem de erro é impressa e o programa termina com uma chamada de `exit` (cabeçalho `<cstdlib>`). Se o índice do vetor for válido, a versão não-`const` de `operator[]` retorna o elemento do vetor apropriado como uma referência para que ele possa ser utilizado como um *`lvalue`* modificável. Por outro lado, se o índice for válido, a versão `const` de `operator[]` irá retornar uma cópia do elemento apropriado do vetor. Neste caso, é retornado um valor que será utilizado como um *`rvalue`*.

REFERÊNCIAS

DEITEL, H. M.; DEITEL, P. J. **C++: como programar**. 5 ed. São Paulo: Pearson Prentice Hall, 2006.