UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 11 – Revisando Polimorfismo e Conhecendo a Sobrecarga de Operadores

1. REVISANDO POLIMORFISMO

Todas os dados que são envolvidos durante a execução de um programa nada mais são do que cadeias de bits para uma linguagem de máquina. Neste tipo de linguagem de zeros e uns, não há a ideia de tipos de dados. A abstração de tipagem surgiu nas linguagens de programação de alto nível, facilitando a organização dos dados e garantindo o tratamento apropriado para cada tipo. Com esta abstração é possível classificar previamente os dados a fim de evitar operações incoerentes, como somar um número real com uma letra. A verificação de tipos pode garantir que operandos de certo operador sejam de tipos compatíveis, evitando a realização de operações incoerentes, e podendo identificar erros já no processo de compilação, o que evita erros durante a execução do programa.

As linguagens de programação podem adotar as posturas de tipagem ditas:

- Fraca: somente alguns tipos de problemas são verificados (ex: C);
- Forte: verificação extensa de erros (ex: Java);
- Estática: verificações de tipagem são feitas somente em tempo de compilação (ex: C) ou durante a compilação e a execução (ex: C++, Java);
- Dinâmica: verificações de tipagem são feitas somente em tempo de execução (ex: Python);

```c
int par (int n){
    return (n % 2 == 0);
}

main(){
    int i = 3;
    par( i );
}
```

Considere o exemplo de código em C apresentado ao lado. Quais verificações estáticas de tipos podem ser realizadas? Considerando as ordens das operações, seria verificada se a constante `3` e a variável `i` são do mesmo tipo, viabilizando a atribuição. Na sequência seria verificado se o tipo da variável `i` é igual ao tipo do parâmetro esperado para a função `par()`. Também seria verificado se a variável `n` e a constante `2` são tipos válidos para a operação de resto (`%`), se o tipo do resultado da expressão `n % 2` é do mesmo tipo da constante 0 (zero), e se o resultado da expressão (`n%2 == 0`) é do tipo inteiro, uma vez que foi especificado que a função `par()` retorna um valor inteiro.

```python
def add(a, b):
    return a+b

res = add(5, 7)
print(res)
res = add("hello", " world")
print(res)
```

Agora considere um novo trecho de código escrito em Python, linguagem na qual as variáveis possuem tipagem dinâmica. Note que a função add() recebe dois parâmetros e realiza uma operação de "soma". Um programador acostumado com tipagem estática poderia inferir que necessariamente os parâmetros a e b devem ser números. Mas nesta situação uma string poderia naturalmente ser passadas como parâmetro para a

função também, sendo que o operador de soma realizaria uma concatenação das letras. Perceba que a mesma função permite receber diferentes tipos de parâmetros e mesmo assim funciona! Entretanto, se um objeto de um tipo (classe) que não possui a operação de soma definida for passado como parâmetro, ocorrerá um erro de execução.

Caso uma operação sem equivalência entre tipos fosse realizada, como por exemplo, multiplicar um inteiro com um caracter, o que uma linguagem de programação poderia fazer? As linguagens costumam utilizar uma das seguintes posturas:

- Intolerante: não permite que a operação seja realizada gerando um erro de compilação;
- Permissiva: permite que a operação seja realizada fazendo uma conversão implícita de tipos;
- Mista: permite que a operação seja realizada desde que seja possível fazer uma conversão implícita coerente ou definida em código (ex: int → long).

Muitas operações são dependentes dos tipos dos operandos, entretanto, existem também algumas operações que são inerentemente genéricas – independentes do tipo de valores manipulados – como por exemplo, pertinência, contingência, união e interseção (operações sobre conjuntos). Se uma linguagem de programação permite que estruturas de dados e algoritmos sejam construídos com a possibilidade de atuar sobre elementos de tipos diversos, podemos dizer que esta linguagem possui um sistema de tipos polimórfico. E quanto mais polimórfico for o sistema de tipos, maior a possibilidade de criar código reutilizável.

O polimorfismo tem relação com a possibilidade de criar um código que seja capaz de operar ou pelo menos aparentar operar sobre valores de tipos distintos. Existem quatro modalidades de Polimorfismo, divididas em dois grupos: o grupo de polimorfismo Adhoc (ou seja, "para uma determinada finalidade") e o grupo universal. Vejamos o que são estes tipos de polimorfismo.

TABELA 1 – Tipos de polimorfismo

Polimorfismo	**Adhoc**	**Coerção**
		Sobrecarga
	Universal	**Paramétrico**
		Inclusão

1.1. Polimorfismo Adhoc

O grupo de polimorfismo Adhoc pode ser entendido também como polimorfismo aparente, pois se criam abstrações de controle que operar sobre cada tipo admissível. Para quem utiliza uma estrutura com este tipo de polimorfismo, sem ver como ela foi implementada, tem a sensação de polimorfismo, ou seja, que a função ou operador pode operar sobre diversos tipos (mal sabe ele que alguém "implementou na unha" cada possível chamada com tipo distinto).

O primeiro tipo de polimorfismo Adhoc é a coerção, que nada mais é do que a conversão implícita ou explícita entre tipos diferentes (*casting*). Por exemplo, atribuir valores inteiros a números reais, ou então associar caracteres com inteiros, ou números negativos com valores sem sinal. O segundo tipo de polimorfismo Adhoc é a sobrecarga, que permite que funções ou operadores possam ser usados para designar duas ou mais operações distintas. Isto é, permite que mesmo operador, ou uma função/método de mesmo nome, seja implementado de diversas formas, desde que os tipos de seus operandos, ou tipos dos parâmetros, sejam diferentes – ou seja, a

combinação de parâmetros não é ambígua. Já vimos em aulas passadas como é possível implementar métodos polimórficos no Java, mas a esta mesma linguagem não aceita a sobrecarga de operadores. Por esse motivo, na segunda seção desta material, vamos ver como podemos sobrecarregar operadores na linguagem de programação C++.

1.2. Polimorfismo Universal

O grupo de polimorfismo Universal é a forma mais elegante de polimorfismo, também chamado de polimorfismo verdadeiro, pois permite que um trecho de código possa ser executar e atuar sobre elementos de diferentes tipos. Para que este tipo de polimorfismo seja possível é necessário o mecanismo de ligação tardia, já mencionado em aulas anteriores, pelo qual o compilador gera código para calcular qual método deve ser chamado em tempo de execução.

O primeiro tipo de polimorfismo Universal é o polimorfismo paramétrico, pelo qual há uma parametrização das estruturas de dados e funções em relação ao tipo de elementos sobre o qual operam. O uso deste tipo de polimorfismo gera a aparência de que um novo tipo de dado genérico é definido e que pode ser substituído por diversos tipos ou classes existentes, de acordo com a necessidade. Já o segundo tipo de polimorfismo Universal é o polimorfismo por inclusão, que tem relação com os mecanismos de herança já estudados em aulas anteriores. Lembrando que este recurso permite o uso de atributos e métodos de superclasses e ainda a redefinição de métodos (escrever um método especializado com a mesma assinatura).

2. SOBRECARGA DE OPERADORES

Toda linguagem de programação possui uma definição sobre o que deve ser feito ao se utilizar um operador (ex: +, -, *, /, ^) sobre determinados tipos. De fato, os operadores podem ser entendidos como funções que recebem um ou mais parâmetros. Por exemplo, considere uma soma entre dois inteiros realizada pelo operador +. É comum que ela seja escrita como x + y. Note que este operador é binário, ou seja, atua sobre dois operandos. Assim, tal operação poderia ser definida na forma de função como: `operador+ (int var1, int var2){...}`. Além de operações binárias, também existem as operações unárias, nas quais o símbolo operador atua apenas sobre um único elemento de um dado tipo; por exemplo, o operador - unário, que transforma um número positivo em um valor negativo: `operador- (int var1){...}`.

Apesar das linguagens de programação implementarem as operações básicas entre os diversos tipos primitivos, qualquer estrutura nova, por padrão, não terá os operadores básicos definidos. Neste sentido as linguagens costumam utilizar duas abordagens possíveis: permitir somente a definição de métodos, impossibilitando o uso de operadores, ou, permitir a definição de métodos ordinários e também a definição de métodos para os operadores. A linguagem Java adota a primeira abordagem, por isso não podemos usar tal linguagem para realizar sobrecarga de operadores. A linguagem C++, por outro lado, permite tal prática. Por esse motivo, vamos utilizar C++ para a parte prática desta aula.

2.1. Definição e Sobrecarga de Operadores em C++

Considere a definição apresentada no quadro abaixo de uma classe que representa uma abstração de um ponto bidimensional escrita na linguagem de programação C++. Note que os conceitos de orientação a objetos vistos e aplicados em Java são relativamente semelhantes à forma de escrita em C++. Foram definidos dois atributos privados x e y que representam as coordenadas do ponto, dois métodos construtores públicos – um que não recebe parâmetros e outro que recebe a posição inicial dos pontos – um método para definir/alterar as coordenadas e outros dois métodos para resgatar os valores de x e y, uma vez que tais atributos são privados.

Arquivo: sobrecarga.cpp

```
 1.  class Ponto2D{
 2.      private:
 3.          int x, y;
 4.
 5.      public:
 6.          Ponto2D(){
 7.              setCoordinate(0,0);
 8.          }
 9.          Ponto2D(int x, int y){
10.              setCoordinate(x, y);
11.          }
12.          void setCoordinate(int x, int y){
13.              this->x = x;
14.              this->y = y;
15.          }
16.          int getX(){ return x; }
17.          int getY(){ return y; }
18.  };
```

Considerando a classe Ponto2D apresentada acima, podemos criar objetos desta classe com o operador *new*, de forma similar ao que é feito em Java. Entretanto, em ambas as linguagens uma operação de soma (+) entre dois objetos Ponto2D não seria permitida:

Arquivo: sobrecarga.cpp (atualização)

```
21.  int main(){
22.      Ponto2D *pa = new Ponto2D();
23.      Ponto2D *pb = new Ponto2D(5,9);
24.      pa->setCoordinate( 2 , 6 );
25.      Ponto2D *pc = *pa + *pb;
26.      return 0;
27.  }
```

Ao tentar compilar o código acima receberíamos um erro apresentado pela seguinte mensagem: *error: no match for 'operator+' (operand types are 'Ponto2D' and 'Ponto2D')*. Esta mensagem de erro informa que não foi encontrada nenhuma definição para o operador + ser utilizado entre dois objetos do tipo Ponto2D. Ou seja, para poder utilizar tal operador da forma que foi codificada no quadro acima, precisamos definir como o operador + deve funcionar.

Para exemplificar a definição do operador + para a classe Ponto2D, considere que ao utilizar tal operador, queremos somar as coordenadas do primeiro ponto com as coordenadas do segundo ponto e retornar um novo ponto resultante. Dessa maneira, poderíamos definir tal operador como:

Arquivo: sobrecarga.cpp (atualização)

```
 1.  class Ponto2D{
         ...
 5.      public:
             ...
12.          Ponto2D* operator+( Ponto2D point ){
13.              Ponto2D* result = new Ponto2D();
14.              int nx = this->x + point.x;
15.              int ny = this->y + point.y;
16.              result->setCoordinate( nx , ny );
17.              return result;
18.          }
             ...
27.  };
```

Perceba pela definição acima que estamos definindo a implementação de um método, como qualquer outro. A diferença fundamental está no seu nome: operator+ o que simboliza o operador +. Como nosso método é referente a um objeto Ponto2D e recebe um parâmetro de entrada, trata-se de uma operação binária utilizando o operador especificado. Para especificar um operador unário, o método vinculado ao operador não pode receber parâmetros. Exemplificando, considere agora a nova definição do operador + unário que aumenta em um as coordenadas de um ponto:

Arquivo: sobrecarga.cpp (atualização)

```
 1.  class Ponto2D{
         ...
 5.      public:
             ...
21.          void operator+( ){
22.              setCoordinate( getX()+1 , getY()+1 );
23.          }
             ...
30.  };

31.  int main(){
         ...
32.      +(*pa); //uso do operador unário +
         ...
43.  }
```

Note que na definição do método do operador unário + o próprio ponto (*this*) está sendo alterado, não havendo um valor de retorno do método. O uso deste novo operador deve ser feito adicionando o operador na frente (lado esquerdo) do objeto ponto. No exemplo, como nosso Ponto2D pa é um ponteiro, precisamos adicionar o símbolo * para resgatar primeiramente a posição (ou o valor) do objeto antes de utilizar o operador unário +.

Como os operadores nada mais são do que métodos, pode-se escrevê-los com sobrecarga. Ou seja, é possível escrever diversos métodos para cada operador desde que a assinatura deles não seja idêntica. Para exemplificar, considere que gostaríamos de escrever uma nova definição de operação de soma entre um Ponto2D e um inteiro, na qual o valor inteiro é acrescentado ao valor de ambas às coordenadas do ponto, alterando seu valor.

ATENÇÃO: nem todo símbolo é um possível operador. Isto pode variar entre linguagens. Em C++ podemos utilizar pelo menos os operadores: +, -, /, *, &, %, ~ e ^.

Arquivo: sobrecarga.cpp (atualização)

```
 1.  class Ponto2D{
         ...
 5.      public:
             ...
21.          void operator+( int value ){
22.              int nx = this->x + value;
23.              int ny = this->y + value;
24.              this->setCoordinate( nx , ny );
             }
34.          ...
     };
37.
     int main(){
43.      ...
44.                  //*pa = [x:2 / y: 6]  (valor antigo)
         (*pa) + 5; //*pa = [x:7 / y: 11] (valor novo)
47.      ...
     }
```

Exercícios

1) Crie uma classe chamada Matriz3x3 em C++ que será utilizada para representar e realizar operações envolvendo matrizes com 3 linhas e 3 colunas. Crie dois construtores, um que não recebe valores de parâmetros e inicializa a matriz 3x3 como uma matriz identidade (diagonal principal preenchida com uns e demais elementos com zeros), e outro que recebe um vetor com nove elementos, que devem ser usados para preencher a matriz de cima para baixo, da esquerda para a direita. Faça com que os elementos da matriz não possam ser acessados ou modificados diretamente fora da classe.

Implemente ainda um método *at()* que recebe dois inteiros como parâmetro, linha e coluna e retorna qual o valor presente na posição matriz[linha][coluna]. Implemente também um método *set()* que recebe três valores como parâmetro, linha, coluna e valor, o qual altera o valor da matriz na linha e coluna especificada. Um método *show()* também deverá ser implementado, de forma que este mostre em tela o objeto na forma matricial; ou seja, cada elemento na sua posição (linha/coluna).

Por fim, implemente e sobrecarregue os operadores permitindo as operações de:

- Soma entre matrizes: somar elemento a elemento duas matrizes. Retorna uma nova matriz;
- Subtração entre matrizes: subtrair elemento a elemento. Retorna uma nova matriz;
- Inversão de sinais da matriz: operador - unário inverte os sinais de todos os elementos;
- Multiplicação entre matriz e constante: multiplicar uma constante sobre todos os valores da matriz. Retorna uma nova matriz.
- Multiplicação entre matrizes: multiplicar uma matriz com outra matriz (lembre-se na multiplicação entre matrizes NÃO se multiplica elemento a elemento!)
- Potenciação de uma matriz por uma constante: realizar a potenciação de uma matriz baseado em um valor inteiro que representa a potencia.

De modo resumido, seu programa deve permitir que as seguintes operações sejam realizadas conforme mostrado abaixo (na ordem das operações apresentadas acima):

```
Matriz3x3 a, b, c;
c = a + b;
c = a – b;
c = -a;
c = 3*a;
c = a * b;
c = a ^ 3;  // a*a*a
```