13. SOBRECARGA DE OPERADORES DE I/O

Todos os exemplos apresentados até o momento usam instruções de fluxo de entrada e de saída de dados para ler e exibir valores. Entretanto, os fluxos não são parte da linguagem C++, mas são implementados como classes da biblioteca C++. As declarações para essas classes ficam armazenadas no arquivo *header* "iostream.h". As instruções de fluxos de entrada e saída de dados podem ser interceptadas para as classes criadas pelo usuário. Sobrepondo os operadores de fluxo de entrada e saída de dados, torna-se possível "ensinar" ao C++ como lidar com instruções de fluxo que incluam qualquer tipo de instância de classe ou para qualquer tipo definido pelo usuário.

Normalmente, fluxos de saída podem lidar apenas com tipos de dados simples, tais como *int*, *float* e *double*. Entretanto, com a sobreposição do operador "<<" de fluxo de saída, pode-se facilmente acrescentar as classes criadas aos tipos de dados que instruções de fluxo de saída são projetadas para usar.

O próximo exemplo apresenta um programa típico que declara uma classe denominada "ponto" que armazena dois valores inteiros, "x" e "y", que representam os valores da coordenada de uma posição. Normalmente, para exibir os valores de campos de dados privados como "x" e "y", deveria-se chamar funções membro como "getx" e "gety". Porém, acrescentando-se a classe "ponto" àquelas com as quais os fluxos de saída podem lidar, torna-se possível exibir variáveis "ponto" sem todo esse problema.

```
# include <iostream>
using namespace std;
class ponto
{
        private:
                int x, y;
        public:
                ponto () { x = y = 0; }
                ponto (int xx, int yy) { x = xx; y = yy; }
                void putx (int xx) { x = xx; }
                void puty (int yy) { y = yy; }
                int getx (void) { return x; }
                int gety (void) { return y; }
                friend ostream& operator<< (ostream& os, ponto &p);
};

void main ()
{
        ponto p;
        cout << p << "\n";
        p.putx(100);
        p.puty(200);
        cout << p << "\n";
}

ostream& operator<< (ostream& os, ponto &p)
{
        os << "x == " << p.x << ", y == " << p.y;
        return os;
}
```

A função *friend* sobrepõe o operador de fluxo de saída <<. Esta função retorna uma referência para "ostream", que é uma das classes definidas em "iostream.h". Dois parâmetros também são listados para a função sobreposta: "os", uma referência a "ostream", e "p", uma referência a uma instância "ponto". Em outras classes pode-se usar esse mesmo formato, neste caso, substituindo-se "ponto" pelo próprio nome da classe. Os outros elementos permanecem os mesmos. Quando o programa for executado, serão exibidos na tela os seguintes resultados:

```
x == 0, y == 0
x == 100, y == 200
```

Como a função *friend* "operator<<" retorna uma referência a "ostream", também é possível alinhar vários usos do operador de fluxo de saída de dados. Por exemplo, se "p1", "p2" e "p3" forem instâncias do tipo "ponto", pode-se exibir seus valores com uma única instrução

```
cout << p1 << "; " << p2 << "; " << p3;
```

Sobrepor o operador ">>" de fluxo de entrada de dados é um processo semelhante à sobreposição de fluxos de saída de dados. Uma função de fluxo sobreposta efetivamente ensina ao C++ como ler instâncias de um tipo de classe específico. A seguir, é acrescentada a característica de fluxo de entrada de dados ao programa do exemplo anterior.

```
# include <iostream>
using namespace std;
class ponto
{
        private:
                int x, y;
        public:
                ponto () { x = y = 0; }
                ponto (int xx, int yy) { x = xx; y = yy; }
                void putx (int xx) { x = xx; }
                void puty (int yy) { y = yy; }
                int getx (void) { return x; }
                int gety (void) { return y; }
                friend ostream& operator<< (ostream& os, ponto &p);
                friend istream& operator>> (istream& is, ponto &p);
};
void main ()
{
        ponto p;
        cout << p << "\n";
        p.putx(100);
        p.puty(200);
        cout << p << "\n";
        cout << "\n Digite valores para x e para y: ";
        cin >> p;
        cout << "\n Voce digitou: " << p;
}

ostream& operator<< (ostream& os, ponto &p)
{
        os << "x == " << p.x << ", y == " << p.y;
        return os;
}

istream& operator>> (istream& is, ponto &p)
{
        is >> p.x >> p.y;
        return is;
}
```

A função *friend* de fluxo de entrada de dados, sobrepõe o operador ">>" para a classe "ponto". Exceto pela referência a "istream" e pelo resultado da função, a função de fluxo de entrada de dados é semelhante à função de saída. A nova implementação da função lê valores para "x" e "y" através de parâmetros de referência do tipo "istream" denominado "is". Depois disso, uma instrução "return" retorna "istream" de modo que instruções de entrada de dados possam ficar alinhadas [4].

Também é simples de se definir um operador de saída para tipos definidos pelo usuário. Por exemplo, suponha que a classe "clock_time" tenha sido definida da seguinte maneira:

```
struct clock_time {
        int h;
        int m;
        int s;
}
```

Se for declarada a variável "t"

```
clock_time t = {11, 45, 20}
```

e deseja-se que o comando

```
cout << t;
```

apresente na tela 11:45:20, pode-se definir então um operador de saída para o objeto "clock_time" [3]:

```
ostream& operator<< (ostream& c, clock_time t)
{
        c << t.h << ":";
        c << t.m << ":";
        c << t.s;
        return c;
}
```

Muitas discussões entre profissionais têm considerado a necessidade ou não da sobrecarga. De um lado, há os programadores mais conservadores que acreditam ser benéfico ao C++ manter o "enxugamento" das definições C. De outro, programadores preocupados com a possibilidade de reutilização de programas afirmam que sobrecarregar operadores é uma forma mais adequada para responder questões do tipo "Como criar programas de fácil manutenção e utilização?".

De fato, a sobrecarga vem ao encontro dos preceitos da Programação Orientada a Objetos de aproximar o máximo possível os dados e a ação que se procede sobre eles. Porém, quando este recurso é usado excessiva e irrestritamente, pode tornar o código obscuro. Uma boa política é sempre colocar comentários nos arquivos com a definição da classe, explicando como o operador funciona quando aplicado ao seus objetos [9].