## Sobrecarga de operadores

Creio que todos vocês, programando na linguagem C, já tentaram, ao menos uma vez, atribuir com o operador = um vetor a outro, ou comparar duas strings com o operador ==.

Realmente, seria ótimo se estes operadores pudessem ter estas funcionalidades.

A linguagem C++ possibilita a sobrecarga de operadores, em outras palavras, possibilita que novas funcionalidades possam ser atribuídas a operadores existente.

Antes de nos aprofundarmos neste conceito, vamos visualizar que na linguagem C já existiam operadores sobrecarregados.

Você, certamente, já tirou proveito do fato do operador / ser utilizado na aritmética de inteiros e na aritmética de ponto flutuante.

## Sobrecarga de operadores

Os operadores sobrecarregados pressupõem uma análise do contexto em que ocorre sua utilização determinando-se qual comportamento será adotado.

Na linguagem C++ também temos exemplo de operadores sobrecarregados. Por exemplo, (<<) o operador de inserção de fluxo. Que no exemplo abaixo tem outra funcionalidade.

```
#include <iostream>
using std::cout;
int main()
{
   int i=1;
   i <<= 1;
   cout << i;
}
```

# Linguagem de Programação C++

Inicialmente, devemos ter em mente que C++ não permite que se crie novos operadores, e sim possibilita que se sobrecarregue operadores existentes. Nem todos operadores podem ser sobre carregados. Porém, a grande maioria pode.

Operadores passíveis de sobrecarga:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | % | ^ | & | \| | ~ | ! | = |
| < | > | += | -= | *= | /= | %= | ^= | &= | \|= | << |
| >> | >>= | <<= | == | != | <= | >= | && | \|\| | ++ | -- |
| ->* | , | -> | [] | () | new | | delete | | new[] | |
| delete[] | | | | | | | | | | |

Operadores não passíveis de sobrecarga:

. .* :: ?:

Para utilizar um operador sobre objetos de classe este deve ser sobrecarregado. Existem três exceções: (=), (&) e (,).

Observações pertinentes:

- A precedência de um operador não pode ser alterada pela sobrecarga. Entretanto, parênteses podem ser utilizados para forçar uma ordem de avaliação de operadores sobrecarregados em uma expressão;

- A associatividade de um operador não pode ser alterada pela sobrecarga;

- Não é possível alterar a 'aridade' de um operador;

- A sobrecarga de operadores funciona apenas com tipos definidos pelo usuário ou ao menos com a combinação de um tipo definido pelo usuário e um tipo fundamental.

Podemos ter funções operadoras implementadas como funções membros ou como funções globais.

Quando implementadas como funções membros o operador a esquerda em operadores binários e o único operador em operadores unários é sempre um objeto de classe (o *this* é utilizado).

Já, em uma implementação como função global ambos os operandos devem ser listados.

Os operadores (), [], -> ou qualquer um dos operadores de atribuição, devem ser sobrecarregados obrigatoriamente como funções membro.

Quando uma função operadora global necessitar manipular diretamente membros privados ou protegidos da classe, esta deve ser especificada com *friend* da classe.

A sobrecarga dos operadores << e >> deve ser feita obrigatoriamente através de funções globais.

Para garantimos a comutatividade na sobrecarga de um operador esta deve ser feita através de uma função global.

Os primeiros operadores que vamos sobrecarregar são os operadores de inserção e extração de fluxo.

Inicialmente deve-se entender que **<<** é uma função já previamente sobrecarregada, do tipo **ostream X outroTipo -> ostream** e que a expressão **x << expr** tem como valor o próprio canal **x**.

É isso que permite fazer **x << expr1 << expr2 << expr3 ...**. Pois, as expressões de tipos diversos: **x << expr1** produz o próprio **x** como resultado, o qual recebe a mensagem **<< expr2**, etc.

Para se efetuar a sobrecarga de um operador utiliza-se o quantificador **operator.**

A forma geral do operador << sobrecarregado é a seguinte:

```
ostream &operator<<(ostream &out, novaClasse &objeto)
{
  //corpo da função
  return out;
}
```

Para uma melhor compreensão analisaremos o exemplo da sobrecarga do operador **<<** na classe Data.

```
//conteúdo do arquivo dataSobrecargaOperador.h
#ifndef DATASOBRECARGAOPERADOR_H
#define DATASOBRECARGAOPERADOR_H
  #include <iostream>
  class DataSobrecargaOperador {
    friend ostream &operator<<(ostream &, DataSobrecargaOperador &);
    public:
      DataSobrecargaOperador(int=1, int=1, int=1900);
      void setDia(int);
      void setMes(int);
      void setAno(int);
      int getDia();
      int getMes();
      int getAno();
    private:
      int dia;
      int mes;
      int ano;
      int verificaDia(int);
  };
#endif
```

```
//conteúdo do arquivo dataSobrecargaOperador.cpp
#include <iostream>
using namespace std;
#include "dataSobrecargaOperador.h"
ostream &operator<<(ostream &out, DataSobrecargaOperador &data)
{
   out << endl << data.getDia() << '/' << data.getMes() << '/' << data.getAno()
<< endl;
   return out;
}
DataSobrecargaOperador::DataSobrecargaOperador(int d, int m, int a)
{
   setMes(m);
   setAno(a);
   setDia(d);
}
void DataSobrecargaOperador::setDia(int d)
{
   dia=verificaDia(d);
}
void DataSobrecargaOperador::setMes(int m)
{
   if (m>0 && m<=12)
      mes=m;
   else
```

```
  {
    mes=1;
    cout << endl << "Mes invalido (" << m << ") setado para 1." << endl;
  }
}
void DataSobrecargaOperador::setAno(int a)
{
  if (a>=1900 && a<2011)
    ano=a;
  else
  {
    ano=1900;
    cout << endl << "Ano invalido (" << a << ") setado para 1900." << endl;
  }
}
int DataSobrecargaOperador::getDia()
{
  return dia;
}
int DataSobrecargaOperador::getMes()
{
  return mes;
}
```

```
int DataSobrecargaOperador::getAno()
{
   return ano;
}
int DataSobrecargaOperador::verificaDia(int diaTeste)
{
   static const int diasPorMes[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
   if (diaTeste>0 && diaTeste<=diasPorMes[getMes()])
      return diaTeste;
   if (getMes()==2 && diaTeste==29 && (getAno()%400==0 ||
(getAno()%4==0 && getAno()%100!=0)))
      return diaTeste;
   cout << endl << "Dia invalido (" << diaTeste << ")setado para 1." << endl;
   return 1;
}

//conteúdo do arquivo PrincipaldataSobrecargaOperador.cpp
#include <iostream>
#include "dataSobrecargaOperador.h"
using namespace std;
int main() {
    DataSobrecargaOperador data(21,12,2010);
    cout << data;
    return 0;
}
```

**Exercício:**

A sobrecarga do operador de extração de fluxo ocorre de forma muito similar. No entanto, o objeto utilizado é o **istream**.

Sendo assim, sobrecarregue o operador **>>** para a classe DataSobrecargaDeOperador e utilize-a em um programa *drive*.

```
//conteúdo do arquivo dataSobrecargaOperador.h
#ifndef DATASOBRECARGAOPERADOR_H
#define DATASOBRECARGAOPERADOR_H
  #include <iostream>
  class DataSobrecargaOperador {
    public:
      DataSobrecargaOperador(int=1, int=1, int=1900);
      void setDia(int);
      void setMes(int);
      void setAno(int);
      int getDia();
      int getMes();
      int getAno();
      friend ostream &operator<<(ostream &, DataSobrecargaOperador &);
      friend istream &operator>>(istream &, DataSobrecargaOperador &);
    private:
      int dia;
      int mes;
      int ano;
      int verificaDia(int);
  };
#endif
```

```
//conteúdo do arquivo dataSobrecargaOperador.cpp
#include <iostream>
using namespace std;
#include "dataSobrecargaOperador.h"
ostream &operator<<(ostream &out, DataSobrecargaOperador &data)
{
   out << endl << data.getDia() << '/' << data.getMes() << '/' << data.getAno()
<< endl;
   return out;
}
istream &operator>>(istream &in, DataSobrecargaOperador &data)
{
   int d, m, a;
   cout << "Dia: ";
   in >> d;
   cout << "Mes: ";
   in >> m;
   cout << "Ano: ";
   in >> a;
   data.setMes(m);
   data.setAno(a);
   data.setDia(d);
   return in;
}
```

## Sobrecarga de operadores

Vamos analisar agora, a sobrecarga de operadores unários e binários.

Ambos podem ser sobrecarregados com uma função membro não-*static* sem argumentos ou com uma função global com um (no caso dos unários) ou com dois (no caso dos binário) argumentos.

Para uma melhor compreensão vamos analisar um exemplo de implementação de uma classe denominada Vetor, na qual sobrecarregaremos diversos operadores.

```
//conteúdo do arquivo vetor.h
#ifndef VETOR_H
#define VETOR_H
  #include <iostream>
  using std::ostream;
  using std::istream;
  class Vetor {
    friend ostream &operator<<( ostream &, Vetor &);
    friend istream &operator>>( istream &, Vetor &);
    public:
      Vetor( int = 10 ); // construtor padrão
      Vetor(Vetor & ); // construtor de cópia
      ~Vetor(); // destrutor
      int getTamanho(); // retorna tamanho
      Vetor &operator=(Vetor &); // operador de atribuição
      bool operator==(Vetor &); // operador de igualdade
      bool operator!=( Vetor &right)// operador de desigualdade
      // retorna o oposto do operador ==
      {
        return ! ( *this == right ); // invoca Array::operator==
      }
      int &operator[]( int ); // operador de subscrito de objetos
    private:
      int tamanho; // tamanho do vetor baseado em ponteiro
      int *ptr; // ponteiro para o primeiro elemento do vetor baseado em
      // ponteiro
  };
#endif
```

```
//conteúdo do arquivo vetor.cpp
#include <iostream>
using namespace std;
#include <iomanip>
#include <cstdlib> // sai do protótipo de função
#include "vetor.h" // definição da classe Vetor
Vetor::Vetor(int vetorTamanho)// construtor padrão para a classe Vetor
{
   tamanho = ( vetorTamanho > 0 ? vetorTamanho : 10 );
   // valida vetorTamanho
   ptr = new int[ tamanho ]; // cria espaço para vetor baseado em ponteiro
   for ( int i = 0; i < tamanho; i++ )
      ptr[ i ] = 0; // configura elemento do vetor baseado em ponteiro
}
Vetor::Vetor( Vetor &vetorParaCopiar ):tamanho(
vetorParaCopiar.tamanho )//construtor de cópia da classe Vetor
{
   ptr = new int[ tamanho ]; // cria espaço para vetor baseado em ponteiro
   for ( int i = 0; i < tamanho; i++ )
      ptr[ i ] = vetorParaCopiar.ptr[ i ]; // copia para o objeto
}
Vetor::~Vetor()// destrutor para a classe Vetor
{
   delete [] ptr; // libera espaço do vetor baseado em ponteiro
}
```

```
int Vetor::getTamanho() // retorna o número de elementos do vetor
{
   return tamanho; // número de elementos no vetor
}
Vetor &Vetor::operator=(Vetor &right ) /*operador de atribuição
Sobrecarregado */
{
   if ( &right != this ) // evita auto-atribuição
   {
      if ( tamanho != right.tamanho )/* para vetores de tamanhos diferentes,
desaloca o vetor do lado esquerdo original, então aloca o novo vetor à
esquerda */
      {
         delete [] ptr; // libera espaço
         tamanho = right.tamanho; // redimensiona esse objeto
         ptr = new int[ tamanho ]; // cria espaço para a cópia do vetor
      }
      for ( int i = 0; i < tamanho; i++ )
         ptr[ i ] = right.ptr[ i ]; // copia o vetor para o objeto
   }
   return *this; // permite x = y = z, por exemplo
}
```