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

int DataSobrecargaOperador::operator- ()
{
   DataSobrecargaOperador data(1,1,getAno());
   return (*this)-data;
}
```

```
int DataSobrecargaOperador::operator- (DataSobrecargaOperador &data)
{
  static const int diasPorMes[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
  int dias1=0, dias2=0;
  for (int i=1900; i<getAno(); i++)
    dias1+=(i%400==0 || (i%4==0 && i%100!=0))?366:365;
  for (int i=1; i<getMes(); i++)
    dias1+=(i==2 && (getAno()%400==0 || (getAno()%4==0 &&
getAno()%100!=0)))?29:diasPorMes[i];
  dias1+=getDia();
  for (int i=1900; i<data.getAno(); i++)
    dias2+=(i%400==0 || (i%4==0 && i%100!=0))?366:365;
  for (int i=1; i<data.getMes(); i++)
    dias2+=(i==2 && (data.getAno()%400==0 || (data.getAno()%4==0 &&
data.getAno()%100!=0)))?29:diasPorMes[i];
  dias2+=data.getDia();
  return dias1-dias2;
}
```

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

```
    private:
      int dia;
      int mes;
      int ano;
      int verificaDia(int);
      static const int numeroDeDiasNoMes[]; // vetor de dias por mês
      void incremento();
      bool fimDoMes( int );
      bool anoBissexto( int );
  };
#endif
```

```
//conteúdo do arquivo dataSobrecargaOperador.cpp

#include <iostream>

using namespace std;

#include "dataSobrecargaOperador.h“

// inicializa membro static no escopo de arquivo
const int DataSobrecargaOperador::numeroDeDiasNoMes[] =
  { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

int DataSobrecargaOperador::operator- ()
{
  DataSobrecargaOperador data(1,1,getAno());
  return (*this)-data;
}
```

```
int DataSobrecargaOperador::operator- (DataSobrecargaOperador &data)
{
  int dias1=0, dias2=0;
  for (int i=1900; i<getAno(); i++)
    dias1+= anoBissexto(i)?366:365;
  for (int i=1; i<getMes(); i++)
    dias1+=(i==2 && (anoBissexto(getAno())))?29:numeroDeDiasNoMes[i];
  dias1+=getDia();
  for (int i=1900; i<data.getAno(); i++)
    dias2+= anoBissexto(i)?366:365;
  for (int i=1; i<data.getMes(); i++)
    dias2+=(i==2 && (anoBissexto(data.getAno())))?29:
numeroDeDiasNoMes[i];
  dias2+=data.getDia();
  return dias1-dias2;
}
```

Sobrecarga de operadores

Trataremos agora de um último detalhe sobre o tópico sobrecarga de operadores.

Vimos, que a escolha sobre qual implementação de um determinado operador sobrecarregado é feita com base na assinatura da função que o implementa. Contudo, como esta diferenciação se dará, por exemplo, no caso do operador ++ que poder ser pré ou pós-fixado?

A implementação prefixada se dará da mesma forma que fazemos a sobrecarga dos demais operadores unários. O problema reside na implementação pós-fixada.

Sobrecarga de operadores

Possíveis protótipos para o operador ++ prefixado seriam:

nomeDaClasse &operator++(); //função-membro

nomeDaClasse &operator++(nomeDaClasse &); //função global

A solução adotada na linguagem C++, para possibilitar que o compilador diferencie a implementação pós-fixada da prefixada, foi a inserção de um parâmetro extra. Ou seja, gerando os seguintes protótipos:

nomeDaClasse &operator++(int); //função-membro

nomeDaClasse &operator++(nomeDaClasse &, int); //função global

Para uma melhor fixação vamos analisar um exemplo de utilização da sobrecarga do operador ++ pós e prefixado.

Observação: Tudo que está sendo dito sobre o operador de incremento (++) vale para o operador de decremento (--).

Vamos imaginar a aplicação do operador de incremento sobre um objeto da classe DataSobrecargaOperador, o qual geraria o incremento de um dia na data em quentão.

```
//conteúdo do arquivo dataSobrecargaOperador.h
#ifndef DATASOBRECARGAOPERADOR_H
#define DATASOBRECARGAOPERADOR_H
  #include <iostream>
  class DataSobrecargaOperador {
      friend DataSobrecargaOperador operator+ (int,
DataSobrecargaOperador &);
      friend DataSobrecargaOperador operator+
(DataSobrecargaOperador &, int);
      friend ostream &operator<<(ostream &, DataSobrecargaOperador &);
      friend istream &operator>>(istream &, DataSobrecargaOperador &);
    public:
      DataSobrecargaOperador(int=1, int=1, int=1900);
      void setDia(int);
      void setMes(int);
      void setAno(int);
      int getDia();
      int getMes();
      int getAno();
      int operator- ();
      int operator- (DataSobrecargaOperador &);
      // operador de incremento pré-fixado
      DataSobrecargaOperador &operator++();
```

```
      // operador de incremento pós-fixado
      DataSobrecargaOperador operator++( int );
   private:
      int dia;
      int mes;
      int ano;
      int verificaDia(int);
      static const int numeroDeDiasNoMes[]; // vetor de dias por mês
      void incremento();
      bool fimDoMes( int );
      bool anoBissexto( int );
  };
#endif

//conteúdo do arquivo dataSobrecargaOperador.cpp
...
// operador de incremento pré-fixado sobrecarregado
DataSobrecargaOperador &DataSobrecargaOperador::operator++()
{
  incremento(); // incrementa data
  return *this; // retorno de referência para criar um lvalue
}
```

```
// operador de incremento pós-fixado sobrecarregado;
// observe que o parâmetro fictício do tipo inteiro não tem nem ao menos
// um nome (identificador) de parâmetro
DataSobrecargaOperador DataSobrecargaOperador::operator++( int )
{
   DataSobrecargaOperador temp = *this; // armazena o estado atual do
objeto
   incremento();
   // retorna o objeto temporário, salvo, não-incrementado
   return temp; // retorno de valor; não um retorno de referência
}
...
```

Linguagem de Programação C++

Com base no que estudamos é possível perceber que podemos definir uma classe String com inúmeras funcionalidades interessantes, com:

- possibilidade de atribuir uma string a outra com o operador =;

- possibilidade de comparar duas strings com o operador ==;

- possibilidade de concatenar duas string com o operador +;

- possibilidade de alocar dinamicamente apenas memória necessária para armazenar a string;

- etc...

Podemos considerar a implementação desta classe String como um ótimo exercício de revisão.

Devido à relevância da existência de uma classe String com tais características, a linguagem C++ disponibiliza esta classe em sua biblioteca-padrão.

A classe String é definida no arquivo cabeçalho <string> e pertence ao namespace std.

Ao invés de listar as mensagens que compõem a interfase da classe String, vamos analisar um exemplo que explora as principais características desta.

```
// Programa que se utiliza da classe string da biblioteca padrão
#include <iostream>
using std::cout;
using std::endl;
#include <string>
using std::string;
int main()
{
   string s1( "feliz" );
   string s2( " aniversario" );
   string s3;
   // testa operadores de igualdade e relacionais sobrecarregados
   cout << "s1 eh \"" << s1 << "\"; s2 eh \"" << s2
      << "\"; s3 eh \"" << s3 << '\"'
      << endl << "O resultado da comparação entre s2 e s1:"
      << endl << "s2 == s1 produz " << ( s2 == s1 ? "verdadeiro" : "falso" )
      << endl << "s2 != s1 produz " << ( s2 != s1 ? "verdadeiro" : "falso" )
      << endl << "s2 >  s1 produz " << ( s2 > s1 ? "verdadeiro" : "falso" )
      << endl << "s2 <  s1 produz " << ( s2 < s1 ? "verdadeiro" : "falso" )
      << endl << "s2 >= s1 produz " << ( s2 >= s1 ? "verdadeiro" : "falso" )
      << endl << "s2 <= s1 produz " << ( s2 <= s1 ? "verdadeiro" : "falso" );
```

```
  // testa a função membro string vazia
  cout << "\n\nTestanto s3.empty():" << endl;
  if ( s3.empty() )
  {
     cout << "s3 eh vazio; atribuindo s1 para s3;" << endl;
     s3 = s1; // atribui s1 a s3
     cout << "s3 agora eh \"" << s3 << "\"";
  }
  // testa operador de concatenação de string sobrecarregado
  cout << "\n\ns1 += s2 produz s1 = ";
  s1 += s2; // testa a concatenação sobrecarregada
  cout << s1;
  // testa operador de concatenação de string sobrecarregado com string
no estilo C
  cout << endl << endl << "s1 += \" para voce\" produz" << endl;
  s1 += " para voce";
  cout << "s1 = " << s1 << endl << endl;
  // testa função membro string substr
  cout << "A substring de s1 inicializando na localizacao 0 para"
     << endl << "14 caracteres, s1.substr(0, 14), eh:" << endl
     << s1.substr( 0, 14 ) << endl << endl;
```

```
// testa a opção de substr "to-end-of-string"
cout << "A substring de s1 inicalizando na" << endl
   << "localizacao 15, s1.substr(15), eh:\n"
   << s1.substr( 15 ) << endl;
// testa o construtor de cópia
string *s4Ptr = new string( s1 );
cout << endl << endl << "*s4Ptr = " << *s4Ptr << endl << endl;
// testa o operador de atribuição (=) com a auto-atribuição
cout << "Atribuindo *s4Ptr para *s4Ptr" << endl;
*s4Ptr = *s4Ptr;
cout << "*s4Ptr = " << *s4Ptr << endl;
// testa o destrutor
delete s4Ptr;
// testa o uso do operador de indexacao para criar lvalue
s1[ 0 ] = 'F';
s1[ 6 ] = 'A';
cout << "\ns1 Depois de s1[0] = 'F' e s1[6] = 'A' eh: "
   << s1 << endl << endl;
//testa o subscrito fora do intervalo com a função membro de string "at"
cout << "Tentativa de atribuir 'd' para s1.at( 30 ) produz:" << endl;
s1.at( 30 ) = 'd'; // ERRO: indice fora do intervalo
return 0; }
```

Polimorfismo

Estudamos detalhadamente um conceito relacionado ao polimorfismo, mais especificamente, o conceito de sobrecarga.

Aprofundaremos, agora, nosso estudo sobre polimorfismo, tratando-o de uma forma mais ampla.

Para tanto, nos utilizaremos do mecanismo de herança, a título de revisão, o mecanismo da herança nos possibilita especializar uma classe genérica (básica).

O polimorfismo nos permite tratar objetos das classes derivadas de forma genérica. Em outras palavras, o polimorfismo nos permite "programar no geral" em vez de "programar no específico".

Polimorfismo

Inicialmente vamos tratar da possibilidade de sobreposição de uma função membro.

Conforme já vimos, sobreposição é a redefinição de um método definido em uma superclasse em uma de suas subclasses.

O exemplo a seguir demonstra este mecanismo.

```
#include <iostream>
using namespace std;
class Base {
  public:
    void func(){
      cout << "Esta eh func() de base" << endl;
    }
};
```

```
class Derivada1: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada1" << endl;
    }
};
class Derivada2: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada2" << endl;
    }
};
int main(){
  Base b;
  Derivada1 d1;
  Derivada2 d2;
  b.func();
  d1.func();
  d2.func();
  return 0;
}
```

Exercício:

O serviço de correio expresso como o SEDEX®, oferece várias opções de entrega, cada qual com custos específicos. Crie uma hierarquia de herança para representar vários tipos de pacote. Utilize Pacote como a classe base da hierarquia, então inclua as classes PacoteDoisDias e PacoteNoite que derivam de Pacote. A classe Pacote deve incluir membros de dados que representam nome, endereço, cidade e CEP tanto do remetente quanto do destinatário do pacote, além dos membros de dados que armazenam peso (em quilos) e custo por quilo para entrega do pacote. O construtor pacote deve inicializar estes membros de dados. Assegure que o peso e o custo por quilo contenham valores positivos. Pacote deve fornecer uma função-membro pública calculaCusto que retorna um double indicando o custo associado com a entrega do pacote. (continua)

A função calculaCusto de Pacote deve determinar o custo multiplicando o peso pelo custo (em quilos). A classe derivada PacoteDoisDias deve herdar a funcionalidade da classe básica Pacote, mas também incluir um membro de dados que representa uma taxa fixa que a empresa de entrega cobra pelo serviço de entrega em dois dias. O construtor de PacoteDoisDias deve receber um valor para inicializar este membro de dados. PacoteDoisDias deve redefinir a função-membro calculaCusto para que ela calcule o custo de entrega adicionando a taxa fixa ao custo baseado em peso calculado pela função calculaCusto da classe básica Pacote. A classe PacoteNoite deve herdar diretamente da classe Pacote e deve conter um membro de dados adicional para representar uma taxa adicional por quilo cobrada pelo serviço de entrega noturno. PacoteNoite deve redefinir a função-membro culculaCusto para que ela acrescente a taxa adicional por quilo ao custo padrão por quilo antes de calcular o custo de entrega. Escreva um programa de teste que cria objetos de todos os tipos definidos e testa a função calculaCusto.

Exercício:

O serviço de correio expresso como o SEDEX®, oferece várias opções de entrega, cada qual com custos específicos. Crie uma hierarquia de herança para representar vários tipos de pacote. Utilize Pacote como a classe base da hierarquia, então inclua as classes PacoteDoisDias e PacoteNoite que derivam de Pacote. A classe Pacote deve incluir membros de dados que representam nome, endereço, cidade e CEP tanto do remetente quanto do destinatário do pacote, além dos membros de dados que armazenam peso (em quilos) e custo por quilo para entrega do pacote. O construtor pacote deve inicializar estes membros de dados. Assegure que o peso e o custo por quilo contenham valores positivos. Pacote deve fornecer uma função-membro pública calculaCusto que retorna um double indicando o custo associado com a entrega do pacote. A função calculaCusto de Pacote deve determinar o custo multiplicando o peso pelo custo (em quilos). A classe derivada PacoteDoisDias deve herdar a funcionalidade da classe básica Pacote, mas também incluir um membro de dados que representa uma taxa fixa que a empresa de entrega cobra pelo serviço de entrega em dois dias. O construtor de PacoteDoisDias deve receber um valor para inicializar este membro de dados. PacoteDoisDias deve redefinir a função-membro calculaCusto para que ela calcule o custo de entrega adicionando a taxa fixa ao custo baseado em peso calculado pela função calculaCusto da classe básica Pacote. A classe PacoteNoite deve herdar diretamente da classe Pacote e deve conter um membro de dados adicional para representar uma taxa adicional por quilo cobrada pelo serviço de entrega noturno. PacoteNoite deve redefinir a função-membro culculaCusto para que ela acrescente a taxa adicional por quilo ao custo padrão por quilo antes de calcular o custo de entrega. Escreva um programa de teste que cria objetos de todos os tipos definidos e testa a função calculaCusto.