Curso Linguagem C++

Capítulo 7
Iniciando Orientação a Objetos

Implementando as funções membras

- Agora, para escrever a implementação das funções membras da **struct**, precisaremos indicar que elas **não são** funções globais ou membras de uma outra **struct** qualquer, e sim, exatamente, funções membras **desta struct**. Isso é feito através de um operador, o "operador de resolução de escopo", que é representado pelo símbolo "**::**" (dois pontos duplos).

```
struct Data
{
    bool m_ok;
    Data();
}
Data::Data()
{
    m_ok = false;
}
```

As palavras reservadas *struct* e *class*

- **C++** tem duas palavras reservadas que fazem a mesma coisa, com uma pequena diferença: **struct** e **class**.
- Podemos usar qualquer uma das duas para declarar uma estrutura de dados e funções. A única diferença entre elas é que, na **struct** o acesso "***default***" é **public**, enquanto na **class** o acesso "***default***" é **private**.

As palavras reservadas *struct* e *class*

Assim, teríamos:

```
struct Qualquer
{
    int x ; /*como não explicitamos o acesso, será usado o "default";
              logo, será public, já que usamos struct */
};


class Qualquer
{
    int x ; /*como não explicitamos o acesso, será usado o "default";
              logo, será private, já que usamos class*/
};
```

Especificando o ponteiro "*this*" como *const*

- Já vimos que se uma função recebe um ponteiro, mas precisa apenas **ler** a variável apontada, deixávamos isso claro especificando o ponteiro como **const**:
  - **void imprimir(const Data * pdt);**

Especificando o ponteiro "*this*" como *const*

- Mas, neste último caso, como fazer isso em **C++**, já que o endereço da variável é passado implicitamente através do parâmetro oculto **this**?

Especificando o ponteiro "*this*" como *const*

- Mas, neste último caso, como fazer isso em **C++**, já que o endereço da variável é passado implicitamente através do parâmetro oculto **this**?

- PARA ESSA SITUAÇÃO, A LINGUAGEM RESERVOU UMA SINTAXE ESPECIAL, JÁ QUE O PARÂMETRO É OCULTO:
- **<TIPO> NOME_FUNCAO (<PARÂMETROS EXPLÍCITOS>) CONST ;**

Especificando o ponteiro "*this*" como *const*

- O especificador **const** é colocado ao final do cabeçalho da função. E isso significa então que, nessa função, o **ponteiro this** é recebido como **const.**
- Então, vamos aplicar o especificador **const** nas funções em que ele se aplique na **class Data.** Analisemos as funções:
  - construtora: ira alterar dados, então o **this** não pode ser **const**;
  - "alterar" : como o próprio nome esta dizendo…
  - "imprimir": idem, pois apenas lê os campos para envia-los a um dispositivo de saída.

Especificando o ponteiro *"this"* como *const nas novas funçoes*

- Para satisfazer os requisitos, a nossa classe data precisará das seguintes novas funções:
  - "validar": Vai verificar se a data é uma data válida e vai alterar o m_ok;
- "ultimoDiaMes": não deve alterar dados, apenas ira apurar o último dia de cada mês; então o **this** deve ser especificado como **const.**
- "anoBissexto": idem, pois devera apenas avaliar se o ano é bissexto;

Declarando e implementando a estrutura

- Agora iremos migrar a estrutura "Data" feita em **C** para **C++**. Além disso, escreveremos novas funções.
- ***Para isso, crie um novo diretório e, dentro dele, crie três arquivos:***
  - **data.h**, para a declaração da estrutura e outros recursos de *interface* de programação que se façam necessários;
  - **data.cpp**, para a implementação das funções-membro;
  - **testadata.cpp**, para testar a estrutura.

Implementando a declaração da estrutura

- No arquivo data.h vamos implementar a interface da nossa classe.

Data.h

```
#include <iostream>
//Também renomeamos a função DATA_ALTERA para altera
class Data
{
    short m_dia;
    short m_mes;
    short m_ano;
    bool  m_ok;
public:
    enum {ANO_MIN_DEF=1900, ANO_MAX_DEF=2100};//Novo
    enum {FEVEREIRO = 2, JULHO = 7 };//Novo - meses de corte, será explicado mais adiante
    Data() ;
    void  alterar(short dia, short mes, short ano) ;
    void  validar();//Novo
    short ultimoDiaMes() const;//Novo
    bool  anoBissexto() const;//Novo
    void  imprimir();
};
```

Data.cpp

```
using namespace std;
Data::Data()
{
    m_ok = false;
}
void Data::alterar(short dia, short mes, short ano)
{
    m_dia = dia;
    m_mes = mes;
    m_ano= ano;
    validar();
}
```

Data.cpp

```
void Data::validar() {
    m_ok = (m_ano >= ANO_MIN_DEF) && (m_ano <= ANO_MAX_DEF) &&
                (m_mes >= 1)  && (m_mes  <= 12) &&
                (m_dia  >=  1) &&  (m_dia <= ultimoDiaMes());
}
void Data::imprimir() {
    if(m_ok) {
        std::cout.fill('0'); // caracter de preenchimento à esquerda:
        std::cout.width(2) ; std::cout << m_dia << '/';
        std::cout.width(2) ; std::cout << m_mes << '/';
        std::cout.width(4) ; std::cout << m_ano << '\n';
    }
    else {
        std::cout << "***ERRO***\n" ;
    }
}
```

Exercício

- 1) Escreva a função "**anoBissexto**", sabendo que:
  - Um ano é bissexto quando é divisível **por 400**
  - Ou então quando é divisível **por 4, mas não por 100**.
- **Exemplos:**
  - 2000 : bissexto, pois é divisível por 400;
  - 1996 : bissexto, pois embora não seja divisível por 400, divide por 4 mas não por 100
  - 1800 : **não-**bissexto, pois não é divisível por 400,e, embora seja divisível por 4, **também é** divisível por 100.

Tempo 45 Minutos

Solução para o exercício

- Serão apresentadas aqui várias formas de se resolver o problema enunciado.
- Contudo, há uma diferença de *performance* entre elas. Primeiramente é apresentada uma versão menos eficiente (execução mais lenta) e, em seguida, uma mais eficiente (execução mais rápida).
- Assim, a última será sempre a mais eficiente.

- **a) AnoBissexto:**

```
bool Data::anoBissexto( ) const
{
    // se for divisível por 400 é bissexto:
    if ( m_ano % 400 == 0 )
        return true;
    // se for divisível por 4, mas não por 100, também é bissexto:
    if ( m_ano % 4 == 0 && m_ano % 100 != 0 )
        return true;
    return false; // se chegou até aqui, é porque não é bissexto.
}
```

- ***B)*** *segunda versão (usa o operador lógico* **OR** [símbolo: ||]):

```
bool Data::anoBissexto( ) const
{
    return m_ano % 400 == 0 ||
        (m_ano % 4 == 0 && m_ano % 100 != 0);
}
```

- *C**)** terceira versão (usa o operador lógico* **NOT** [símbolo: **!**]):

```
bool Data::anoBissexto( ) const
{
    return !(m_ano % 400) || ( !(m_ano % 4) && (m_ano % 100) ) ;
}
```

- ***d)*** *quarta versão (procura executar as operações da* ***mais frequente para a menos frequente -*** afinal, a comparação com 400 colocada em primeiro lugar, esta privilegiando uma situação **menos frequente):**

```
bool Data::anoBissexto( ) const
{
    return !(m_ano % 4) && ( (m_ano % 100) || !(m_ano % 400));
}
```

- ***e)*** *quinta versão (usa o operador de bit's* ***AND****, para apurar* **o resto da divisão por 4,** pois, nesse caso, **através de um and, bit a bit, entre o ano e a constante 3,** é possível saber se é divisível por 4 sem usar o operador de **módulo, mais lento**):

  /* Observe que:

  1 [0001] & 3 [0011] -> 0001 -> 1

  2 [0010] & 3 [0011] -> 0010 -> 2

  3 [0011] & 3 [0011] -> 0011 -> 3

  4 [0100] & 3 [0011] -> 0000 -> **0**

  5 [0101] & 3 [0011] -> 0001 -> 1

  6 [0110] & 3 [0011] -> 0010 -> 2

  7 [0111] & 3 [0011] -> 0011 -> 3

  8 [1000] & 3 [0011] -> 0000 -> **0**

  9 [1001] & 3 [0011] -> 0001 -> 1

  10 [1010] & 3 [0011] -> 0010 -> 2

  11 [1011] & 3 [0011] -> 0011 -> 3

  12 [1100] & 3 [0011] -> 0000 -> **0**

  ... etc ...

  Enfim: apenas múltiplos de 4 em uma operação **and**, bit a bit,

  contra a constante 3,**apresentam resultado zero.**

  Então:*/*

```
bool Data::anoBissexto( ) const
{
    return !(m_ano & 3) && ( (m_ano % 100) || !(m_ano % 400) );
}
```

Exercício

- 2) Escreva a função "**ultimoDiaMes**", sabendo que:
  - - fevereiro …… : 28 ou 29 dias, se o ano for bissexto;
  - - janeiro a julho … : meses pares tem 30 dias, e impares 31 dias;
  - - agosto a dezembro: meses pares tem 31 dias, e impares 30 dias;
- Exemplos:
  - mes 6(junho)…… : anterior a agosto e par -> 30 dias;
  - mes 7(julho) …… : anterior a agosto e impar -> 31 dias;
  - mes 8(agosto)…….. : posterior a julho e par -> 31 dias;
  - mes 9(setembro)…. : posterior a julho e impar -> 30 dias;

Tempo 45 Minutos

Último dia do mês

A) primeira versão:

```
short Data::ultimoDiaMes( ) const
{
    if ( m_mes == FEVEREIRO ) // 'FEVEREIRO' é uma constante da class
    {
        if ( anoBissexto() ) // se o ano for bissexto…
            return 29;
        return 28; // se chegou até aqui, é porque o ano não é bissexto.
    }
    // 'JULHO' é uma constante da class:
    if ( m_mes <= JULHO ) // janeiro a julho, exceto fevereiro
    {
        if ( m_mes & 1 ) // se o mês é ímpar
            return 31 ;
        return 30 ; // se chegou até aqui, é porque o mês é par.
    }
    // se chegou até aqui, é porque o mês está entre agosto e dezembro, inclusive estes;
    então:
    if ( m_mes & 1 ) // se o mês é ímpar
        return 30 ;
    return 31 ; // se chegou até aqui, é porque o mês é par.
}
```

- ***B)*** *segunda versão:*

  (nesta versão usaremos ***o operador condicional ternário;***

  esse operador funciona como um "if inline":

  (condição) **?** resultado_se_condicao_verdadeira**:** resultado_se_condicao_falsa ;

  portanto o símbolo **"?"** e uma **pergunta** que exige resposta **verdadeira e o símbolo ":"**(dois pontos) **significa "do contrário" (else).**

```
short Data::ultimoDiaMes( ) const
{
    if ( m_mes == FEVEREIRO )
        return ( anoBissexto() ) ? 29 : 28; // usa o operador condicional
    if (m_mes <= JULHO ) // janeiro a julho exceto fevereiro
        return ( m_mes & 1 ) ? 31 : 30;
    // aqui, só pode ser agosto a dezembro:
    return (m_mes & 1 ) ? 30 : 31;
}
```

- ***c)*** *terceira versão (economiza testes de condição):*

```
char Data::ultimoDiaMes( ) const
{
    if ( m_mes == FEVEREIRO )
        return 28 + anoBissexto(); /* soma 1, se for bissexto,
                                      e zero se não for */
    if (m_mes <= JULHO ) // janeiro a julho exceto fevereiro
        return 30+( m_mes & 1 ); // soma 1, se for ímpar e zero
// se não for agosto a dezembro:
    return 31-(m_mes & 1);
                // subtrai 1, se for ímpar e zero se não for.
}
```

D) quarta versão (usa operador condicional no lugar do último 'if'):

```
short Data::ultimoDiaMes( ) const
{
    if ( m_mes == FEVEREIRO )
        return 28 + anoBissexto();
    return ( m_mes <= JULHO ) ? 30 + ( m_mes & 1 ) :
                                31 - ( m_mes & 1 );
}
```

E) *quinta versão (usa o operador* ***xor*** *bit a bit).*

XOR é verdade se os valores de suas entradas forem diferentes e será falsa se os valores das entradas forem iguais.

Entrada		Saída
A	B	X = A + B
0	0	0
0	1	1
1	0	1
1	1	0

```
short Data::ultimoDiaMes( ) const
{
    return (m_mes == FEVEREIRO) ? 28 + anoBissexto( ) :
                                  30 +  ( (m_mes & 1) ^ (m_mes > JULHO));
  // usando o xor bit a bit
}
```

Simulações:

Imaginem que o m_mes seja Janeiro (1).

Se o m_mes é igual a 1, logo é diferente de 2, sendo assim o que seria avaliado seria:

30 + ((m_mes & 1) ^ (m_mes > JULHO)).

30 + ((1 & 1) ^ (1 > 7))

30 + ((1) ^ (0))

30 + 1

31

```
short Data::ultimoDiaMes( ) const
{
    return (m_mes == FEVEREIRO) ? 28 + anoBissexto( ) :
                                  30 +  ( (m_mes & 1) ^ (m_mes > JULHO));
     // usando o xor bit a bit
}
```

Simulações:

Imaginem que o m_mes seja Junho (6).

Se o m_mes é igual a 6, logo é diferente de 2, sendo assim o que seria avaliado seria:

30 + ((m_mes & 1) ^ (m_mes > JULHO)).

30 + ((6 & 1) ^ (6 > 7))

30 + ((0) ^ (0))

30 + 0

30

```
short Data::ultimoDiaMes( ) const
{
  return (m_mes == FEVEREIRO) ? 28 + anoBissexto( ) :
                                30 + ( (m_mes & 1) ^ (m_mes > JULHO));
  // usando o xor bit a bit
}
```

Simulações:

Imaginem que o m_mes seja Outubro (10).

Se o m_mes é igual a 10, logo é diferente de 2, sendo assim o que seria avaliado seria:

30 + ((m_mes & 1) ^ (m_mes > JULHO)).

30 + ((10 & 1) ^ (10 > 7))

30 + ((0) ^ (1))

30 + 1

31

```
short Data::ultimoDiaMes( ) const
{
   return (m_mes == FEVEREIRO) ? 28 + anoBissexto( ) :
                                 30 + ( (m_mes & 1) ^ (m_mes > JULHO));
   // usando o xor bit a bit
}
```

Testando a nossa classe data

```
#include <iostream>
#include <data.h>
using namespace std;
int main()
{
   Data d1;
   d1.alterar(31,1,2001); d1.imprimir(); // resultado: 31/01/2001
   d1.alterar(29,2,2001); d1.imprimir(); // resultado: ERRO | Não bissexto
   d1.alterar(29,2,1997); d1.imprimir(); // resultado: ERRO | Não bissexto
   d1.alterar(29,2,1800); d1.imprimir(); // resultado: ERRO | Não bissexto
   d1.alterar(29,2,1996); d1.imprimir(); // resultado: 29/02/1996
   d1.alterar(29,2,2000); d1.imprimir(); // resultado: 29/02/2000
   d1.alterar(31,6,2001) ; d1.imprimir(); // resultado: ERRO | Junho tem apenas 30 dias
   d1.alterar(31,7,2001) ; d1.imprimir(); // resultado: 31/07/2001
   d1.alterar(31,8,2001) ; d1.imprimir(); // resultado: 31/08/2001
   d1.alterar(31,9,2001) ; d1.imprimir(); // resultado: ERRO | Setembro tem apenas 30 dias
   d1.alterar(31,12,2001); d1.imprimir(); // resultado: 31/12/2001
   return 0;
}
```