

POO – continuação

- Objetos e membros *const*
- Gerenciamento dinâmico de memória (new e delete)
- Membros *static*
- O ponteiro *this*

Objetos *const* e métodos *const*

- ▶ Princípio do menor privilégio
 - Modificações somente onde são realmente necessárias
- ▶ Palavra-chave *const*
 - Especificar objetos que não são modificáveis
 - Erro de compilador caso tente modificar objetos *const*
 - Exemplo


```
const Hora meio_dia( 12, 0, 0 );
```

 - Declara objeto *const* *meio_dia* da classe *hora*
 - Inicializa com 12

2

Const - continuação

- ▶ Métodos *const*
 - Se objeto é constante, então somente pode chamar métodos constantes
 - Deve-se especificar *const* tanto no protótipo como na definição
 - Protótipo


```
void imprime() const;
```

 - Após a lista de parâmetros
 - Definição


```
void horario::imprime() const
{
    cout << hora << ":" << minute << endl;
}
```

 - Antes da {

3

Const - continuação

- ▶ Objetos *const*
 - Métodos não *const* não podem ser acessadas
 - Mesmo que sejam métodos *get* que não modificam dados
 - Não basta não modificar: devem ser declaradas *const*
- ▶ Declarar como *const* todos os métodos que não alteram os dados do objeto
- ▶ Compilador não permite que métodos *const* modifiquem atributos
 - Métodos *const* não podem chamar métodos não *const*

Const - Continuação

- ▶ Construtores e destrutores
 - Não podem ser *const*
 - Devem poder modificar objetos
 - Construtor
 - Inicializa objetos
 - Destrutor
 - Limpeza
- ▶ *Const*: depois do construtor e antes do destrutor
 - ▶ Importância de se inicializar os dados no construtor

5

```

1 //
2 // Class Hora
3 //
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Hora {
8
9 public:
10     Hora( int = 0, int = 0, int = 0 ); // construtor padrão
11
12     // métodos tipo SET
13     void acerta_horario( int, int, int );
14     void acerta_hora ( int );
15     void acerta_minuto ( int );
16     void acerta_segundo ( int );
17

```

```

18 // métodos tipo GET (normalmente devem ser const)
19 int recupera_hora() const;
20 int recupera_minuto() const;
21 int recupera_segundo() const;
22
23 // métodos auxiliares de impressão (normalmente const)
24 void hora_Universal() const;
25 void hora_padrao();
26
27 private:
28 int hora;
29 int minuto;
30 int segundo;
31
32 }; // fim da classe hora
33
34 #endif

```

```

// DEFINIÇÃO
#include <iostream>

using std::cout;
#include <iomanip>
using std::setfill;
using std::setw;

// incluir definição da classe
#include "hora.h"

// construtor inicializa dados privados;
// chama o método acerta_hora;
// valor default é 0
Hora::Hora( int h, int m, int s )
{
    acerta_horario( h, m, s );
} // fim do construtor

```

```

// acerta hora, minuto e segundo
void Hora::acerta_horario( int hour, int minute, int second )
{
    acerta_hora( hour );
    acerta_minuto( minute );
    acerta_segundo( second );
} // fim do método acerta_horario

// acerta_hora
void Hora::acerta_hora( int h )
{
    hora = ( h >= 0 && h < 24 ) ? h : 0;
}

// acerta_minuto
void Hora::acerta_minuto( int m )
{
    minuto = ( m >= 0 && m < 60 ) ? m : 0;
}

```

```

// acerta_segundo
void Hora::acerta_segundo( int s )
{
    segundo = ( s >= 0 && s < 60 ) ? s : 0;
}

// recupera_hora
int Hora::recupera_hora() const
{
    return hora;
}

// recupera_minuto
int Hora::recupera_minuto() const
{
    return minuto;
}

```

métodos const não modificam objetos.

```

// recupera_segundo
int Hora::recupera_segundo() const
{
    return segundo;
}

// Horário no formato universal
void Hora::hora_Universal() const
{
    cout << setfill( '0' ) << setw( 2 ) << recupera_hora() << " : "
    << setw( 2 ) << recupera_minuto() << " : "
    << setw( 2 ) << recupera_segundo();
}

// Formato padrão
void Hora::hora_padrao() // sem const
{
    cout << ( ( recupera_hora() == 0 || recupera_hora() == 12 ) ? 12 : recupera_hora() % 12 )
    << " : " << setfill( '0' ) << setw( 2 ) << recupera_minuto()
    << " : " << setw( 2 ) << recupera_segundo()
    << ( recupera_hora() < 12 ? " AM" : " PM" );
}

```

Métodos const não modificam objetos.

```

// programa principal
// tentativa de acessar objeto const com
// método não onst

// incluir a classe
#include "hora.h"

```

```

int main()
{
    Hora acordar( 6, 45, 0 ); // não const
    const Hora almocar( 12, 0, 0 ); // const
}

```

almocar é objeto const.

Note que um construtor não const pode inicializar objeto const.

```

// OBJETO      METODO
acordar.acerta_hora( 18 ); // non-const non-const

almocar.acerta_hora( 12 ); // const non-const

acordar.recupera_hora(); // non-const
almocar.recupera_minuto(); // const
almocar.hora_Universal(); // const
almocar.hora_padrao(); // const non-const

return 0;
} // fim do programa principal

```

Tentando invocar membro não-const de um objeto const resulta em erro de compilador

Tentando invocar métodos não-const de objeto const resulta em erro de compilação mesmo que o método não modifique o objeto.

Membros de dados const

- ▶ Inicializador de membro
 - Todos os dados podem ser inicializados utilizando a inicialização de membros
 - Dados const devem

```

EX. Para classe Aluno
class Aluno
{
private:
    const int ra;
    string nome;
    int p1, p2, ps;
-
}

Aluno(int _ra, string _nome, int _prova1, int _prova2, int _provasub) : ra(_ra) {
...
}

```

14

Exercício

- ▶ Classe para armazenar objetos do tipo Data
 - Dia, mês e ano
 - Validar valores:
 - número de dias no mês
 - Bissexto
 - É divisível por 400 ou
 - (não é divisível por 100 E é divisível por 4)
 - Declarar 2 datas:
 - Nascimento: 20/01/1980 (constante)
 - Casamento: 10/05/2010

Gerenciamento dinâmico de memória:

new e delete

- ▶ Controla alocação e liberação da memória
- ▶ Uso dos operadores new e delete
 - O programador é responsável por 'devolver' o recurso alocado (solicitado)
 - NEW: comumente usado nos construtores
 - Delete: comumente usado nos destrutores

16

Gerenciamento dinâmico de memória (new e delete)

- ▶ new
 - Exemplo:


```

Hora *HoraPtr;
HoraPtr = new Hora;

```
 - new
 - Cria objeto do tamanho apropriado para **Hora**
 - Erro se não houver espaço em memória
 - Chama construtor padrão (default)
 - Retorna ponteiro do tipo especificado

17

Gerenciamento dinâmico de memória (new e delete)

- Inicialização


```

double *ptr = new double( 3.14159 );
Hora *HoraPtr = new Hora( 12, 0, 0 );

```
- Alocando arrays (vetores)


```

int *meuArray = new int[ 10 ];

```

18

Gerenciamento dinâmico de memória (new e delete)

- ▶ **Malloc()** não deve ser usada para alocação dinâmica de objetos
 - Não chama construtor
 - Retorna ponteiro void
- ▶ **New**
 - Chama construtor automaticamente
 - Retorna ponteiro apropriado

Gerenciamento dinâmico de memória: new e delete

- ▶ **delete**
 - Destrói dinamicamente objetos alocados (libera espaço)
 - Exemplo:


```
delete HoraPtr;
```
 - Operador **delete**
 - Chama destrutor para o objeto
 - Libera memória associada ao objeto
 - Pode ser reutilizada
 - Delete para arrays


```
delete [] meuArray;
```

20

Exercício

- ▶ Desenvolver uma classe que implemente um "Vetor dinâmico" de inteiros
 - Tamanho do vetor definido pelo usuário (tempo de execução)
 - Métodos set e métodos get
 - Construtor e destrutor

Membros static

- ▶ **Dados static**
 - Dados da classe, não do objeto
 - Eficiente quando cópia única do dado é suficiente / necessária
 - Parece com variáveis globais mas tem escopo de classe
 - Compartilhado pelos objetos da classe
 - Pode ser **public**, **private** ou **protected**
 - Precisam ser inicializados fora da classe

22

static – continuação


- ▶ Existem mesmo quando nenhum objeto da classe foi instanciado
 - ▶ Acessando dados **static**
 - **public static**
 - Podem também ser acessadas pelo operador de escopo (::)

```
Empregado::contador
```
 - **private static**
 - Quando nenhum membro da classe existir
 - Só pode ser acessado via métodos **public static**

```
Empregado::getContador()
```
- Também é acessível através de qualquer objeto da classe

23

Métodos static

- ▶ Independentes de objetos declarados
 - Existem mesmo quando nenhum objeto da classe foi instanciado
- ▶ Acessam somente dados static 
- ▶ Chamadas pelo operador de escopo ::
 - Contador::getCont()
- Exemplo: contador

```
// EMPREGADO
#ifndef EMPREGADO_H
#define EMPREGADO_H

class Empregado {
public:
    Empregado( const char *, const char * ); // construtor
    ~Empregado(); // destrutor
    const char *get_prim_nome() const;
    const char *get_ult_nome() const;
    // método static
    static int get_contador();

private:
    char *prim_nome;
    char *ult_nome;
    // dado static
    static int contador; // numero de objetos instanciados
};
#endif
```

Métodos static só
podem acessar dados
static.

Dado static.

```
// MÉTODOS DE EMPREGADO
#include <iostream>
using std::cout;
using std::endl;
#include <new> // C++ P/ USAR NEW
#include <cstring> // strcpy E strlen

#include "empregado.h"

// define e inicializa membros static
int Empregado::contador = 0;

// define método que retorna nro de objetos
// instanciados
int Empregado::get_contador()
{
    return contador;
}
```

```
// construtor que faz alocação dinamica
Empregado::Empregado( const char *first, const char *last )
{
    prim_nome = new char[ strlen( first ) + 1 ];
    strcpy( prim_nome, first );

    ult_nome = new char[ strlen( last ) + 1 ];
    strcpy( ult_nome, last );
    ++contador; // incrementa contador static
    cout << "Construtor de empregado" << prim_nome
        << " " << ult_nome << " chamado." << endl;
}

// destrutor libera memória
Empregado::~Empregado()
{
    cout << "=-Empregado() chamado = " << prim_nome
        << " " << ult_nome << endl;
}
```

```
delete [] prim_nome; // libera memória
delete [] ult_nome; // libera memória
```

```
--contador;
```

```
} // fim do destrutor
```

Atualizar total de objetos
ativos.

```
// retorna primeiro nome do empregado
const char *Empregado::get_prim_nome() const
{
    // const antes do tipo de retorno previne que o usuario
    // modifique dados privados
    return prim_nome;
}

// retorna ultimo nome do empregado
const char *Empregado::get_ult_nome() const
{
    return ult_nome;
}
```

```
// PROGRAMA PRINCIPAL
#include <iostream>

using namespace std;

#include "empregado.h" // DEFINIÇÕES
```

```
int main()
{
    cout << "Numero atual de empregados é "<< Empregado::get_contador() << endl;

    Empregado *e1Ptr = new Empregado( "Susana", "Alves" );
    Empregado *e2Ptr = new Empregado( "Roberto", "Dinamite" );
    cout << "Numero atual de empregados é "<< Empregado::get_contador() << endl;
```

Métodos static
podem ser invocadas por
qualquer objeto da classe.

```
cout << "\n\nEmpregado 1: " << e1Ptr->get_prim_nome()
    << " " << e1Ptr->get_ult_nome()
    << "\n\nEmpregado 2: " << e2Ptr->get_prim_nome()
    << " " << e2Ptr->get_ult_nome() << "\n\n";
```

```
delete e1Ptr; // libera memória
e1Ptr = 0; // aponta para NULL
delete e2Ptr; // libera memória
e2Ptr = 0; // aponta para NULL
```

```
cout << "Numero atual de empregados é "<< Empregado::get_contador() << endl;
```

```
return 0;
```

```
} // fim
```

Alunos PC

- ▶ Exercício
 - Contador de alunos
 - Usar contador para definir RA
 - Sempre crescente

Usando o ponteiro `this`

- ▶ `this`
 - Ponteiro para o próprio objeto
 - Permite acessar próprio endereço
 - Não é parte do objeto
 - Tipo do ponteiro `this` depende de:
 - Tipo do objeto
 - Se o método é ou não `const`

32

```
// EXEMPLO
#include <iostream>

using std::cout;
using std::endl;

class Teste {
public:
    Teste( int = 0 );
    void imprime() const;
private:
    int x;
};

// construtor
Teste::Teste( int valor ) : x( valor ) // inicializa x com valor
{
    // ...
} // fim do construtor
```

```
// imprime x usando ponteiro this ;
// parenteses ao redor de *this é necessário
void Teste::imprime() const
{
    // uso implicito
    cout << "    x = " << x;
    // uso explicito
    cout << "\n this->x = " << this->x;
    // alternativa
    cout << "\n(*this).x = " << ( *this ).x << endl;
}

int main()
{
    Teste Objeto( 12 );
    Objeto.imprime();
    return 0;
}
```

```
x = 12
this->x = 12
(*this).x = 12
```

Exemplo

- ▶ Classe AlunosPC
- ▶ Obs.: `this` não pode ser usado em métodos `static`. Por que?