INF 213 – Estrutura de Dados
Prof. Marcus V. A. Andrade

Classes
Estudo aprofundado 1

- **Empacotador de pré-processadores**
  - **Evita que o código seja incluído mais de uma vez.**
    - **`#ifndef` – 'se não definido'**
      - **Pula esse código se já tiver sido incluído.**
    - **`#define`**
      - **Define um nome para que esse código não seja incluído novamente.**
    - **`#endif`**
  - **Se o cabeçalho tiver sido incluído previamente**
    - **O nome já será definido e o arquivo de cabeçalho não será novamente incluído.**
  - **Evita erros de múltiplas definições.**
  - **Exemplo**
    - ```
      #ifndef TIME_H
      #define TIME_H
      … // code
      #endif
      ```
      ```

Time.h

```
1  // Figura 9.1: Time.h
2  // Declaração da classe Time.
3  // Funções-membro são definidas em Time.cpp
4
5  // impede múltiplas inclusões de arquivo de cabeçalho
6  #ifndef TIME_H
7  #define TIME_H
8
9  // definição da classe Time
10 class Time
11 {
12 public:
13    Time(); // construtor
14    void setTime( int, int, int ); // configura hora, minuto e segundo
15    void printUniversal(); // imprime a hora no formato de data/hora universal
16    void printStandard(); // imprime a hora no formato-padrão de data/hora
17 private:
18    int hour; // 0 - 23 (formato de relógio de 24 horas)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // fim da classe Time
22
23 #endif
```

A diretiva do pré-processador **#ifndef** determina se foi definido um nome.

A diretiva de pré-processador **#define** define um nome (por exemplo, **TIME_H**).

A diretiva de pré-processador **#endif** marca o fim do código que não deve ser incluído várias vezes.

Boa prática de programação

Utilize o nome do arquivo do cabeçalho em maíusculo, substituindo o ponto por um sublinhado nas diretivas de pré-processador `#ifndef` e `#define` de um arquivo de cabeçalho.

Escopo de classe e acesso a membros de classe

- **Operador de seleção de membro ponto (.)**
  - **Acessa os membros do objeto.**
  - **Usado com o nome de um objeto ou com uma referência a um objeto.**
- **Operador de seleção de membro seta (->)**
  - **Acessa os membros do objeto.**
  - **Usado com um ponteiro para um objeto.**

fig09_04.cpp
(1 de 2)

```
1  // Figura 9.4: fig09_04.cpp
2  // Demonstrando os operadores de acesso ao membro de classe com . e ->
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // definição da classe Count
8  class Count
9  {
10 public: // dados public são perigosos
11    // configura o valor do membro de dados private x
12    void setX( int value )
13    {
14       x = value;
15    } // fim da função setX
16
17    // imprime o valor do membro de dados private x
18    void print()
19    {
20       cout << x << endl;
21    } // fim da função print
22
23 private:
24    int x;
25 }; // fim da classe Count
```

fig09_04.cpp
(2 de 2)

Usando o operador de seleção de membro ponto com um objeto.

Usando o operador de seleção de membro ponto com uma referência.

Usando o operador de seleção de membro seta com um ponteiro.

```
26
27 int main()
28 {
29    Count counter; // cria objeto counter
30    Count *counterPtr = &counter; // cria ponteiro para counter
31    Count &counterRef = counter; // criar referência para counter
32
33    cout << "Set x to 1 and print using the object's name: ";
34    counter.setX( 1 ); // configura membro de dados x como 1
35    counter.print(); // chama função-membro print
36
37    cout << "Set x to 2 and print using a reference to an object: ";
38    counterRef.setX( 2 ); // configura membro de dados x como 2
39    counterRef.print(); // chama função-membro print
40
41    cout << "Set x to 3 and print using a pointer to an object: ";
42    counterPtr->setX( 3 ); // configura membro de dados x como 3
43    counterPtr->print(); // chama função-membro print
44    return 0;
45 } // fim de main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Separando a interface da implementação

- **Separando uma definição de classe e as definições de função-membro da classe**
  - **Facilita a modificação de programas.**
    - **Mudanças na implementação da classe não afetam o cliente, visto que a interface da classe permanece inalterada.**
  - **As coisas não são tão promissoras assim.**
    - **Os arquivos de cabeçalho contêm algumas partes da implementação e dicas sobre outras.**
      - **As funções inline precisam ser definidas no arquivo de cabeçalho.**
      - **O membros `private` são listados na definição de classe no arquivo de cabeçalho.**

Observação de engenharia de software

As informações importantes para a interface para uma classe devem ser incluídas no arquivo de cabeçalho. As informações que só serão utilizadas internamente na classe e não serão necessárias aos clientes da classe devem ser incluídas no arquivo-fonte não publicado. Esse é também outro exemplo do princípio do menor privilégio.

Estudo de caso da classe `Time`: construtores com argumentos-padrão

- **Os construtores podem especificar argumentos-padrão**
  - **Podem inicializar membros de dados em um estado consistente.**
    - **Mesmo se não for fornecido nenhum valor em uma chamada de construtor.**
  - **O construtor que assume um padrão para todos os seus argumentos é também chamado de construtor-padrão.**
    - **Pode ser invocado sem nenhum argumento.**
    - **É possível existir no máximo um construtor-padrão por classe.**

Time.h
(1 de 2)

```
1  // Figura 9.8: Time.h
2  // Declaração da classe Time.
3  // Funções-membro definidas em Time.cpp.
4
5  // impede múltiplas inclusões de arquivo de cabeçalho
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Definição de tipo de dados abstrato Time
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // construtor-padrão
14
15    // funções set
16    void setTime( int, int, int ); // configura hour, minute, second
17    void setHour( int ); // configura hour (depois da validação)
18    void setMinute( int ); // configura minute (depois da validação)
19    void setSecond( int ); // configura second (depois da validação)
```

Protótipo de um construtor com argumentos-padrão.

Time.h
(2 de 2)

```
20
21    // funções get
22    int getHour(); // retorna hour
23    int getMinute(); // retorna minute
24    int getSecond(); // retorna second
25
26    void printUniversal(); // gera saída da hora no formato universal de data/hora
27    void printStandard(); // gera saída da hora no formato-padrão de data/hora
28 private:
29    int hour; // 0 - 23 (formato de relógio de 24 horas)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // fim da classe Time
33
34 #endif
```

Time.cpp
(1 de 3)

```
1  // Figura 9.9: Time.cpp
2  // Definições de função-membro para a classe Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
11
12 // Construtor de Time inicializa cada membro de dados como zero;
13 // assegura que os objetos Time iniciem em um estado consistente
14 Time::Time( int hr, int min, int sec )
15 {
16    setTime( hr, min, sec ); // valida e configura time
17 } // fim do construtor de Time
18
19 // configura novo valor de Time utilizando a hora universal; assegura que
20 // os dados permaneçam consistentes configurando valores inválidos como zero
21 void Time::setTime( int h, int m, int s )
22 {
23    setHour( h ); // configura campo private hour
24    setMinute( m ); // configura campo private minute
25    setSecond( s ); // configura campo private second
26 } // fim da função setTime
27
28 // configura valor de hour
29 void Time::setHour( int h )
```

Os parâmetros poderiam receber os valores-padrão.

```
27
28 // configura valor de hour
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
32 } // fim da função setHour
33
34 // configura valor de minute
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
38 } // fim da função setMinute
39
40 // configura valor de second
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
44 } // fim da função setSecond
45
46 // retorna valor de hour
47 int Time::getHour()
48 {
49    return hour;
50 } // fim da função getHour
51
52 // retorna valor de minute
53 int Time::getMinute()
54 {
55    return minute;
56 } // fim da função getMinute
```

```
57
58 // retorna valor de second
59 int Time::getSecond()
60 {
61    return second;
62 } // fim da função getSecond
63
64 // imprime a hora no formato universal de data/hora (HH:MM:SS)
65 void Time::printUniversal()
66 {
67    cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68       << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // fim da função printUniversal
70
71 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
72 void Time::printStandard()
73 {
74    cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76       << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // fim da função printStandard
```

Observação de engenharia de software

Se a função-membro de uma classe já fornecer toda ou parte da funcionalidade requerida por um construtor (ou por outra função-membro) da classe, chame essa função-membro por meio do construtor (ou de outra função-membro). Isso simplifica a manutenção do código e reduz a probabilidade de erro se a implementação do código for modificada. Via de regra, evite a repetição de código.

Observação de engenharia de software

Qualquer alteração nos valores de argumento-padrão de uma função exige que o código-cliente seja recompilado (para assegurar que o programa permaneça funcionando corretamente).

fig09_10.cpp
(1 de 3)

```
1  // Figura 9.10: fig09_10.cpp
2  // Demonstrando um construtor-padrão para a classe Time.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // inclui a definição da classe Time a partir de Time.h
8
9  int main()
10 {
11    Time t1; // todos os argumentos convertidos para sua configuração-padrão
12    Time t2( 2 ); // hour especificada; minute e second convertidos para o padrão
13    Time t3( 21, 34 ); // hour e minute especificados; second convertido para o padrão
14    Time t4( 12, 25, 42 ); // hour, minute e second especificados
15    Time t5( 27, 74, 99 ); // valores inválidos especificados
16
17    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
18    t1.printUniversal(); // 00:00:00
19    cout << "\n  ";
20    t1.printStandard(); // 12:00:00 AM
21
22    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
23    t2.printUniversal(); // 02:00:00
24    cout << "\n  ";
25    t2.printStandard(); // 2:00:00 AM
```

Inicializando objetos **`Time`** por meio dos argumentos 0, 1, 2 e 3.

```
26
27     cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
28     t3.printUniversal(); // 21:34:00
29     cout << "\n  ";
30     t3.printStandard(); // 9:34:00 PM
31
32     cout << "\n\nt4: hour, minute and second specified\n  ";
33     t4.printUniversal(); // 12:25:42
34     cout << "\n  ";
35     t4.printStandard(); // 12:25:42 PM
36
37     cout << "\n\nt5: all invalid values specified\n  ";
38     t5.printUniversal(); // 00:00:00
39     cout << "\n  ";
40     t5.printStandard(); // 12:00:00 AM
41     cout << endl;
42     return 0;
43  } // fim de main
```

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

Valores inválidos são passados ao construtor, de modo que o objeto **t5** contém todos os dados-padrão.

Erro comum de programação

Um construtor pode chamar outras funções-membro da classe, como as funções `set` e `get`, mas, como ele está inicializando o objeto, os membros de dados talvez ainda não estejam em um estado consistente. Utilizar os membros de dados antes de serem adequadamente inicializados pode provocar erros de lógica.

9.7 Destrutores

- **Destrutor**
  - **Uma função-membro especial.**
  - **O nome é o caractere til (~) seguido pelo nome da classe, por exemplo, `~Time`.**
  - **É chamado implicitamente quando um objeto é destruído.**
    - **Por exemplo, isso ocorre quando um objeto automático é destruído porque a execução do programa deixou o escopo no qual esse objeto estava instanciado.**
  - **Na verdade, não libera a memória do objeto.**
    - **Realiza uma faxina de terminação.**
    - **Em seguida, o sistema reivindica a memória do objeto.**
      - **De modo que a memória pode ser reutilizada para abrigar novos objetos.**

9.7 Destrutores (cont.)

- **Destrutor (cont.)**
  - **Não recebe nenhum parâmetro e não retorna nenhum valor.**
    - **Não especifique nenhum tipo de retorno — nem mesmo `void`.**
  - **Uma classe pode ter um único destrutor.**
    - **A sobrecarga de destrutores não é permitida.**
  - **Se o programador não fornecer um destrutor explicitamente, o compilador criará um destrutor 'vazio'.**

Erro comum de programação

É um erro de sintaxe tentar passar argumentos para um destrutor, especificar um tipo de retorno para um destrutor (mesmo `void` não pode ser especificado), retornar valores de um destrutor ou sobrecarregar um destrutor.

9.8 Quando construtores e destrutores são chamados

- **Construtores e destrutores**
  - **São chamados implicitamente pelo compilador.**
    - **A ordem dessas chamadas de função depende da ordem segundo a qual a execução entra e sai dos escopos em que os objetos estão instanciados.**
  - **Geralmente,**
    - **As chamadas de destrutor são feitas na ordem inversa às chamadas de construtor correspondentes.**
  - **Entretanto,**
    - **As classes de armazenamento de objetos podem alterar a ordem segundo a qual os destrutores são chamados.**

9.8 Quando construtores e destrutores são chamados (cont.)

- **Para os objetos definidos no escopo global**
  - **Os construtores são chamados antes que qualquer outra função (incluindo `main`) nesse arquivo inicie a execução.**
  - **Os destrutores correspondentes são chamados quando `main` termina.**
    - **Função `exit`**
      - **Força um programa a terminar imediatamente.**
        - **Não executa os destrutores de objetos automáticos.**
      - **Em geral, é usada para terminar um programa quando é detectado um erro.**
    - **Função `abort`**
      - **É semelhante à função `exit`.**
        - **Mas força o programa a terminar imediatamente sem permitir que os destrutores de qualquer objeto sejam chamados.**
      - **Normalmente, é usada para indicar uma terminação anormal do programa.**

9.8 Quando construtores e destrutores são chamados (cont.)

- **Para um objeto local automático**
  - **O construtor é chamado quando esse objeto é definido.**
  - **O destrutor correspondente é chamado quando a execução sai do escopo do objeto.**
- **Para objetos automáticos**
  - **Os construtores e destrutores são chamados toda vez que a execução entra e sai do escopo do objeto.**
  - **Os destrutores de objeto automático não serão chamados se o programa terminar com uma função `exit` ou `abort`.**

9.8 Quando construtores e destrutores são chamados (cont.)

- **Para um objeto local `static`**
  - **O construtor é chamado uma única vez**
    - **Quando a execução atinge pela primeira vez o local em que o objeto é definido.**
  - **O destrutor é chamado quando `main` termina ou o programa chama a função `exit`.**
    - **O destrutor não será chamado se o programa terminar com uma chamada para a função `abort`.**
- **Os objetos global e `static` são destruídos na ordem inversa à que foram criados.**

```
1  // Figura 9.11: CreateAndDestroy.h
2  // Definição da classe CreateAndDestroy.
3  // Funções-membro definidas em CreateAndDestroy.cpp.
4  #include <string>
5  using std::string;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // construtor
14    ~CreateAndDestroy(); // destrutor
15 private:
16    int objectID; // Número de ID do objeto
17    string message; // mensagem descrevendo o objeto
18 }; // fim da classe CreateAndDestroy
19
20 #endif
```

Protótipo para o destrutor.

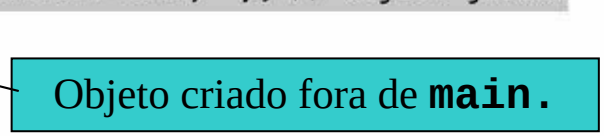
CreateAndDestroy.cpp

```
1  // Figura 9.12: CreateAndDestroy.cpp
2  // Definições de função-membro da classe CreateAndDestroy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "CreateAndDestroy.h"// inclui a definição da classe CreateAndDestroy
8
9  // construtor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12    objectID = ID; // configura o número de ID do objeto
13    message = messageString; // configura mensagem descritiva do objeto
14
15    cout << "Object " << objectID << "   constructor runs    "
16       << message << endl;
17 } // fim do construtor CreateAndDestroy
18
19 // destrutor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22    // gera saída de nova linha para certos objetos; ajuda a legibilidade
23    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
24
25    cout << "Object " << objectID << "   destructor runs    "
26       << message << endl;
27 } // fim do destrutor ~CreateAndDestroy
```

Definindo o destrutor da classe.

fig09_13.cpp
(1 de 3)

```
1  // Figura 9.13: fig09_13.cpp
2  // Demonstrando a ordem em que construtores e
3  // destrutores são chamados.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "CreateAndDestroy.h" // inclui a definição da classe CreateAndDestroy
9
10 void create( void ); // protótipo
11
12 CreateAndDestroy first( 1, "(global before main)" ); // objeto global
13
14 int main()
15 {
```



Objeto local automático criado em **main.**

```
16    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
17    CreateAndDestroy second( 2, "(local automatic in main)" );
18    static CreateAndDestroy third( 3, "(local static in main)" );
19
20    create(); // chama função para criar objetos
21
22    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
23    CreateAndDestroy fourth( 4, "(local automatic in main)" );
24    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
25    return 0;
26 } // fim de main
27
28 // função para criar objetos
29 void create( void )
30 {
31    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32    CreateAndDestroy fifth( 5, "(local automatic in create)" );
33    static CreateAndDestroy sixth( 6, "(local static in create)" );
34    CreateAndDestroy seventh( 7, "(local automatic in create)" );
35    cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
36 } // fim da função create
```

Objeto local **static** criado em **main.**

Objeto local automático criado em **main.**

Objeto local automático criado em **create.**

Objeto local **static** criado em **create.**

Objeto local automático criado em **create.**

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

9.10 Atribuição-padrão de membro a membro

- **Atribuição-padrão de membro a membro**
  - **Operador de atribuição (=)**
    - **Pode ser usado para atribuir um objeto a outro objeto do mesmo tipo.**
      - **Cada membro de dados do objeto à direita é atribuído ao mesmo membro de dados do objeto à esquerda.**
    - **Isso pode provocar sérios problemas quando os membros de dados contêm ponteiros para a memória alocada dinamicamente.**

```
1  // Figura 9.17: Date.h
2  // Declaração da classe Date.
3  // Funções-membro são definidas em Date.cpp
4
5  // impede múltiplas inclusões de arquivo de cabeçalho
6  #ifndef DATE_H
7  #define DATE_H
8
9  // definição da classe Date
10 class Date
11 {
12 public:
13    Date( int = 1, int = 1, int = 2000 ); // construtor-padrão
14    void print();
15 private:
16    int month;
17    int day;
18    int year;
19 }; // fim da classe Date
20
21 #endif
```

Date.cpp

```
1  // Figura 9.18: Date.cpp
2  // Definições de função-membro da classe Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // inclui a definição da classe Date a partir de Date.h
8
9  // construtor Date (deve fazer verificação de intervalo)
10 Date::Date( int m, int d, int y )
11 {
12    month = m;
13    day = d;
14    year = y;
15 } // fim do construtor Date
16
17 // imprime Date no formato mm/dd/aaaa
18 void Date::print()
19 {
20    cout << month << '/' << day << '/' << year;
21 } // fim da função print
```

```
1  // Figura 9.19: fig09_19.cpp
2  // Demonstrando que os objetos de classe podem ser atribuídos
3  // um ao outro utilizando atribuição-padrão de membro a membro.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Date.h" // inclui a definição da classe Date a partir de Date.h
9
10 int main()
11 {
12    Date date1( 7, 4, 2004 );
13    Date date2; // date2 assume padrão de 1/1/2000
14
15    cout << "date1 = ";
16    date1.print();
17    cout << "\ndate2 = ";
18    date2.print();
19
20    date2 = date1; // atribuição-padrão de membro a membro
21
22    cout << "\n\nAfter default memberwise assignment, date2 = ";
23    date2.print();
24    cout << endl;
25    return 0;
26 } // fim de main
```

A atribuição membro a membro atribui membros de dados de **`date1`** a **`date2`**.

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

`date2` agora armazena a mesma data como **`date1`**.

9.10 Atribuição-padrão de membro a membro (cont.)

- **Construtor de cópia**
  - **Permite que os objetos sejam passados por valor.**
    - **É usado para copiar valores originais do objeto em um novo objeto passado a uma função ou que retornou de uma função.**
  - **O compilador fornece um construtor-padrão de cópia.**
    - **Copia cada membro do objeto original no membro correspondente do novo objeto (ou seja, é uma atribuição de membro a membro).**
  - **Também pode provocar sérios problemas quando os membros de dados contêm ponteiros para memória alocada dinamicamente.**

Dica de desempenho 9.3

A passagem de um objeto por valor é adequada do ponto de vista de segurança, porque a função chamada não tem acesso ao objeto original no chamador, mas pode diminuir o desempenho ao fazer uma cópia de um objeto grande. É possível passar um objeto por referência passando um ponteiro ou uma referência ao objeto. A passagem por referência oferece bom desempenho, mas menor segurança, porque a função chamada recebe acesso ao objeto original. A passagem por referência `const` é uma alternativa segura de bom desempenho (pode ser implementada com um parâmetro de referência `const` ou com um parâmetro de ponteiro para dados `const`).