



BCC221

Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho

2014/2



UFOP

Endereços Importantes

- Site da disciplina:
<http://www.decom.ufop.br/marco/>
- Moodle:
www.decom.ufop.br/moodle
- Lista de e-mails:
bcc221-decom@googlegroups.com
- Para solicitar acesso:
<http://groups.google.com/group/bcc221-decom>



UFOP



Avisos



Avisos

UFOP



Na aula passada

UFOP

- Processo de Criação de um Programa C++
- Programando em C++
 - Operadores
 - Palavras reservadas
 - *cin*
 - *cout*
 - Blocos de código
 - Referências
 - Ponteiros e Alocação Dinâmica
 - Sobrecarga de Funções



Na aula de hoje

UFOP

- Classes
- Objetos
- Métodos
 - Construtores
 - Destrutores
 - Construtores Parametrizados e Vetores de Objetos
 - Objetos como Parâmetros de Métodos
 - Métodos que Retornam Objetos
- Separando a Interface da Implementação
- Composição: Objetos como Membros de Classes
- Funções Amigas
- Sobrecarga de Operadores
- O Ponteiro *This*



Introdução

UFOP

- Estamos acostumados a criar programas que:
 - Apresentam mensagens ao usuário;
 - Obtêm dados do usuário;
 - Realizam cálculos e tomam decisões.
- Todas estas ações eram delegadas à função *main* ou outras;
- A partir de agora, nossos programas terão uma função *main* e uma ou mais classes
 - Cada classe consistindo de dados e funções.



Introdução

UFOP

- Suponhamos que queremos dirigir um carro e acelerá-lo, de modo que ele fique mais veloz;
- Antes disto, alguém precisa **projetar** e **construir** o carro;
- O projeto do carro tipicamente começa com desenhos técnicos
 - Que incluem o pedal do acelerador que utilizaremos.



Introdução

UFOP

- De certa forma, o pedal do acelerador esconde os mecanismos complexos que fazem com que o carro acelere
 - Isto permite que pessoas sem conhecimento de mecânica acelerem um carro;
 - Acelerar é uma “interface” mais amigável com a mecânica do motor.
- Acontece que não podemos dirigir os desenhos técnicos
 - É necessário que alguém construa o carro, com o pedal.



Introdução

UFOP

- Depois de construído, o carro não andará sozinho
 - Precisamos apertar o pedal de acelerar.
- Vamos fazer uma analogia com programação orientada a objetos.



Introdução

UFOP

- Realizar uma tarefa em um programa requer uma função
 - O *main*, por exemplo.
- A função descreve os mecanismos que realizam a tarefa
 - Escondendo toda a complexidade do processo, assim como o acelerador.
- Começaremos pela criação de uma classe, que abriga uma função
 - Assim como o desenho técnico de um carro abriga um acelerador.



Introdução

UFOP

- Uma função pertencente a uma classe é chamada método
 - São utilizadas para desempenhar as tarefas de uma classe.
- Da mesma forma que não é possível dirigir o projeto de um carro, você não pode “dirigir” uma classe;
- É necessário antes construir o carro para dirigí-lo;
- É necessário criar um **objeto** de uma classe antes para poder executar as tarefas descritas por uma classe.



Introdução

UFOP

- Ainda, vários carros podem ser criados a partir do projeto inicial
 - Vários objetos podem ser criados a partir da mesma classe.
- Quando dirigimos um carro, pisar no acelerador manda uma mensagem para que o carro desempenhe uma tarefa
 - “Acelere o carro”.



Introdução

UFOP

- Similarmente, enviamos **mensagens** aos objetos
 - As **chamadas aos métodos**;
 - Dizemos aos métodos para desempenharem suas tarefas.
- Além das competências de um carro, ele possui diversos atributos
 - Número de passageiros;
 - Velocidade atual;
 - Nível do tanque de combustível;
 - Etc.



Introdução

UFOP

- Assim como as competências, os atributos também são representados no projeto de um carro
 - Cada carro possui seus atributos próprios;
 - Um carro não conhece os atributos de outro.
- Da mesma forma ocorre com os objetos
 - Cada objeto tem os seus próprios atributos.

Classes



Classes

UFOP

- Vejamos um exemplo de uma classe que descreve um “diário de classe”
 - Utilizado para manter dados sobre a avaliação de alunos.
- Vejamos também como criar objetos desta classe.

DIÁRIO DE CLASSE

ANO LETIVO DE 20_____
Nome do Estabelecimento _____

Cidade _____	Estado _____
Curso _____	Componente Curricular _____
Série _____	Turno _____
Professor(a) _____	

HORARIO

1º AULA	2º AULA	3º AULA	4º AULA	5º AULA	6º AULA	7º AULA	8º AULA	9º AULA	10º AULA	11º AULA	12º AULA	13º AULA	14º AULA	15º AULA	16º AULA	17º AULA	18º AULA	19º AULA	20º AULA	21º AULA	22º AULA	23º AULA	24º AULA	25º AULA	26º AULA	27º AULA	28º AULA	29º AULA	30º AULA	31º AULA	32º AULA	33º AULA	34º AULA	35º AULA	36º AULA	37º AULA	38º AULA	39º AULA	40º AULA	41º AULA	42º AULA	43º AULA	44º AULA	45º AULA	46º AULA	47º AULA	48º AULA	49º AULA	50º AULA	51º AULA	52º AULA	53º AULA	54º AULA	55º AULA	56º AULA	57º AULA	58º AULA	59º AULA	60º AULA	61º AULA	62º AULA	63º AULA	64º AULA	65º AULA	66º AULA	67º AULA	68º AULA	69º AULA	70º AULA	71º AULA	72º AULA	73º AULA	74º AULA	75º AULA	76º AULA	77º AULA	78º AULA	79º AULA	80º AULA	81º AULA	82º AULA	83º AULA	84º AULA	85º AULA	86º AULA	87º AULA	88º AULA	89º AULA	90º AULA	91º AULA	92º AULA	93º AULA	94º AULA	95º AULA	96º AULA	97º AULA	98º AULA	99º AULA	100º AULA
---------	---------	---------	---------	---------	---------	---------	---------	---------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	-----------

Old School Class		GRADE RECORD										
Fall, 2005-06		Mr. Tryon - Period 2										
1	Auburke, Allen	16	95	95	95	95	95	95	95	95	95	A
2	Bauer, Eric	15	95	95	95	95	95	95	95	95	95	A
3	Bauer, Jeremy	14	95	95	95	95	95	95	95	95	95	A
4	Bennett, Steve	9	95	95	95	95	95	95	95	95	95	A
5	Benson, Jason	15	95	95	95	95	95	95	95	95	95	A
6	Bergman, Leah	15	95	95	95	95	95	95	95	95	95	A
7	Bishop, Andrew	15	95	95	95	95	95	95	95	95	95	A
8	Bruce, Holly	9	95	95	95	95	95	95	95	95	95	A
9	Bruggeman, Leah	9	95	95	95	95	95	95	95	95	95	A
10	Cameron, Michael	15	95	95	95	95	95	95	95	95	95	A
11	Carey, Neva	19	95	95	95	95	95	95	95	95	95	A
12	Coburn, James	12	95	95	95	95	95	95	95	95	95	A
13	Davis, Dorothy	10	95	95	95	95	95	95	95	95	95	A
14	Ernestine, Natalie	9	95	95	95	95	95	95	95	95	95	A
15	Farmer, Freddie	9	95	95	95	95	95	95	95	95	95	A
16	George, Graham	19	95	95	95	95	95	95	95	95	95	A
17	Goldschein, Thomas	12	95	95	95	95	95	95	95	95	95	A
18	Hagan, Michael	12	95	95	95	95	95	95	95	95	95	A
19	Jacobson, Neil	10	95	95	95	95	95	95	95	95	95	A
20	Jacobsen, Heidi	11	95	95	95	95	95	95	95	95	95	A
21	Johnston, Heidi	11	95	95	95	95	95	95	95	95	95	A
22	Jordan, Leah	9	95	95	95	95	95	95	95	95	95	A
23	Kaaret, Karen	12	95	95	95	95	95	95	95	95	95	A
24	Kitterman, Shannon	10	95	95	95	95	95	95	95	95	95	A
25	Lettman, Sandy	12	95	95	95	95	95	95	95	95	95	A
26	Lindquist, Linda	12	95	95	95	95	95	95	95	95	95	A
27	Lindquist, Linda	12	95	95	95	95	95	95	95	95	95	A
28	Lindquist, Linda	12	95	95	95	95	95	95	95	95	95	A
29	Martin, Mary	9	95	95	95	95	95	95	95	95	95	A
30	Morris, Nata	10	95	95	95	95	95	95	95	95	95	A
31	O'Brien, Oscar	11	95	95	95	95	95	95	95	95	95	A
32	Reed, Leah	12	95	95	95	95	95	95	95	95	95	A
33	Smith, Sally	12	95	95	95	95	95	95	95	95	95	A
34	Turpin, Taylor	10	95	95	95	95	95	95	95	95	95	A
35	Quinton, Gwene	10	95	95	95	95	95	95	95	95	95	A
36	Reed, Leah	12	95	95	95	95	95	95	95	95	95	A
37	Allen, Arthur	12	95	95	95	95	95	95	95	95	95	A
38	Macdonald, Maria	12	95	95	95	95	95	95	95	95	95	A



Classes

UFOP

```
#include <iostream>
using namespace std;
```

```
// Definição da classe GradeBook
class GradeBook
{
    public:
        // método que exibe uma mensagem de boas-vindas ao usuário do GradeBook
        void displayMessage()
    {
        cout << "Welcome to the Grade Book!" << endl;
    } // fim do método displayMessage
}; // fim da classe GradeBook
```

```
// a função main inicia a execução do programa
int main()
{
    GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook
    myGradeBook.displayMessage(); // chama o método displayMessage do objeto
    return 0; // indica terminação bem-sucedida
} // fim do main
```



Classes

UFOP

- Definir uma classe é dizer ao compilador quais métodos e atributos pertencem à classe
 - A definição começa com a palavra **class**;
 - Em seguida, o nome da classe
 - Por padrão, a primeira letra de cada palavra no nome é maiúscula.
 - O corpo da classe é delimitado por **{** e **}**;
 - Não se esqueça do **;** no final.



Classes

UFOP

- Dentro da classe definimos os métodos e os atributos
 - Separados pelo **especificador de acesso** (visibilidade)
 - Basicamente, **público** (`public`) , **privado** (`private`) e **protegido** (`protected`).
 - O método `displayMessage` é público, pois está declarado depois deste especificador
 - Ou seja, pode ser chamado por outras funções do programa e por métodos de outras classes.
 - Especificadores de acesso são sempre seguidos de :



Métodos

UFOP

- A definição de um método se assemelha a definição de uma função
 - Possui valor de retorno;
 - Assinatura do método
 - Por padrão, começa com uma letra minúscula e todas as palavras seguintes começam com letras maiúsculas.
 - Lista de parâmetros entre parênteses (tipo e identificador)
 - Eventualmente, vazia
- Além disso, o corpo de um método também é delimitado por { e }.



Métodos

UFOP

- O corpo de um método contém as instruções que realizam uma determinada tarefa
 - Neste exemplo, simplesmente apresenta uma mensagem.



Classes

UFOP

```
#include <iostream>
using namespace std;

// Definição da classe GradeBook
class GradeBook
{
    public:
        // método que exibe uma mensagem de boas-vindas ao usuário do GradeBook
        void displayMessage()
        {
            cout << "Welcome to the Grade Book!" << endl;
        } // fim do método displayMessage
}; // fim da classe GradeBook

// a função main inicia a execução do programa
int main()
{
    GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook
    myGradeBook.displayMessage(); // chama o método displayMessage do objeto
    return 0; // indica terminação bem-sucedida
} // fim do main
```



Objetos

UFOP

- Para utilizarmos a classe *GradeBook* em nosso programa, precisamos criar um objeto
 - Há uma exceção em que é possível usar métodos sem criar objetos.
- Para criarmos um objeto, informamos o nome da classe como um tipo, e damos um identificador ao objeto;
- Uma vez criado o objeto, podemos utilizar seus métodos;
- Note que também podemos criar vetores e matrizes de objetos.



Métodos

UFOP

- Para utilizar um método, utilizamos o operador .
 - Informamos o nome do objeto, seguido por . e o nome do método e eventuais parâmetros
 - Ou somente o par de parênteses.
 - O método possui acesso aos atributos do objeto que o chamou
 - Não possui acesso aos atributos de outros objetos, a não ser que estes sejam passados por parâmetro;
 - Os atributos são acessados pelo nome definido na classe.



Exercício

UFOP

- Como seria o diagrama de classe UML para a classe *GradeBook*?





Classe *string*

UFOP

- Como dito anteriormente, existem várias classes prontas, com métodos úteis para várias finalidades
 - Uma delas é a classe *string*, utilizada para manipulação de *strings*.
- Vamos incorporar a classe *string* em nosso exemplo anterior
 - Como um parâmetro para o método do exemplo.



Classes

UFOP

```
#include <iostream>
#include <string> // o programa utiliza classe de string padrão C++
using namespace std;

// Definição da classe GradeBook
class GradeBook
{
public:
    // função que exibe uma mensagem de boas-vindas ao usuário do GradeBook
    void displayMessage( string courseName )
    {
        cout << "Welcome to the grade book for\n" << courseName << "!"
            << endl;
    } // fim da função displayMessage
}; // fim da classe GradeBook

// a função main inicia a execução do programa
int main()
{
    string nameOfCourse; // strings de caracteres para armazenar o nome do curso
    GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook

    cout << "Please enter the course name:" << endl;
    getline( cin, nameOfCourse ); // lê o nome de um curso com espaços em branco
    cout << endl; // gera saída de uma linha em branco
    // passa nameOfCourse como um argumento
    myGradeBook.displayMessage( nameOfCourse );
    return 0; // indica terminação bem-sucedida
} // fim de main
```



Classe *string*

UFOP

- Para utilizarmos a classe *string*, precisamos incluir o arquivo **<string>**
 - Depois podemos criar um objeto desta classe.
- Para lermos uma *string* com espaços em branco, utilizamos a instrução **getline**.



getline

UFOP

- Existem duas sintaxes para a instrução **getline**
 - Uma lê caracteres até que seja encontrado o final da linha
getline(cin, nameOfCourse);
 - A outra lê caracteres até que seja encontrado um caractere de terminação especificado por nós
getline(cin, nameOfCourse, 'A');
 - Neste exemplo, a instrução lê caracteres até achar um '**A**', que não será incluído na *string*.



Métodos

UFOP

- Métodos podem receber parâmetros assim como as funções
 - Basta definir o tipo do parâmetro e seu identificador entre parênteses
 - Em C++ existe flexibilidade entre o tipo do argumento enviado e o tipo do argumento recebido pelo método.
 - Se não houver parâmetros, deixamos os parênteses sem conteúdo.



Exercício

UFOP

- Como fica o diagrama de classe UML para a classe *GradeBook* agora que temos um parâmetro para o método *displayMessage*?





Atributos

UFOP

- Nossa classe não possui atributos, apenas um método;
- Atributos são representados como variáveis na definição de uma classe
 - Declarados dentro da classe
 - Porém, fora dos métodos.
- Cada objeto da classe possui sua própria cópia dos atributos.



Getters e Setters

UFOP

- Atributos são definidos como **privados**
 - Logo, só podem ser alterados dentro da própria classe
 - Tentar acessá-los fora da classe causará erro.
 - **Ocultação de informação;**
 - Um método que altere o valor de um atributo é chamado de **setter**
 - Por padrão, a nomenclatura é *set+[nome do atributo]*.
 - Um método que retorne o valor de um atributo é chamado de **getter**
 - Por padrão, a nomenclatura é *get+[nome do atributo]*.



Getters e Setters

UFOP

- Uma vez que nossa classe possui *getter* e *setter*, os atributos só devem ser acessados e alterados por eles
 - Mesmo dentro de outros métodos que porventura necessitem acessar/alterar os atributos.
- Vamos alterar nosso exemplo anterior para que a *string* utilizada seja agora um atributo
 - Com *getter* e *setter*.



Classes

UFOP

```
class GradeBook
{
public:
    // função que configura o nome do curso
    void setCourseName( string name )
    {
        courseName = name; // armazena o nome do curso no objeto
    } // fim da função setCourseName

    // função que obtém o nome do curso
    string getCourseName()
    {
        return courseName; // retorna o courseName do objeto
    } // fim da função getCourseName

    // função que exibe uma mensagem de boas-vindas
    void displayMessage()
    {
        // essa instrução chama getCourseName para obter o
        // nome do curso que esse GradeBook representa
        cout << "Welcome to the grade book for\n" << getCourseName() << "!"
            << endl;
    } // fim da função displayMessage
private:
    string courseName; // nome do curso para esse GradeBook
}; // fim da classe GradeBook
```



Classes

UFOP

```
int main()
{
    string nameOfCourse; // strings de caracteres para armazenar o nome do curso
    GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook

    // exibe valor inicial de courseName
    cout<<"Initial course name is: "<<myGradeBook.getCourseName()
    <<endl;

    // solicita, insere e configura o nome do curso
    cout << "\nPlease enter the course name:" << endl;
    getline( cin, nameOfCourse ); // lê o nome de um curso com espaços em branco
    myGradeBook.setCourseName( nameOfCourse ); //configura o nome do curso

    cout << endl; // gera saída de uma linha em branco
    myGradeBook.displayMessage(); // exibe a mensagem com o novo nome do curso
    return 0; // indica terminação bem-sucedida
}
```



Getters e Setters

UFOP

- Para garantir a consistência dos valores dos atributos, convém realizar **validação de dados** nos *getters* e *setters*
 - O valor passado como parâmetro é adequado em relação ao tamanho ou faixa de valores?
 - Temos que definir se permitiremos valores negativos, tamanho máximo para vetores, etc.
 - Se um parâmetro for inadequado, devemos atribuir um valor padrão ao atributo
 - Zero, um, NULL, '\0', etc.



Classe *string*

UFOP

- Notem um detalhe interessante:
 - Quando usamos a classe *string*, podemos fazer atribuição direta entre os objetos, utilizando o operador de atribuição
 - **De fato, podemos fazer atribuição direta entre quaisquer objetos de uma mesma classe**
 - O que pode causar erros se entre os atributos possuirmos ponteiros para memória alocada dinamicamente.
 - Diferentemente do que ocorre com vetores de caracteres, que precisam da função *strcpy*.



Exercício

UFOP

- Como fica o diagrama de classe UML para a classe *GradeBook* agora que temos um *getter* e um *setter*?





UFOP



**Continua na
próxima aula...**

Construtores



Construtores

UFOP

- Quando um objeto da classe *GradeBook* é criado, a sua cópia do atributo *courseName* é inicializada como vazia
 - Por padrão.
- Mas e se quiséssemos que o atributo fosse inicializado com um valor padrão?
 - Podemos criar um método **construtor**, para inicializar cada objeto criado.



Construtores

- Um construtor é um método especial, definido com o **mesmo nome da classe** e executado automaticamente quando um objeto é criado
 - Não retorna valores;
 - Não possui valor de retorno;
 - Deve ser declarado como público.
- Se não especificarmos um construtor, o compilador utilizará o construtor padrão
 - No nosso exemplo, foi utilizado o construtor padrão da classe *string*, que a torna vazia.
- Vejamos nosso exemplo, agora com um construtor.



Construtores

UFOP

```
class GradeBook
{
public:
    // o construtor inicializa courseName com a string fornecida como argumento
    GradeBook( string name )
    {
        setCourseName( name ); // chama a função set para inicializar courseName
    } // fim do construtor GradeBook

    void setCourseName( string name )
    {
        courseName = name;
    }

    string getCourseName()
    {
        return courseName;
    }

    void displayMessage()
    {
        cout << "Welcome to the grade book for\n" << getCourseName()
           << "!" << endl;
    }

private:
    string courseName;
};
```



Construtores

UFOP

```
int main()
{
    // cria dois objetos GradeBook
    GradeBook gradeBook1("BCC221 - POO");
    GradeBook gradeBook2("BCC202 - AED's I");

    cout << "gradeBook1 created for course: "
        << gradeBook1.getCourseName()
        << "\ngradeBook2 created for course: "
        << gradeBook2.getCourseName()<< endl;

    return 0;
}
```



Construtores

UFOP

- Notem que um construtor pode possuir parâmetros ou não
 - Por exemplo, poderíamos não passar nenhum parâmetro e definir um valor padrão dentro do próprio construtor.
- Quando um atributo for objeto de outra classe, podemos chamar o construtor da outra classe em um construtor definido por nós
 - E opcionalmente, especificar inicializações adicionais.
- Todas nossas classes devem possuir construtores, para evitarmos lixo em nossos atributos.



Construtores

UFOP

- É possível criarmos mais de um construtor na mesma classe
 - Sobrecarga de construtores
 - O **construtor default** não possui parâmetros.
 - Da mesma forma que sobrecregamos funções;
 - A diferenciação é feita pelo número de parâmetros enviados no momento da criação do objeto
 - Diferentes objetos de uma mesma classe podem ser inicializados por construtores diferentes.
 - Escolhemos qual construtor é mais adequado a cada momento.



Construtores

UFOP

- Podemos ainda ter construtores com parâmetros padronizados
 - O construtor recebe parâmetros para inicializar atributos;
 - Porém, define parâmetros padronizados, caso não receba nenhum parâmetro.
- Suponha uma classe *Venda*, em que temos os atributos valor e peças
 - Ao criar um objeto, o programador pode definir a quantidade de peças e o valor da venda;
 - Porém, se nada for informado, inicializaremos os atributos com o valor -1, usando o mesmo construtor;
 - É uma forma de economizar o trabalho de sobrecarregar um construtor.



Construtores

```
class Vendas
{
public:
//parametros padrão
Vendas(int p= -1, float v= -1.0)
{
    valor = v;
    pecas = p;
}
float getValor()
{
    return valor;
}
int getPecas()
{
    return pecas;
}
private:
float valor;
int pecas;
};
```

```
int main()
{
//inicializa um objeto com -1 e outro com 10
Vendas a, b(10, 10);

cout <<a.getPecas()<<endl
<<a.getValor()<<endl
<<b.getPecas()<<endl
<<b.getValor()<<endl;
return 0;
}
```



Exercício

UFOP

- Como fica o diagrama de classe UML para a classe *GradeBook* agora que temos um construtor?
 - **Dica:** para diferenciar um construtor em relação aos outros métodos, escrevemos «constructor» antes de seu nome.
 - **Nota:** geralmente construtores são omitidos em diagramas de classes.



Destruidores



Destruidores

UFOP

- De forma análoga aos construtores, que inicializam objetos, temos os **destrutores**, que **finalizam objetos**
 - São chamados automaticamente quando um objeto for destruído, por exemplo, ao terminar o seu bloco de código;
 - São indicados por um `~` antes do nome do método, que deve ser igual ao da classe.



Destruidores

UFOP

■ Destruidores:

- Não possuem valor de retorno;
- Não podem receber argumentos;
- Não podem ser chamados explicitamente pelo programador.

■ Atenção!

- Se um programa terminar por uma chamada ***exit()*** ou ***abort()***, o destrutor não será chamado.



Destruidores

UFOP

- Vejamos um exemplo em que o construtor de uma classe incrementa um atributo a cada vez que um objeto é criado e o destrutor decrementa o mesmo atributo a cada vez que um objeto é destruído;
- Como seria possível se cada objeto possui uma cópia diferente de cada atributo?
 - Usamos o modificador ***static***, que faz com que haja apenas um atributo compartilhado por todos os objetos.



Destruidores

UFOP

```
class Rec
{
    private:
        static int n;//cria um único item para todos os objetos
    public:
        Rec()
        {
            n++;
        }

        int getRec()
        {
            return n;
        }

        ~Rec()
        {
            n--;
        }
};
```



Destruidores

```
int Rec::n=0;//necessário para que o compilador crie a variável
```

```
int main()
{
    Rec r1, r2, r3;
    cout<<r1.getRec()<<endl;

    {
        Rec r4, r5, r6;//só valem dentro deste bloco
        cout<<r1.getRec()<<endl;
    }

    cout<<r1.getRec();

    return 0;
}
```



Destruidores

UFOP

- Construtores e destrutores são especialmente úteis quando os objetos utilizam alocação dinâmica de memória
 - Alocamos a memória no construtor;
 - Desalocamos a memória no destrutor.
- Novamente, destrutores são geralmente omitidos em diagramas de classes UML.



Exercício

UFOP

- Como fica o diagrama de classe UML para a classe *Rec* agora que temos um destrutor?
 - Dica: para diferenciar um construtor em relação aos outros métodos, escrevemos «*destructor*» antes de seu nome.



Construtores Parametrizados e Vetores de Objetos

Construtores Parametrizados e Vetores de Objetos



UFOP

- No caso de termos um vetor de objetos, recomenda-se não utilizar construtores parametrizados
 - Ou então utilizar construtores com parâmetros padronizados.
- Caso seja realmente necessário, no momento da declaração do vetor é necessário inicializá-lo, fazendo a atribuição de **objetos anônimos**
 - De forma parecida com a inicialização de vetores de tipos primitivos;
 - Cada objeto anônimo deve enviar seus parâmetros para o construtor.

Construtores Parametrizados e Vetores de Objetos



UFOP

```
#include<iostream>
using namespace std;

class Numero
{
public:
    Numero(int n)
    {
        valor = n;
    }
    int getNumero()
    {
        return valor;
    }
private:
    int valor;
};
```

```
int main()
{
    Numero vet[3] = {Numero(0),
                     Numero(1), Numero(2)};
    int i;

    for(i=0; i<3; i++)
        cout<<vet[i].getNumero()<<endl;

    return 0;
}
```

Objetos como Parâmetros de Métodos

Objetos como Parâmetros de Métodos



UFOP

- Entre os parâmetros que um método pode receber, podemos incluir objetos
 - Como dito anteriormente, um método só possui acesso aos atributos do objeto que o chamou;
 - E se precisarmos acessar os atributos de outros objetos?
 - Podemos passá-los como parâmetros.
 - Note que para o método acessar os atributos de outros objetos é necessário a utilização do operador .
- Suponha uma classe *Venda*, em que temos os atributos valor e peças
 - Deseja-se totalizar os valores e as peças de uma venda.

Objetos como Parâmetros de Métodos



UFOP

```
class Vendas
{
    public:
        void setValor(float preco)
        {
            valor = preco;
        }
        void setPecas(int quantidade)
        {
            pecas = quantidade;
        }
        float getValor()
        {
            return valor;
        }
        int getPecas()
        {
            return pecas;
        }
}
```

```
void totaliza(Vendas v[], int n)
{
    int i;
    valor = 0; //evita lixo
    pecas = 0; //evita lixo
    for(i=0; i<n; i++)
    {
        valor+=v[i].getValor();
        pecas+=v[i].getPecas();
    }
}

private:
float valor;
int pecas;
};
```

Objetos como Parâmetros de Métodos



UFOP

```
int main()
{
    Vendas total, v[5];
    v[0].setPecas(1);
    v[1].setPecas(2);
    v[2].setPecas(3);
    v[3].setPecas(4);
    v[4].setPecas(5);
    v[0].setValor(1.0);
    v[1].setValor(2.0);
    v[2].setValor(3.0);
    v[3].setValor(4.0);
    v[4].setValor(5.0);
    total.totaliza(v, 5);
    cout<<total.getPecas()<<endl<<total.getValor();
}
```

Métodos que Retornam Objetos



Métodos que Retornam Objetos

UFOP

- Podemos modificar nosso exemplo anterior para retornar um objeto com a totalização dos valores
 - Devemos definir o tipo de retorno como sendo um objeto da classe;
 - Algum objeto deve receber o valor retornado.



Métodos que Retornam Objetos

```
Vendas totaliza(Vendas v[], int n)
{
    int i;
    Vendas temp;

    temp.valor = 0; //evita lixo
    temp.pecas = 0; //evita lixo

    for(i=0; i<n; i++)
    {
        temp.valor+=v[i].getValor()
        temp.pecas+=v[i].getPecas();
    }

    return temp;
}
```

```
int main()
{
    Vendas total, v[5];
    v[0].setPecas(1);
    v[1].setPecas(2);
    v[2].setPecas(3);
    v[3].setPecas(4);
    v[4].setPecas(5);
    v[0].setValor(1.0);
    v[1].setValor(2.0);
    v[2].setValor(3.0);
    v[3].setValor(4.0);
    v[4].setValor(5.0);

    total = v[0].totaliza(v, 5);

    cout<<total.getPecas()
       <<endl
       <<total.getValor();
}
```

Separando a Interface da Implementação

Separando a Interface da Implementação



UFOP

- Uma vantagem de definir classes é que, quando empacotadas apropriadamente, elas podem ser reutilizadas
 - Como por exemplo, a classe *string*.
- Nossa exemplo não pode ser reutilizado em outro programa
 - Já contém um *main*, e todo programa deve possuir apenas um *main*.
- Claramente, colocar um *main* no mesmo arquivo que contém uma classe impede sua reutilização
 - Para resolver isto, separamos os arquivos.



Headers

UFOP

- A classe fica em um arquivo de cabeçalhos **.h** (*header*)
 - Lembre-se que ao final da classe colocamos ;
- O *main* fica em um arquivo de código-fonte **.cpp** (*source*);
- Desta forma, o arquivo **.cpp** deve incluir o arquivo **.h** para reutilizar o código
 - Compilamos apenas o arquivo **.cpp**.
- Vejamos como fica nosso exemplo.



GradeBook.h

UFOP

```
#include <iostream>
#include <string>
using namespace std;

class GradeBook
{
public:
    GradeBook( string name )
    {
        setCourseName( name );
    }

    void setCourseName( string name )
    {
        courseName = name;
    }

    string getCourseName()
    {
        return courseName;
    }

    void displayMessage()
    {
        cout<<"Welcome to the grade book for\n" <<getCourseName() <<"!"<<endl;
    }

private:
    string courseName;
};
```



Main.cpp

UFOP

```
#include <iostream>
using namespace std;
#include "GradeBook.h" // inclui a definição de classe GradeBook

int main()
{
    // cria dois objetos GradeBook
    GradeBook gradeBook1("BCC221 - POO");
    GradeBook gradeBook2("BCC202 – AED's I");

    cout << "gradeBook1 created for course: "
        << gradeBook1.getCourseName()
        << "\ngradeBook2 created for course: "
        << gradeBook2.getCourseName()<< endl;
    return 0;
}
```



Separando a Interface da Implementação

UFOP

- Um problema relacionado a esta divisão de arquivos é que o usuário da classe vai conhecer a implementação
 - O que não é recomendável.
- Permite que o usuário escreva programas baseado em detalhes da implementação da classe
 - Quando na verdade deveria apenas saber quais métodos chamar, sem saber seu funcionamento;
 - Se a implementação da classe for alterada, o usuário também precisará alterar seu programa.
- Podemos então separar a **interface** da **implementação**.



Interfaces

UFOP

- A **interface** de uma classe especifica quais serviços podem ser *utilizados* e como *requisitar* estes serviços
 - E não como os serviços são realizados.
- A interface pública de uma classe consiste dos métodos públicos
 - Em nosso exemplo, o construtor, o *getter*, o *setter* e o método *displayMessage*.



Interfaces

- Separamos então nossa classe em dois arquivos:
 - A definição da classe e protótipos dos métodos são feitos no arquivo **.h**;
 - A implementação dos métodos é definida em um arquivo **.cpp** separado;
 - Por convenção os arquivos possuem o mesmo nome, diferenciados apenas pela extensão.
- O *main* é criado em um terceiro arquivo, também com extensão **.cpp**
 - *GradeBook.h*
 - *GradeBook.cpp*
 - *Main.cpp*



GradeBook.h

UFOP

```
#include <string>
using namespace std;

// Definição da classe GradeBook
class GradeBook
{
    public:
        GradeBook( string );
        void setCourseName( string );
        string getCourseName();
        void displayMessage();
    private:
        string courseName;
};
```



GradeBook.h

UFOP

- Na definição da classe, temos apenas a declaração dos atributos e dos protótipos dos métodos
 - Apenas o cabeçalho dos métodos
 - Sempre terminados com ;
 - Note que não é necessário definir um nome para os atributos, apenas o tipo.



GradeBook.cpp

UFOP

```
#include <iostream>
using namespace std;
#include "GradeBook.h" // inclui a definição de classe GradeBook

GradeBook::GradeBook( string name )
{
    setCourseName( name );
}

void GradeBook::setCourseName( string name )
{
    courseName = name;
}

string GradeBook::getCourseName()
{
    return courseName;
}

void GradeBook::displayMessage()
{
    cout << "Welcome to the grade book for\n" << getCourseName()
        << "!" << endl;
}
```



GradeBook.cpp

UFOP

- No arquivo de definição dos métodos deve ser incluído o arquivo .h com a definição da classe;
- Note que após o tipo de cada método, incluímos o nome da classe seguido de ::
 - Operador de resolução de escopo
 - “Amarra” a implementação ao protótipo definido no outro arquivo;
 - Sem isto, serão considerados como funções, não relacionadas à classe.
- Na definição dos métodos é necessário dar nomes aos parâmetros.



Main.cpp

UFOP

```
#include <iostream>
using namespace std;

#include "GradeBook.h" //inclui a definição de classe GradeBook

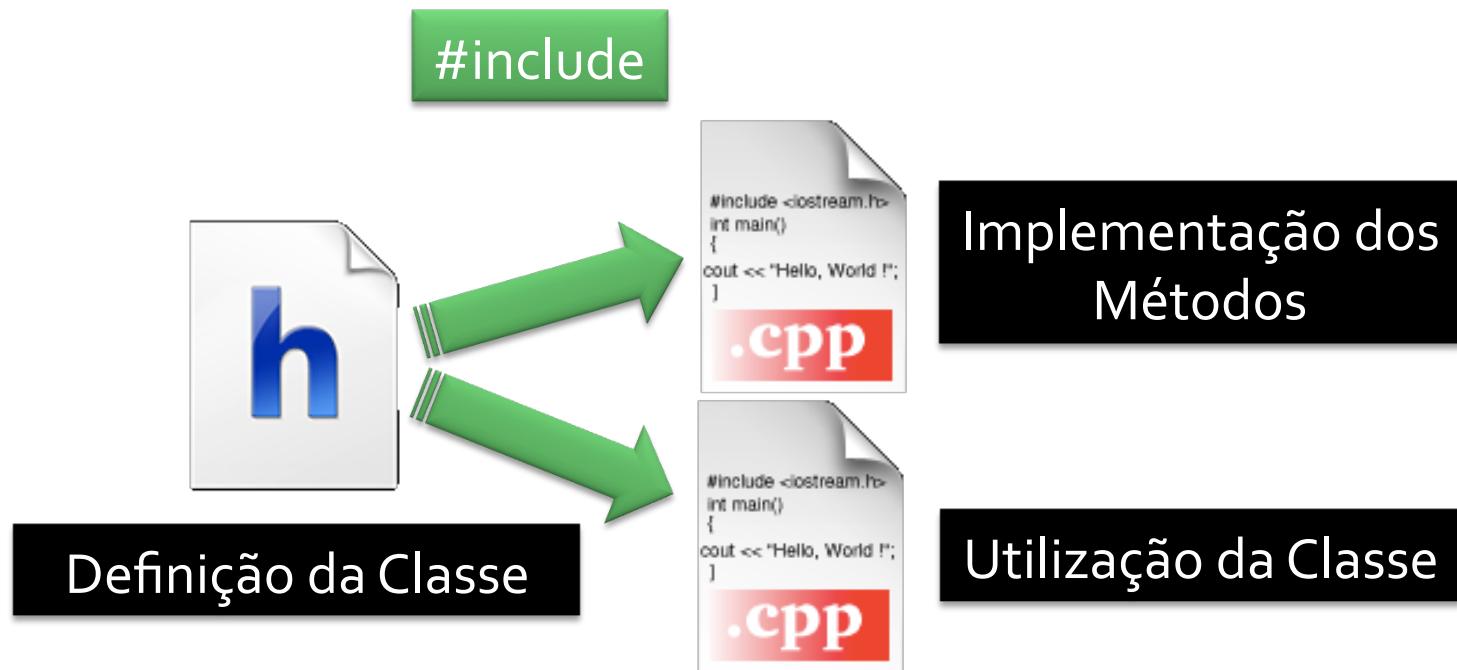
int main()
{
    GradeBook gradeBook1("BCC221 - POO");
    GradeBook gradeBook2("BCC202 - AED's I");

    cout << "gradeBook1 created for course: "
        <<gradeBook1.getCourseName()
        << "\ngradeBook2 created for course: "
        <<gradeBook2.getCourseName()<< endl;
    return 0;
}
```



Interfaces – Inclusão de Arquivos

UFOP





Interfaces - Compilação

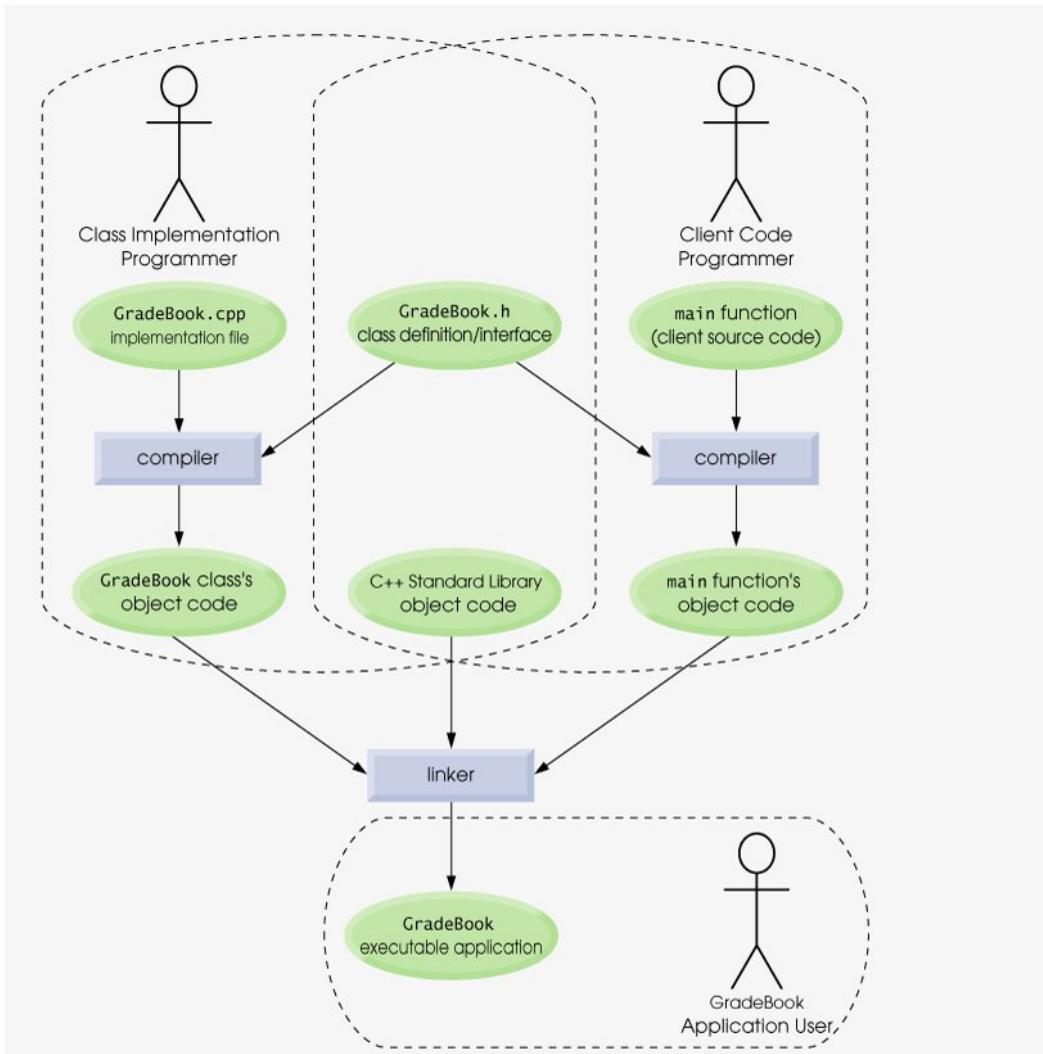
UFOP

- Quando separados desta forma, precisamos compilar três vezes
 - `g++ -c Classe.cpp`
 - Gera um arquivo **Classe.o**, que deve ser entregue ao usuário da classe;
 - Precisa dos arquivos **Classe.h** e **Classe.cpp**.
 - `g++ -c Main.cpp`
 - Gera um arquivo **main.o**;
 - Precisa dos arquivos **Classe.h** e **main.cpp**.
 - `g++ Classe.o main.o -o programa`
 - Gera o executável;
 - Não precisa do arquivo **Classe.cpp**, escondendo a implementação.



Interfaces - Compilação

UFOP





Interfaces - Compilação

UFOP

- Uma alternativa para compilar múltiplos arquivos é utilizar ***makefiles***
 - É uma forma automatizada para executar tarefas em terminal;
 - No próximo slide há um *Makefile* para realizar a compilação exemplificada anteriormente
 - O conteúdo deve ser salvo em um arquivo chamado ***Makefile*** (sem extensão);
 - Os arquivos devem se chamar ***main.cpp*** e ***metodos.cpp***;
 - Para executar, digite ***make*** no terminal, dentro da pasta onde estão os arquivos;
 - O resultado será um arquivo executável chamado ***programa***.



Makefile

UFOP

```
OBJS=main.o metodos.o
all: programa
programa: $(OBJS)
        g++ $(OBJS) -o @@
main.o: main.cpp
        g++ -c main.cpp -o main.o -Wall
metodos.o: metodos.cpp
        g++ -c $< -o $@ -Wall
```

Composição: Objetos Como Membros de Classes



Composição

UFOP

- Uma classe hipotética *RelogioComAlarme* deve saber o horário para soar o alarme
 - Então ele pode incluir um objeto da classe hipotética *Relogio*.
- Este relacionamento é do tipo “**tem um**” e é denominado **composição**;
- Vejamos um exemplo com estas duas classes hipotéticas.



Relogio.h

UFOP

```
#include<iostream>
using namespace std;

class Relogio
{
    public:
        Relogio(int, int, int);
        void setRelogio(int, int, int);
        void printRelogio();

    private:
        int h, m, s;
};
```



Relogio.cpp

UFOP

```
#include<iostream>
using namespace std;
#include "Relogio.h"

Relogio::Relogio(int hr=0, int min=0, int seg=0)
{
    setRelogio (hr, min, seg);
}
void Relogio::setRelogio(int hr, int min, int seg)
{
    h = hr;
    m = min;
    s = seg;
}
void Relogio::printRelogio()
{
    cout<<h<<":"<<m<<":"<<s;
```



Composição

UFOP

- Note que quando temos um construtor com parâmetros padronizados e separamos a interface da implementação, só utilizamos os parâmetros padronizados na implementação;
- Depois de definida e implementada a classe *Relogio*, podemos utilizar objetos dela em outra classe
 - Caracterizando a **composição** ou **agregação**;
 - Qual a diferença entre as duas?



RelogioComAlarme.h

```
#include<iostream>
#include<string>
using namespace std;
#include"Relogio.h"

class RelogioComAlarme
{
    public:
        RelogioComAlarme();
        void setAlarme(string, bool, int, int, int);
        void printAlarme();

    private:
        bool ligado;
        Relogio alarme;
        string tom;
};
```



Composição

UFOP

- Na classe *RelogioComAlarme*, um dos atributos é um objeto da classe *Relogio*
 - Quando um objetos *RelogioComAlarme* for destruído, o objeto *Relogio* também será;
 - Caracterizando assim uma **composição**.



RelogioComAlarme.cpp

UFOP

```
#include<iostream>
#include<string>
using namespace std;
#include"RelogioComAlarme.h"

RelogioComAlarme::RelogioComAlarme():alarme(10, 10, 10)
{
    ligado = false;
    tom = "Battery";
}

void RelogioComAlarme::setAlarme(string musica, bool flag, int h, int m, int s)
{
    ligado = flag;
    alarme.setRelogio(h, m, s);
    tom = musica;
}

void RelogioComAlarme::printAlarme()
{
    alarme.printRelogio();
}
```



Composição

UFOP

- Há um detalhe importante no construtor da classe *RelogioComAlarme*:
 - Precisamos chamar o construtor do objeto da classe *Relogio* também;
 - Fazemos isso depois da assinatura da implementação do construtor;
 - Colocamos : e depois chamamos o construtor do objeto da composição
 - Mesmo que não haja parâmetros.
- O objeto da composição pode ser utilizado normalmente, chamando seus próprios métodos
 - Não é possível acessar os membros privados do objeto da composição, exceto por *getters* e *setters*.



driverRelogioComAlarme.cpp

UFOP

```
#include<iostream>
using namespace std;
#include"RelogioComAlarme.h"

int main()
{
    RelogioComAlarme despertador;
    despertador.setAlarme("Enter Sandman", true, 6, 0, 0);
    despertador.printAlarme();
    return 0;
}
```



Composição

UFOP

- Novamente, compilamos as implementações das classes usando a *flag* –c
 - Depois compilamos o programa principal;
 - g++ -c Relogio.cpp;
 - g++ -c RelogioComAlarme.cpp;
 - g++ *.o driverRelogioComAlarme.cpp –o programa.



UFOP



**Continua na
próxima aula...**

Funções Amigas



Funções Amigas

UFOP

- Uma **função amiga** de uma classe é uma função definida completamente fora da classe
 - Porém, possui acesso aos membros públicos e não públicos de uma classe;
 - Funções isoladas ou mesmo classes inteiras podem ser declaradas como amigas de outra classe.
 - Funções amigas podem melhorar a performance de uma aplicação, e também são utilizadas na **sobrecarga de operadores** e na criação de **iteradores**.
- Podemos também declarar uma **classe amiga**
 - Todos os métodos terão acesso aos membros da outra classe.



Funções Amigas

- Para declararmos uma função amiga, utilizamos a palavra ***friend*** antes do protótipo da função dentro da classe
 - Note que a função não será um método.
- Se uma classe *ClasseUm* será declarada como amiga de uma classe *ClasseDois*, dentro de *ClasseUm* declaramos

friend class ClasseDois;

- **Atenção!**
 - Estas declarações devem ser feitas antes de qualquer especificador de acesso (*public*, *private*, etc.).



RelogioFriend.h

UFOP

```
#include<iostream>
using namespace std;

class Relogio
{
    friend void alteraHMS(Relogio &r);
public:
    Relogio(int, int, int);
    void setHora(int, int, int);
    void printHora();

private:
    int h, m, s;
};
```



RelogioFriend.cpp

UFOP

```
#include<iostream>
using namespace std;
#include "RelogioFriend.h"

Relogio::Relogio(int hr=0, int min=0, int seg=0)
{
    setHora(hr, min, seg);
}
void Relogio::setHora(int hr, int min, int seg)
{
    h = hr;
    m = min;
    s = seg;
}
void Relogio::printHora()
{
    cout<<h<<'.'<<m<<'.'<<s<<endl;
}

//passagem por referência
void alteraHMS(Relogio &r)
{
    r.h = 10;
    r.m = 10;
    r.s = 10;
}
```



Funções Amigas

UFOP

- Note que o objeto deve ser enviado como parâmetro para a função
 - Uma vez que o objeto não chama a função, é necessário indicar qual é o objeto cujos atributos serão alterados;
 - Para que a alteração tenha efeito, precisamos enviar o objeto por referência
 - Por isso o uso do operador &.
- Qualquer outra tentativa de acesso a membros privados por funções que não são amigas de uma classe resultará em erro de compilação.



driverRelogioFriend.cpp

UFOP

```
#include<iostream>
using namespace std;
#include"RelogioFriend.h"

int main()
{
    Relogio r(12,0,0);
    r.printHora();
    alteraHMS(r); //é necessário enviar o objeto como parâmetro
    r.printHora();

    return 0;
}
```

Sobrecarga de Operadores



Sobrecarga de Operadores

UFOP

- Como vimos até agora, operações relacionadas a objetos são realizadas através da chamada de métodos
 - Porém, podemos utilizar os operadores da linguagem C++ para especificar operações comuns de manipulação de objetos;
 - Novos operadores não podem ser criados;
 - A adaptação dos operadores nativos da linguagem para nossas classes é chamada de **sobrecarga de operadores**
 - Similar à sobrecarga de funções e métodos;
 - Dependendo dos operandos envolvidos, o operador se comporta diferentemente;
 - Ou seja, se torna **sensível ao contexto**.



Sobrecarga de Operadores

UFOP

- Por exemplo, os operadores aritméticos são sobrecarregados por padrão
 - Funcionam para quaisquer tipo de número (*int*, *float*, *double*, *long*, etc.);
 - Outro operador que sobrecarregado é o de atribuição.
- Qualquer operação realizada por um operador sobrecarregado também pode ser realizada por um método sobrecarregado
 - No entanto, a notação de operador é muito mais simples e natural.
- Um bom exemplo de classe com operadores sobrecarregados é a *string*.



Sobrecarga de Operadores

UFOP

- A principal convenção a respeito da sobrecarga de operadores é a de que o comportamento do operador utilizado seja análogo ao original
 - Ou seja, `+` só deve ser utilizado para realizar adição.
- Para sobrestrar um operador criamos um método ou função global cujo nome será ***operator***, seguido pelo símbolo do operador a ser sobrestrado
 - Por exemplo, `operator +();`



Sobrecarga de Operadores

UFOP

■ Atenção!

- Por padrão, não se sobrecarrega os operadores `=`, `,` e `&`;
- Os operadores `.` , `.*` , `?:` e `::` não podem ser sobrecarregados;
- Não é possível sobrecarregar um operador para lidar com tipos primitivos
 - Soma de inteiros não pode ser alterada.
- Não é possível alterar a precedência de um operador através de sobrecarga;
- O número de operandos de um operador não pode ser alterado através de sobrecarga;
- Sobrecarregar um operador não sobrecarrega os outros
 - Sobrecarregar `+` não sobrecarrega `+=`;



Sobrecarga de Operadores

UFOP

■ Métodos vs. Funções

- Quando sobrecarregamos um operador usando um método, o operando da esquerda deve sempre ser um objeto
 - Somente neste caso será utilizado o operador sobrecarregado.
- Se não possuirmos certeza sobre isto, devemos usar uma função
 - Permitirá a comutatividade.
 - Se for necessário acessar os atributos de um objeto, declaramos a função como **amiga**.
- Os operadores (), [], -> e qualquer operador de atribuição devem ser sobrecarregados como métodos.



Sobrecarga de Operadores

UFOP

- Podemos, por exemplo, sobrecarregar o operador `>>` utilizado para leitura de dados
 - Note que o operando do lado esquerdo é um objeto da classe ***istream***, da biblioteca padrão C++
 - Não podemos alterá-la para incluir a sobreposição;
 - Mas podemos criar uma função que altere o operador do lado direito.
- Por exemplo, vamos sobrecarregar os operadores `>>` e `<<` para objetos que representam números de telefone
 - Criaremos uma classe para representar os número de telefone com **funções amigas** que sobreponham os operadores `>>` e `<<`.



NumeroTelefone.h

UFOP

```
#include <iostream>
#include <string>
using namespace std;

class NumeroTelefone
{
    friend ostream &operator<<( ostream &, const NumeroTelefone & );
    friend istream &operator>>( istream &, NumeroTelefone & );

private:
    string DDD;      // código de área de 2 algarismos
    string inicio;   // linha de 4 algarismos
    string fim;       // linha de 4 algarismos
};
```



Sobrecarga de Operadores

UFOP

friend ostream &**operator<<**(ostream &, **const** NumeroTelefone &);

- A sobrecarga do operador **<<** é implementada através de uma função amiga, que retorna uma referência a um objeto *ostream* (fluxo de saída) e recebe como parâmetros referências para um objeto *ostream* e para um objeto *NumeroTelefone*;

friend istream &**operator>>**(istream &, NumeroTelefone &);

- A sobrecarga do operador **>>** é implementada através de uma função amiga, que retorna uma referência a um objeto *istream* (fluxo de entrada) e recebe como parâmetros referências para um objeto *istream* e para um objeto *NumeroTelefone*.



Sobrecarga de Operadores

UFOP

- O fato de ambas as funções retornarem um objeto, além de alterarem os atributos do objeto da classe *NumeroTelefone* permite construções do tipo:
 - `cin>>a;`
 - `cin>>a>>b>>c;`
 - `cout<<a;`
 - `cout<<a<<b<<c;`
- Caso contrário não seria possível realizar chamadas “em cascata”.



NumeroTelefone.cpp

UFOP

```
#include <iomanip>
using namespace std;
#include "NumeroTelefone.h"

ostream &operator<<( ostream &output, const NumeroTelefone &numero )
{
    output << "(" << numero.DDD << ")"
        << numero.inicio << "-" << numero.fim;
    return output; // permite cout << a << b << c;
}

istream &operator>>( istream &input, NumeroTelefone &numero )
{
    input.ignore(); // pula (
    input >> setw( 2 ) >> numero.DDD; // entrada do código de área
    input.ignore( 2 ); // pula ) e espaço
    input >> setw( 4 ) >> numero.inicio; // primeira parte
    input.ignore(); // pula traço (-)
    input >> setw( 4 ) >> numero.fim; // segunda parte
    return input; // permite cin >> a >> b >> c;
}
```



driverNumeroTelefone.cpp

UFOP

```
#include <iostream>
using namespace std;
#include "NumeroTelefone.h"

int main()
{
    NumeroTelefone telefone; // cria objeto telefone

    cout << "Digite o numero no formato (12) 3456-7890:" << endl;

    // cin >> telefone invoca operator>> emitindo implicitamente
    // a chamada da função global operator>>( cin, telefone )
    cin >> telefone;

    cout << "O numero informado foi: ";

    // cout << telefone invoca operator<< emitindo implicitamente
    // chamada da função global operator<<( cout, telefone )
    cout << telefone << endl;
    return 0;
}
```



Sobrecarga de Operadores

UFOP

- Quando a chamada “`cin >> telefone;`” é executada, na verdade chamamos “`operator>>(cin, telefone);`”;
- Nas funções de sobrecarga, `cin` e `cout` recebem outro nome, `input` e `output`, respectivamente;
- Note que na sobrecarga do operador `<<` o objeto da classe `NumeroTelefone` é recebido como uma constante
 - Menor privilégio: se o método não alterará o objeto, então este deve ser uma constante;
 - Um programa só pode chamar os métodos `const` de um objeto constante.
- Note ainda que o programa pressupõe que o usuário digitará corretamente os dados
 - Como alterar isto?



Sobrecarga de Operadores

UFOP

- Vamos alterar nosso exemplo para sobrecarregar o operador *
 - Ele será utilizado para atribuição
 - Não sobrecregaremos o operador = porque este já é sobrecregido.
 - Será implementado como um método;
- Este exemplo também ilustra uma sobrecrecarga que não retorna nada
 - Desta forma não será possível realizar chamadas do tipo

a * b* c;



Telefone.h

UFOP

```
#include <iostream>
#include <string>
using namespace std;

class NumeroTelefone
{
    friend ostream &operator<<( ostream &, const NumeroTelefone & );
    friend istream &operator>>( istream &, NumeroTelefone & );
public:
    void operator*(NumeroTelefone ); // sobrecarga do operador *
private:
    string DDD;
    string inicio;
    string fim;
};
```



Telefone.cpp

UFOP

```
ostream &operator<<( ostream &output, const NumeroTelefone &numero )
{
    output << "(" << numero.DDD << ")" " << numero.inicio << "-" <<
    numero.fim;
    return output;
}
```

```
istream &operator>>( istream &input, NumeroTelefone &numero )
{
    input.ignore();
    input >> setw( 2 ) >> numero.DDD;
    input.ignore( 2 );
    input >> setw( 4 ) >> numero.inicio;
    input.ignore();
    input >> setw( 4 ) >> numero.fim;
    return input;
}
```

```
void NumeroTelefone::operator*(NumeroTelefone ladoDireito)
{
    DDD = ladoDireito.DDD; // Realiza a cópia dos atributos
    inicio = ladoDireito.inicio;
    fim = ladoDireito.fim;
}
```



DriverTelefone.cpp

UFOP

```
#include <iostream>
using namespace std;
#include "NumeroTelefone.h"

int main()
{
    NumeroTelefone telefone, celular; // cria objeto telefone

    cout << "Digite o numero no formato (12) 3456-7890:" << endl;
    cin >> telefone;
    cout << "O numero informado foi: ";

    celular * telefone; // chamada do método celular.*(telefone);

    cout << celular << endl;
    return 0;
}
```



Exercício

UFOP

- Como sobrecregamos o operador ++?
 - Prefixado ou pós-fixado?
 - Como diferenciar um do outro?



Ponteiro *this*



Ponteiro *this*

- Como um método sabe quais são os atributos de um objeto que o chamou?
 - Quando um objeto chama um método, implicitamente é passado o ponteiro ***this***
 - Aponta para os atributos do objeto.
 - O ponteiro é utilizado implicitamente pelos métodos, como fizemos até aqui;
 - O ponteiro não faz parte do objeto.
- O uso do ponteiro é opcional, exceto em casos em que se faz necessário a chamada de métodos em cascata.



Ponteiro *this*

UFOP

```
#include <iostream>
using namespace std;

class Test
{
public:
    Test( int = 0 ); // construtor padrão
    void print() const;
private:
    int x;
};
```



Ponteiro *this*

UFOP

```
// construtor
Test::Test( int value ):x( value ) // inicializa x como value
{
    // corpo vazio
}

// imprime x utilizando ponteiros this implícito e explícito;
void Test::print() const
{
    // utiliza implicitamente o ponteiro this para acessar o membro x
    cout << "      x = " << x;

    // utiliza explicitamente o ponteiro this e o operador seta
    // para acessar o membro x
    cout << "\n  this->x = " << this->x;
}
```

Comentários Finais

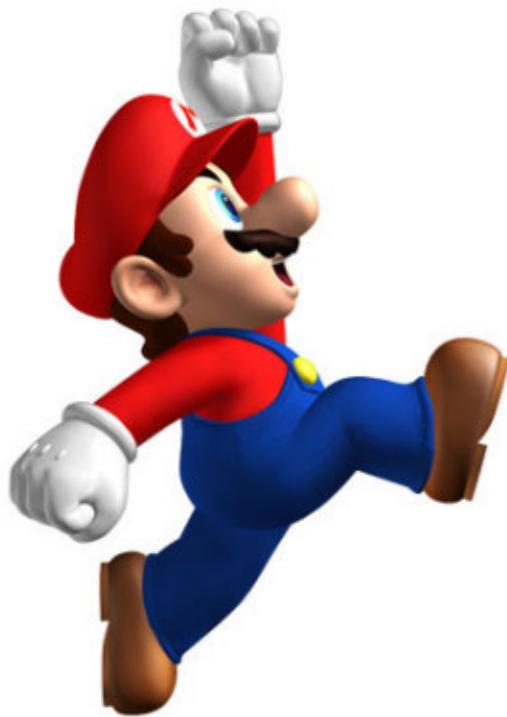


Comentários Finais

- No site da disciplina está disponível um exemplo maior de sobrecarga de operadores
 - Classe ***String*** (não é a da biblioteca padrão C++);
 - Sobrecarrega os operadores `>>`, `<<`, `=`, `+=`, `!`, `==`, `<`, `[]` e `()`;
 - Utiliza **construtor cópia** e o ponteiro ***this***.
- Há também um *driver* da classe `string` da biblioteca padrão C++
 - Mostra a utilização de algumas das funcionalidades;
 - Mais funcionalidades no `c++ reference` ou na bibliografia.



UFOP



Perguntas?



Na próxima aula

UFOP

- Herança
 - Compilação
 - Redefinicao de Metodos
 - Construtores e Destrutores em Classes Derivadas
- Herança Pública vs. Privada vs. Protegida
- Conversões de Tipo entre Base e Derivada
- Herança Múltipla
 - Compilação
 - Construtores em Herança Múltipla



UFOP



FIM