Tipos Abstratos de Dados -TADs

Joaquim Quinteiro Uchôa,
Juliana Galvani Greghi,
Renato Ramos da Silva

- TADs e POO
- Conceitos Básicos de POO
- Visibilidade e Encapsulamento
- Implementando Classes
- Construtores e Destrutores / Sobrecarga de Métodos
- Atributos Alocados Dinamicamente
- Sobrecarga de Operadores
- Projeto de Classes

TADs e POO

Tipos Abstratos de Dados - I

Representação abstrata de dados a serem utilizados com foco nas operações a serem realizadas, sem se preocupar com detalhes de implementação.

Tipo de dado independente de sua representação.

Tipos Abstratos de Dados - II

Pensemos no exemplo de representar um estudante. Sem abstração, estudantes seriam representados por variáveis soltas (nome, idade, matrícula, etc.):

```
string nome;
int idade;
int matricula;
```

Para o desenvolvedor, podem ser encaradas como variáveis independentes, sem ligação direta.

Tipos Abstratos de Dados - III

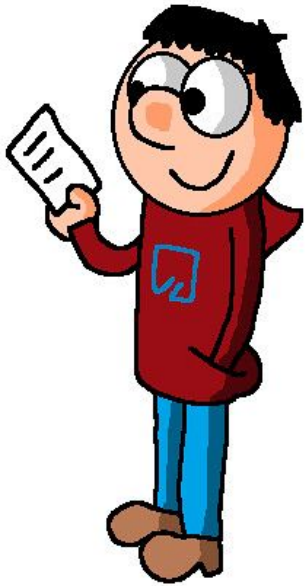
Usando a visão de tipos abstrados de dados, um algoritmo é projetado tendo-se como base que não há o nome, idade e matrícula do estudante, mas simplesmente o tipo estudante.

Este tipo, como um tipo simples (inteiro ou string), deve ter operadores próprios. Assim, TAD estudante deve possuir operações desejáveis ao programador, como validar a matrícula, verificar idade, etc.

Tipos Abstratos de Dados - IV

Uma representação em estilo de estruturas já ajuda melhor a visualizar um TAD:

```
struct estudante {
   string nome;
   int idade;
   int matricula;
};
```



Tipos Abstratos de Dados - V

Entretanto... como as operações em registros são implementadas geralmente como funções separadas dos registros, não há como garantir que as operações garantam a interdependência dos campos.

Assim, uma melhor abordagem é o uso de classes, um dos pilares da orientação a objetos.

Programação Orientada a Objetos

Conjunto de princípios que orientam a criação de sistemas computacionais abordando os problemas e suas soluções como objetos que interagem entre si.

Paradigma Orientado a Objetos

A programação orientada a objetos (POO) geralmente implica em mudança de visão para a programação. Para parte dos pesquisadores é apenas uma evolução da programação estruturada.

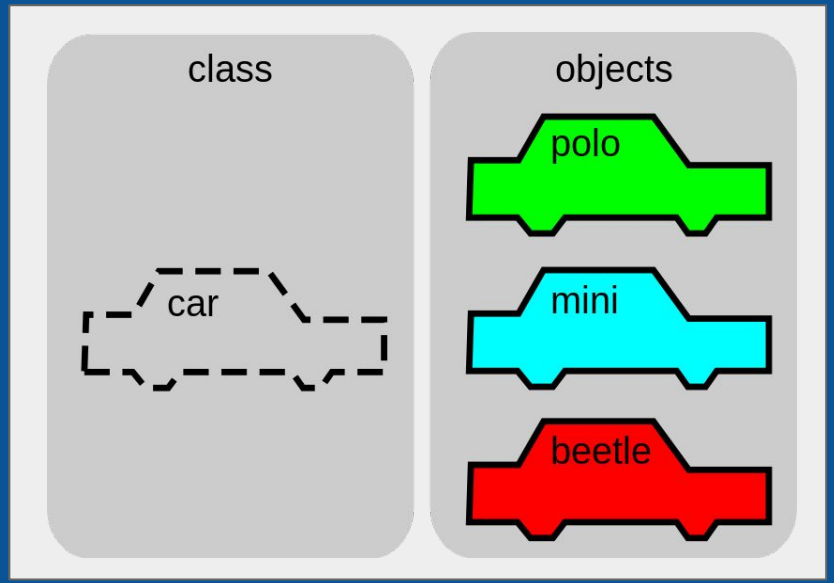
Entretanto, a maior parte dos pesquisadores entendem esse tipo de programação como um paradigma, uma vez que implica em formas diferenciadas de desenvolvimento de software.

Suporte a POO em Linguagens

POO é suportada por várias linguagens (ex: C++, Python, Ruby, C#, atualmente sua maior expressão comercial é dada pelo Java).

Em nosso curso, usaremos técnicas básicas de POO em C++.

Conceitos Básicos de POO



Pilares do Paradigma Orientado a Objetos

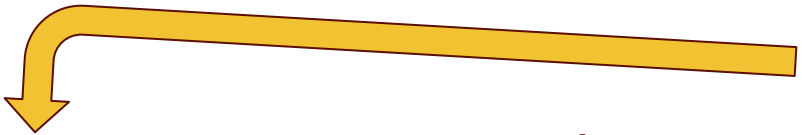
Existem quatro conceitos fundamentais (**pilares**) que norteiam o desenvolvimento POO:

- Abstração;
- Encapsulamento;
- Herança;
- Polimorfismo.

Pilares do Paradigma Orientado a Objetos

Existem quatro conceitos fundamentais (**pilares**) que norteiam o desenvolvimento POO:

- **Abstração;**
- **Encapsulamento;**
- Herança;
- Polimorfismo.



Neste curso só iremos abordar, e apenas os elementos básicos, dos dois primeiros pilares. O aprofundamento e os outros dois pilares serão vistos em curso específico.

Abstração

Representação de uma entidade do mundo real, com seu comportamento e características:

- Objetos;
- Classes;
- Atributos;
- Métodos.

Objetos

- Um objeto pode ser entendido como um ser, lugar, evento, coisa ou conceito do mundo real ou imaginário que possa ser aplicável a um sistema.
- É comum que haja objetos diferentes com características semelhantes. Esses objetos são agrupados em *classes*.

Classes

Uma classe é um tipo abstrato de dados, que reúne objetos com características similares.

O comportamento destes objetos é descrito pelo conjunto de métodos disponíveis e o conjunto de atributos da classe descrevem as características de um objeto.

Exemplo: Uma Classe Estudante

```
class estudante {
  private:
    string nome;
    int idade;
    int matricula;
    string disciplinas[10];
  public:
    void inicializarDados(string n, int i, int m);
    void imprimirDados();
    bool matricularEmDisciplina(string disciplina);
};
```

Atributos

- Um atributo é uma característica de um grupo de entidades do mundo real, agrupados em uma classe.
- Um atributo pode ser um valor simples (um inteiro, por exemplo) ou estruturas complexas (um outro objeto, por exemplo).

Exemplo

Considere a classe Foo. Seus atributos são um número inteiro n, um caractere c, um número real f, uma cadeia de caracteres (string) s e um ponteiro para o tipo inteiro vet.

```
class Foo {
  private:
    int n;
    char c;
    float f;
    string s;
    int* vet;
  public:
    bool inicializa();
    bool finaliza();
    float calcula(int x);
    void imprime();
};
```

Métodos

- Semelhante a uma função, é a implementação de uma ação da entidade representada pela classe;
- Conjunto de métodos define o comportamento dos objetos de uma classe.

Exemplo

Considere a classe Foo.

Seus métodos são:

inicializa(),

finaliza(),

calcula()

e imprime().

```
class Foo {
  private:
    int n;
    char c;
    float f;
    string s;
    int* vet;
  public:
    bool inicializa();
    bool finaliza();
    float calcula(int x);
    void imprime();
};
```

Exemplo: Uma Classe Estudante

```
class estudante {
  private:
    string nome;
    int idade;
    int matricula;
    string disciplinas[10];
  public:
    void inicializarDados(string n, int i, int m);
    void imprimirDados();
    bool matricularEmDisciplina(string disciplina);
};
```

Atributos

Métodos

E o tal do Objeto?

- *Classes são um agrupamento de objetos com características similares!*
- *Objetos são entidades (instâncias) únicas de uma classe!*

Em C++: classes definem tipos, objetos são instanciados em variáveis.

Exemplo - Objetos da Classe Estudante

```
int main() {
    estudante aluno1;
    estudante sala[20];
    ...
    ...
}
```

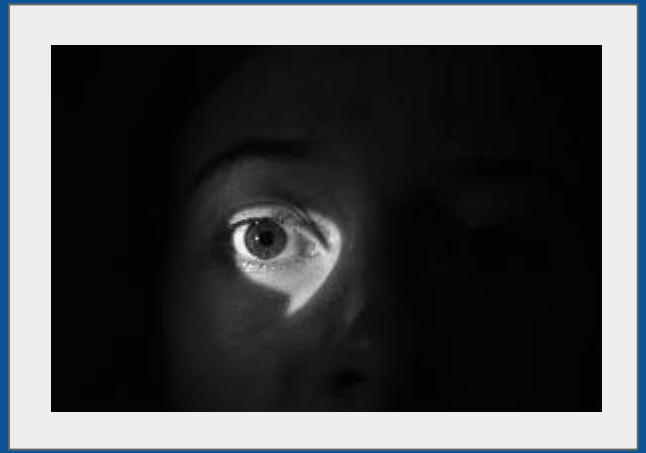
Exemplo - Objetos da Classe Estudante

```
int main() {
    estudante aluno1;
    estudante sala[20];
    ...
    ...
}
```

Um único objeto da classe estudante

Um vetor de objetos da classe estudante

Visibilidade e Encapsulamento



Visibilidade

- A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;
- Três modos distintos:
  - Público;
  - Privado;
  - Protegido.

Visibilidade

- A visibilidade é utili
  acesso de um determina
- Três modos distin
  - Público
  - Privado
  - Protegido

objetos de quaisquer classes podem ter acesso a atributos ou métodos públicos.

Visibilidade

- A visibilidade é utiliza
  acesso de um determinad
- Três modos distintos:
  - Público
  - Privado
  - Protegido

apenas a classe que define atributos ou métodos privados pode ter acesso a eles.

Visibilidade

- A visibilidade é utilizada para indicar o nível de acesso de um determinad
- Três modos distintos:
  - Público
  - Privado
  - Protegido

apenas a classe que define atributos ou métodos protegidos, ou classes filhas, podem ter acesso a eles.

Visibilidade

- A visibilidade é utilizada para indicar o nível de acesso de um determin
- Três modos distintos:
  - Público
  - Privado
  - Protegido

apenas a classe que define atributos ou métodos protegidos, ou classes filhas, podem ter acesso a eles.

Sem uso de herança, o modo protegido funciona de maneira similar ao privado.

Exemplo de Visibilidade

Na classe Foo, os atributos estão privados, enquanto os métodos estão todos públicos.

Isso implica que os atributos só podem ser acessados por métodos da classe.

```
class Foo {
  private:
    int n;
    char c;
    float f;
    string s;
    int* vet;
  public:
    bool inicializa();
    bool finaliza();
    float calcula(int x);
    void imprime();
};
```

Encapsulamento

- Ocultação de dados;
- Possibilita o melhor aproveitamento dos componentes do software, facilitando:
  - Entendimento
  - Reuso
  - Manutenção

Encapsulamento

- Minimiza a interdependência;
- Não se conhece seu funcionamento internamente, apenas se sabe como utilizar;
- A classe define as operações/atributos acessíveis a outras classes.
- Possui como vantagens:
  - Segurança;
  - Independência;
  - Elevado grau de abstração.
- Impactado pela amizade entre classes (friend).

Classes e Funções "Amigas"

Em alguns casos, pode ser interessante permitir que outras classes ou funções externas possam acessar os dados privados, sem modificar o encapsulamento.

Em C++, isso é feito por meio da diretiva friend.

Exemplo de Função Amiga

O procedimento dobra foi declarado como amigo da classe Foo.

Nesse caso, esse procedimento consegue acessar e alterar atributos privados de Foo.

```
class Foo {
  friend void dobra(Foo f);
  private:
    int n;
    char c;
    float f;
    string s;
  ...
  ...
};

void dobra(Foo f){
  f.n = 2* f.n;
}
```

Implementando Classes

Implementação dos Métodos

A classe Foo encontra-se declarada, entretanto é necessário implementar seus métodos.

A implementação dos métodos de uma classe pode ser feita interna ou externamente à classe.

```
class Foo {
  private:
    int n;
    char c;
    float f;
    string s;
    int* vet;
  public:
    bool inicializa();
    bool finaliza();
    float calcula(int x);
    void imprime();
};
```

Formas de Implementação dos Métodos da Classe

Os métodos em uma classe podem ser implementados:

- diretamente na classe, logo após a declaração, o que é chamado de implementação inline;
- externamente, utilizando o operador de escopo (::), para indicar que o método pertence à classe.

Implementação Interna

```
class Foo {
 ...
public:
    bool inicializa() {
      n = 0;
      c = 'a';
      f = 0;
      s = "a";
      vet = new int[10];
      return true;
    }
    ...
```

```
...
    bool finaliza() {
      delete[] vet;
      return true;
    }

    float calcula(int x) {
      return x * f;
    }

    void imprime() {
      cout << n << " "
           << c << " "
           << f << " "
           << s << endl;
    }
};
```

Implementação Externa

```
class Foo {
  ...
};
```

```
bool Foo::inicializa() {
  n = 0;
  c = 'a';
  f = 0;
  s = "a";
  vet = new int[10];
  return true;
}
```

```
bool Foo::finaliza() {
  delete[] vet;
  return true;
}

float Foo::calcula(int x) {
      return x * f;
}

void Foo::imprime() {
  cout << n << " "
       << c << " "
       << f << " "
       << s << endl;
}
```

Funções Inline - I

Uma função inline é uma **sugestão** para o compilador fazer uma cópia de seu código ao invés da chamada de função. Isso gera um pequeno aumento de eficiência.

Geralmente usa-se inline com funções pequenas e que são chamadas com frequência.

O compilador pode aceitar ou não a sugestão, dependendo, inclusive, de opções de otimização.

Funções Inline - II

Funções ou procedimentos que são implementados dentro da declaração da classe são compiladas como inline.

Outra alternativa é usar a diretiva inline antes da declaração da função.

Função Inline - Exemplo

```
class Foo {
 ...
    bool finaliza() {
      delete[] vet;
      return true;
    }
    ...
    inline float calcula(int x);
 ...
};
```

```
float Foo::calcula(int x) {
      return x * f;
}
```

Nesse caso, finaliza() e calcula() são funções inline.

Implementação Interna ou Externa? - I

Uma diferença prática entre implementação interna ou externa é que a interna é automaticamente inline.

É importante lembrar que inline é apenas uma sugestão ao compilador e pode não ser acatada. Além disso pode-se usar a diretiva inline com implementação externa.

Implementação Interna ou Externa? - II

A implementação externa permite que o código seja separado em arquivos diferentes: um arquivo .h ou .hpp com a declaração (o cabeçalho) e um arquivo .cpp com a implementação dos métodos.

Nesse caso, pode-se disponibilizar a implementação já compilada em forma de código-objeto, sem necessidade de disponibilizar o código da implementação.

Implementação Interna ou Externa? - III

Tirando-se os detalhes da declaração inline automática, ou a possibilidade de disponibilização de bibliotecas sem o código-fonte da implementação, na grande maioria dos casos a opção por implementação interna ou externa é uma questão de estilo.

É importante que o aluno tenha contato com as duas formas de desenvolvimento.

Um exemplo passo a passo: vectorx

Vetores tradicionais (arrays) não possuem controle de tamanho, o que pode gerar erros de acesso a posições indevidas. Também não possuem recurso para impressão.

Um TAD específico poderia acrescentar essas e outras funcionalidades.

Vamos a um passo a passo de construção desse TAD, inicialmente usando registros.

VectorX: Implementação utilizando registros puros

Arquivo vecstruct1.cpp

Poucas modificações seriam necessários nesse código para funcionamento em C puro.

É um código de difícil manutenção, sendo complicado verificar erros e adicionar novos recursos.

VectorX: Implementação utilizando registros de C++

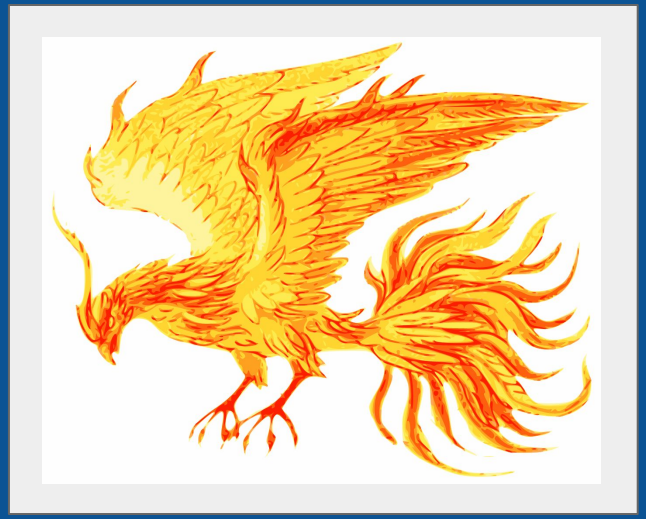
Arquivo vecstruct2.cpp

Em C++ é possível usar métodos dentro de estruturas. O código é mais claro e fácil de manter.

Em C++ estruturas (structs) são apenas classes em que tudo é público por padrão. Assim, falta encapsulamento.

O exemplo também pode ser melhorado usando construtores e destrutores.

Construtores e Destrutores / Sobrecarga de Métodos



Construtores

Um construtor é um método especial para a criação e inicialização de uma nova instância de uma classe.

Um construtor inicializa um objeto e suas variáveis, cria quaisquer outros objetos de que ele precise, garantindo que ele seja configurado corretamente quando criado.

Na maioria das linguagens de programação, o construtor é um método que tem o mesmo nome da classe, que geralmente é chamado quando um objeto da classe é declarado ou instanciado.

Destrutores

De forma similar aos construtores, os destrutores são métodos fundamentais das classes, sendo geralmente chamados quando termina o tempo de vida do objeto.

Em algumas linguagens, como C++, ocupam papel tão importante quanto os construtores, por conta da necessidade de desalocação de memória.

Construtor/Destrutor

Em C++, o construtor tem o próprio nome da classe sem definição de tipo (já que o tipo é a própria classe).

O destrutor, em C++, é indicado pelo nome da classe, precedido de do til (~), também sem definição de tipo.

```
class Foo {
 ...
public:
    Foo() {
      n = 0;
      c = 'a';
      f = 0;
      s = "a";
      vet = new int[10];
    }
    ~Foo() {
      delete[] vet;
    }
 ...
};
```

Inicialização de atributos em construtores - I

Atributos em classes são inicializados geralmente de duas maneiras:

- inicialização direta
- inicialização em lista (repasse ao construtor do atributo)

Inicialização de atributos em construtores - II

```
class myClass
{
  public:
    myClass(float f, char a,
            bool b, int i);
  private:
    float mFloat;
    char mCharacter;
    bool mBoolean;
    int mInteger;
};
```

```
myClass::myClass(float f, char a,
    bool b, int i) :
       // inicialização em lista
       mFloat( f ), mBoolean( true )
{
    // inicialização direta
    mCharacter = a;
    mInteger = 0;
}
```

Inicialização de atributos em construtores - III

Apesar de a inicialização direta ser mais comum entre os desenvolvedores, pode ser mais eficiente utilizar inicialização em lista (depende do compilador). Ainda:

- atributos constantes ou referências só podem ser inicializados em lista;
- vetores ou estruturas sem construtores só podem ser inicializados diretamente.

Sobrecarga de Métodos - I

Em C++, é possível ter vários métodos diferentes com o mesmo nome, desde que os parâmetros utilizados sejam diferentes:

```
int foo(int x);
float foo(float f);
int foo(char c);
float foo(float f, char c, int b);
```

Sobrecarga de Métodos - II

Note que não é possível utilizar essa abordagem com o mesmo conjunto de parâmetros, o seguinte trecho não funcionaria:

```
int   foo(int x, float f);
float foo(int x, float f);
```

Sobrecarga de Métodos - III

Como a ordem dos parâmetros é importante , o seguinte trecho funcionaria sem problemas:

```
int   foo(float f, int x);
float foo(int x, float f);
```

Sobrecarga de Métodos - IV

Construtores também podem ser sobrecarregados.

Assim, é possível ter inicialização de objetos de maneiras diferenciadas, de acordo com os parâmetros utilizados.

```
class Foo {
 ...
    Foo();
    Foo(int n);
    Foo(int n, float f);
 ...
};
```

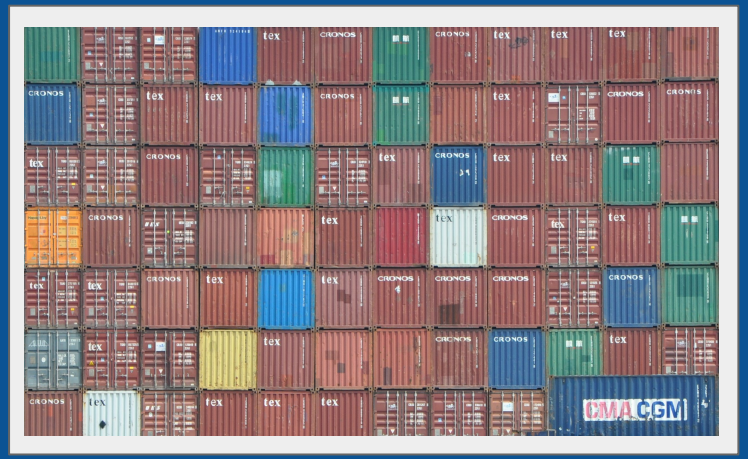
VectorX: Implementação utilizando classes em C++

Arquivo vecclass1.cpp

Apesar de estruturas serem classes, a boa prática de programação recomenda utilizar class para classes normais e struct apenas em situações que o encapsulamento não é necessário.

Esse código usa construtores, com aumento de manutenibilidade.

Atributos Alocados Dinamicamente



Atributos alocados dinamicamente

Quando são utilizados atributos alocados dinamicamente, é necessário um cuidado adicional com a forma que os objetos são utilizados.

Pode ser necessário implementar um **construtor de cópia** e sobrecarregar o **operador de atribuição** para evitar maiores problemas.

Operações implementadas por padrão

Mesmo quando não se implementa, são disponibilizados em C++:

- **Construtor padrão (sem parâmetros)** - apenas aloca espaço na memória para valores estáticos
- **Construtor de cópia** - faz cópia do objeto, copiando valores estáticos
- **Operador de atribuição** - copia os valores do objeto à direita para o objeto à esquerda

E O QUE TEM ISSO COM PONTEIROS?

Ponteiros são variáveis estáticas (contém um endereço de memória), mas que são geralmente usados para apontar variáveis alocadas dinamicamente.

Ou seja, o endereço de memória (o valor do ponteiro) é copiado…

A área apontada por ele, não…

E se a cópia faz a desalocação dos dados???

Um exemplo comum

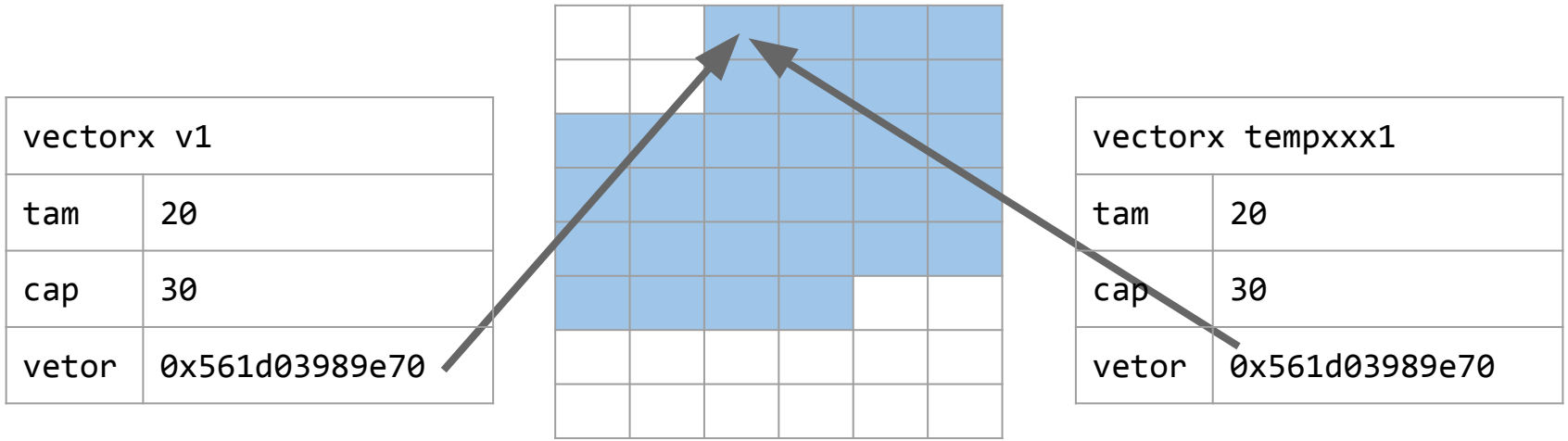
vectorx v1	
tam	20
cap	30
vetor	0x561d03989e70

Chamada por valor (cópia)

```
foo(vectorx v)
```

vectorx tempxxx1	
tam	20
cap	30
vetor	0x561d03989e70

Ou seja



As duas variáveis possuem atributos apontando a mesma região da memória!

Mas...

A região da memória apontada por temp é desalocada automaticamente quando a variável encerra seu tempo de vida...

```
vectorx::~vectorx() {
    cout << "morri" << endl;
    delete[] vetor;
    tam = 0;
}
```

Mas...

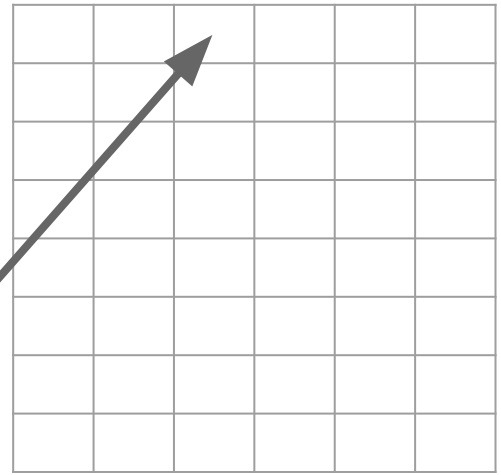
A região memória apontada por vec é desalocada automaticamente quando a variável encerra seu tempo de vida...

```
vectorx::~vectorx() {
    cout << "morri" << endl;
    delete[] vetor;
    tam = 0;
}
```

O que vai acontecer quando a função foo( ) terminar???

Muita caca!

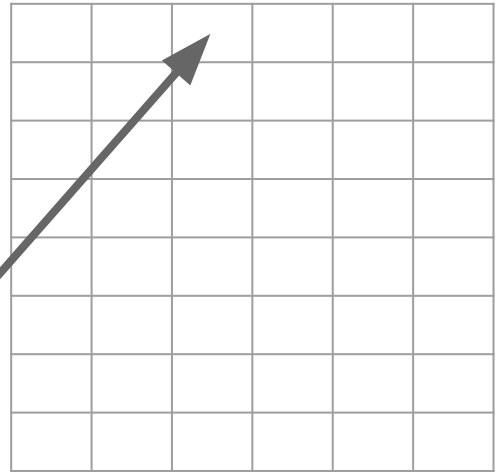
vectorx v1	
tam	20
cap	30
vetor	0x561d03989e70



v1 possui atributo apontando para uma região que foi desalocada

Muita caca mesmo!

vectorx v1	
tam	20
cap	30
vetor	0x561d03989e70



v1 possui atributo apontando para uma região que foi desalocada

Tentativa de acessar os dados irá gerar falha de segmentação!

Atribuição irá gerar problemas similares

```
vectorx  v1;

// construtor de cópia
foo(v1);

// construtor de cópia
vectorx v2 = v1;
```

```
vectorx v2, v3, v4;

// atribuição -> operador =
vectorx v5;
v5 = v2;

// atribuição -> operador =
v4 = v2 + v3;
```

Caminhos possíveis

Caminho 1	Caminho 2
Não utilizar cópias, usando apenas ponteiros e referências para passagem de parâmetros em funções.	Implementar construtor de cópia (sobrescrevendo o construtor de cópia padrão).
Não efetuar atribuições.	Implementar sobrecarga do operador de atribuição (=).

Construtor de cópia

Um construtor de cópia recebe um outro objeto para efetuar cópia dos dados.

```
vectorx::vectorx(const vectorx& vec) {
  cout << "construtor de cópia" << endl;
  tam = vec.tam;
  vetor = new int[tam];
  for (int i = 0; i < tam; i++) {
     vetor[i] = vec.vetor[i];
   }
}
```

Construtor de cópia

Um construtor de cópia recebe um outro objeto para efetuar cópia dos dados.

```
vectorx::vectorx(const vectorx& vec) {
  cout << "construtor de cópia" << endl;
  tam = vec.tam;
  vetor = new int[tam];
  for (int i = 0; i < tam; i++) {
    vetor[i] = vec.vetor[i];
  }
}
```

Dados dinâmicos são alocados e copiados manualmente.

Construtor de cópia

Um construtor de cópia recebe um outro objeto para efetuar cópia dos dados.

```
#include <cstring> // para uso da memcpy
...
vectorx::vectorx(const vectorx& vec) {
  cout << "construtor de cópia" << endl;
  tam = vec.tam;
  vetor = new int[tam];
  memcpy(vetor, vec.vetor, tam);
}
```

Cópia dos dados feita de maneira mais eficiente

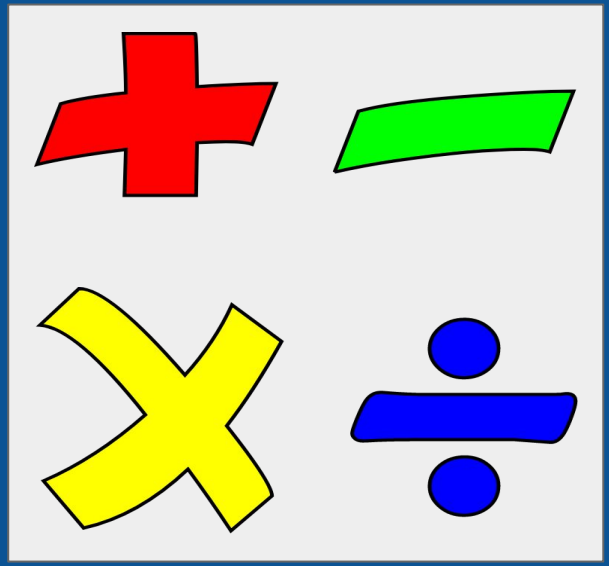
Construtor de cópia - observações importantes

Parâmetro recebido como referência (para evitar cópia, chamada recursiva).

Recomenda-se o uso de const, para evitar alterar os dados do objeto recebido como parâmetro.

```
vectorx::vectorx(const vectorx& vec)
```

Sobrecarga de Operadores



Atribuição com atributos alocados dinamicamente

Assim como com o construtor, vimos que atributos alocados dinamicamente podem gerar problemas ao se realizar uma atribuição.

Um dos caminhos possíveis para resolver esse problema é sobrecarregar o operador de atribuição.

Sobrecarga de Operadores - Visão Geral

Mecanismo que permite modificar operadores dependendo dos argumentos fornecidos. Possui vantagens e desvantagens:

- maior facilidade de escrita
- risco de menor legibilidade

```
int main() {

    Ponto p1;
    Ponto p2;
    Ponto p3 = p1 + p2;

}
```

Sobrecarga de Operadores - Exemplo

```
public class Ponto {
private:
   int x;
   int y;
public:
   Ponto operator+(const Ponto &outro){
      return new Ponto(this->x + outro.x, this->y + outro.y);
   }
   ...
}
```

Sobrecarga de Operadores - Exemplo

```
public class Ponto {
private:
   int x;
   int y;
public:
   Ponto operator+(const Ponto &outro){
      return new Ponto(this->x + outro.x, this->y + outro.y);
   }
   ...
}
```

O Operador This - I

O operador this, em C++, é um ponteiro para o objeto atual. Em várias situações ele é desnecessário, mas pode aumentar a clareza. Os dois trechos a seguir são equivalentes:

```
   Ponto operator+(const Ponto &outro){
      return new Ponto(this->x + outro.x, this->y + outro.y);
   }
   Ponto operator+(const Ponto &outro){
      return new Ponto(x + outro.x, y + outro.y);
   }
```

O Operador This - II

Outro motivo para usar o this é quando o parâmetro do método e o atributo da classe possuem o mesmo nome:

```
   Noh::alteraValor(int valor){
      this->valor = valor;
   }
```

Ressalta-se que alguns desenvolvedores gostam de utilizar this mesmo quando não há problemas de nomenclatura, com a justificativa de aumento de clareza. Por conta de uso de dois objetos de mesma classe, é bastante usado em sobrecarga de operadores.

Sobrecarga de Operadores - I

Em C++ é possível sobrecarregar não apenas métodos, mas também operadores.

Nesse caso, os operadores podem ser entendidos como uma função com nome especial e que é chamada quando o operador é encontrado.

Sobrecarga de Operadores - II

Operadores podem ser geralmente implementados como funções externas (geralmente “amigas”) ou como métodos das classes. Na maioria das vezes, cabe ao desenvolvedor escolher a melhor forma de implementação.

Em alguns casos (por exemplo sobrecarga do operador “<<” para saída), é necessário implementar como função externa.

Sobrecarga de Operadores - III

O operador + na operação a+b, em que a e b sejam da classe Foo, pode ser entendido como:

```
operator+(Foo& a, Foo& b)
```

Função + aplicada a a e b.

Ou

```
Foo::operator+(Foo& b)
```

Método + de a aplicado a b.

Sobrecarga de Operadores - iv

Questões importantes na sobrecarga de operadores:

- Qual o tipo de retorno? Em geral, operadores possuem retorno para permitir continuidade. Ex: `x = a + b + c;` (essa operação não seria possível sem retorno).
- Qual o tipo de passagem de parâmetro (geralmente referência, com parâmetros constantes).

Cuidados com sobrecarga de operadores

Sobrecarga de operadores podem gerar problemas semânticos e devem ser usados com cautela. Se mal utilizados, podem gerar problemas de legibilidade de código.

Por exemplo, dado dois vetores a e b, o que representa a+b? Concatenação dos vetores ou soma dos elementos um a um? Note que as duas operações existem matematicamente e poderiam usar o mesmo símbolo.

Cuidados com operador *

É recomendável o uso de () para evitar confusões entre o operador de indireção e a multiplicação:

```
int p = 10;
int q = 20;
int *ptr1 = &p;
int *ptr2 = &q;
```

```
cout << *ptr1 << endl;
cout << *ptr2 << endl;
// trecho com erro!
cout << *ptr1 *ptr2 << endl;
// trecho sem erros
cout << (*ptr1) * (*ptr2)
     << endl;
```

Exemplo de sobrecarga de operadores - I

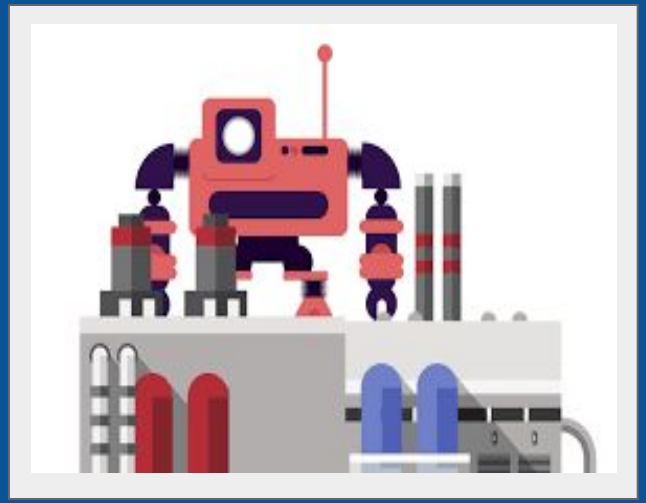
```
class vectorx {

    vectorx& operator+(const vectorx& vec);
    vectorx& operator=(const vectorx& vec);
    int operator*(const vectorx& vec); // produto interno
    vectorx operator*(const int& n); // produto de vetor por n
    friend vectorx operator*(const int& n, vectorx& vec);
    friend ostream& operator<<(ostream& out, const vectorx& vec);
```

Exemplo de sobrecarga de operadores - II

Arquivo vecclass2.cpp

Projeto de Classes



Projetando Classes - I

Classes são utilizadas para representar tipos de dados. Esses dados, por sua vez, são usados para resolver problemas.

Em geral, esses problemas podem ser resolvidos utilizando diferentes conjuntos de classes/objetos.

Além disso, não existe uma única forma de representar uma classe.

Projetando Classes - I

Classes são utilizadas para representar tipos de dados. Es s para resolver

Em geral idos utilizan /objetos.

Existem, entretanto, boas práticas, que facilitam ampliar e utilizar classes/objetos, facilitando manutenção de código.

Além disso, não existe uma única forma de representar uma classe.

Projetando Classes - II

Uma questão que sempre figura ao projetar uma classe é: quais atributos e métodos uma classe deve ter.

Não é incomum, por exemplo verificar como atributos, valores que deveriam ser apenas retornos de métodos.

Uma recomendação inicial é: utilizar o mínimo necessário de atributos para representar a classe, agregando novos elementos apenas quando esses facilitarem o uso e desenvolvimento.

Projetando Classes - Um Exemplo - 1

Suponha a classe VectorX, criada para agregar a possibilidade de checagem de tamanho. Os atributos sugeridos para essa classe foram um vetor dinâmico e um atributo para armazenar o tamanho dos dados:

```
int* vetor;
int tam;
```

Projetando Classes - Um Exemplo - II

Suponha que seja interessante adicionar a possibilidade de redimensionar o vetor dinamicamente durante a execução.

Para não ficar redimensionando o vetor a todo momento, pode se adotar a estratégia de redimensioná-lo de tempos em tempos, adicionando um determinado tamanho a cada passo. Assim, pode ser que num determinado momento exista um espaço alocado para 100 elementos, mas usando apenas 20.

Projetando Classes - Um Exemplo - III

Para adicionar a capacidade de redimensionamento, a maneira mais adequada é adicionar um novo atributo na classe:

```
int* vetor;
int tam;
int capacidade;
```

Projetando Classes - Um Exemplo - iv

Suponha agora, que seja necessário calcular o maior e menor valor da classe VectorX. A pergunta geralmente feita é: deixar maior e menor como retorno de métodos ou como atributos na classe?

Projetando Classes - Um Exemplo - iv

Suponha agora, que seja necessário calcular o maior e menor valor da classe VectorX. A pergunta geralmente feita é: deixar maior e menor como retorno de métodos ou como atributos na classe?

Em geral, não há uma resposta única correta, a resposta depende do problema sendo resolvido.

Projetando Classes - Um Exemplo - iv

Suponha agora, que seja necessário calcular o maior e menor valor da classe VectorX. A pergunta geralmente feita é: deixar maior e menor como retorno de métodos ou como atributos na classe?

Em geral, não há uma resposta única correta, a resposta depende do problema sendo resolvido.

Qual é o problema sendo resolvido???

Projetando Classes - Um Exemplo - V

Caso o cálculo do maior ou menor elemento da classe VectorX seja para o uso mais corriqueiro e tradicional, é melhor implementá-los como retorno de métodos:

```
int maior( ) { … }
int menor( ) { … }
```

Observe que nesse caso, não há manutenção de dados, apenas o cálculo. Além disso, o método poderia retornar tanto o valor como a posição (e a escolha de qual depende novamente do problema sendo resolvido).

Projetando Classes - Um Exemplo - VI

Suponha, entretanto, que por algum motivo específico, o problema sendo resolvido precise ficar utilizando a cada momento o maior ou menor elemento da VectorX. Nesse caso, pode não ser razoável ficar calculando isso a cada momento e ser mais interessante armazenar maior (ou menor) como um atributo:

```
int maior;
void atualizaMaior( ); // método para atualizar o
                       // atributo maior
```

Projetando Classes - Um Exemplo - VII

Adicionar maior como atributo em VectorX não vem de graça, entretanto.

Observe a necessidade de criar um método para atualizá-lo. Mais importante que isso: é preciso garantir que o atributo seja sempre válido para uso.

Por exemplo ao inserir um novo elemento, é necessário verificar se ele não deve substituir o atualmente armazenado em maior.

Ou então: o que fazer se o elemento em maior for removido?

Atributos - Recomendações

Todo atributo tem um custo.

Inserir um atributo implica em avaliar como mantê-lo “saudável” para uso.

Atributos devem ser inseridos apenas quando são realmente necessários, ou irão facilitar o uso/desenvolvimento.

Escolha entre Protegido, Privado e Público - I

Recomenda-se o uso do princípio de menor privilégio possível ao usar POO. Ou seja: atributos e métodos que não precisem ser acessados externamente devem ser mantidos privados ou protegidos.

Na medida do possível, exceto pouquíssimas exceções, atributos devem ser mantidos privados ou protegidos. Um exemplo de exceção são atributos estáticos, que fogem ao escopo deste curso.

Escolha entre Protegido, Privado e Público - II

Métodos auxiliares, que só são usados por outros métodos da própria classe, também devem ser mantidos privados ou protegidos.

Ao se usar herança, métodos e atributos protegidos serão acessíveis por classes filhas, o que ocorre na maioria dos casos. Sem herança, não há distinção entre privado e protegido.

Escolha entre Protegido, Privado e Público - III

Usando-se o princípio de menor privilégio, uma recomendação é começar com atributos e métodos privados e ir movendo-os para a seção protegida ou pública, de acordo com a necessidade.

Deve-se evitar tornar público o acesso a atributos e métodos críticos da classe, que podem alterar sua estrutura sem uma devida verificação.

Dando Nome aos Bois - I

Existem vários padrões para nomear atributos e métodos em uma classe. Alguns padrões, por exemplo, definem que nomes de classes devem iniciar com letras maiúsculas.

Parte dos padrões especificam que os métodos devem ser verbos. Alguns desses padrões especificam verbos no infinitivo, enquanto outros especificam verbos no imperativo ou no presente do indicativo:

```
estudante::realizarMatricula()
estudante::realizeMatricula()
estudante::realizaMatricula()
```

Dando Nome aos Bois - II

No que diz respeito aos atributos, para evitar problemas de usar o mesmo nome em um atributo e um parâmetro, alguns padrões utilizam alguma letra no início de cada atributo, geralmente o m (de my, meu):

```
class Foo {
  int mValor;
  float mDado;
  ...
```

Dando Nome aos Bois - III

Não existe um padrão consensual para nomear métodos e atributos, assim duas recomendações são feitas:

1. Evitar misturar padrões diferentes em um mesmo código, evitando prejuízos à legibilidade.
2. Conhecer diferentes padrões, para evitar problemas ao ter que se integrar em uma equipe que já tenha um padrão pré-estabelecido.

Sobre o Material

Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Joaquim Quinteiro Uchôa
- Juliana Galvani Greghi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.