# Programação Orientada a Objetos

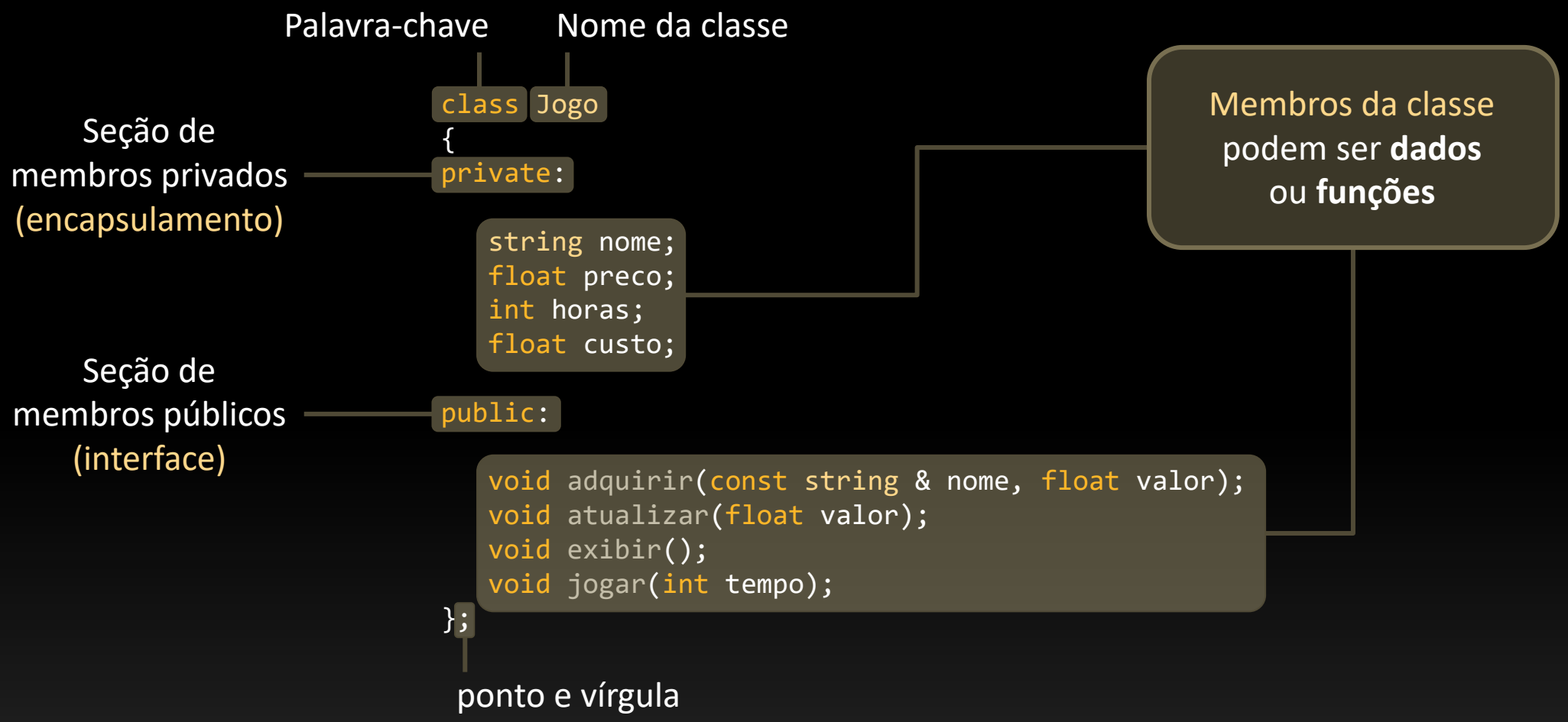
# Introdução

- A classe encapsula atributos
  - Os métodos funcionam como uma interface

```
class Jogo
{
private:
   string nome;
   float preco;
   int horas;
   float custo;

public:
   void adquirir(const string & titulo, float valor);
   void atualizar(float valor);
   void jogar(int tempo);
   void exibir();
};
```

# Introdução


Palavra-chave
Nome da classe
class Jogo
{
Seção de membros privados (encapsulamento)
private:
string nome;
float preco;
int horas;
float custo;
Membros da classe podem ser **dados** ou **funções**
Seção de membros públicos (interface)
public:
void adquirir(const string & nome, float valor);
void atualizar(float valor);
void exibir();
void jogar(int tempo);
};
ponto e vírgula

# Objetos e Instâncias

- Uma vez declarada a classe, podemos criar objetos

```
class Jogo
{
private:
   string nome;
   float preco;
   int horas;
   float custo;
   ...
};

int main()
{
   Jogo gears;
   Jogo doom;
}
```


Objeto / Instância da Classe Jogo (variável tipo Jogo)

Objeto / Instância da Classe Jogo (variável tipo Jogo)

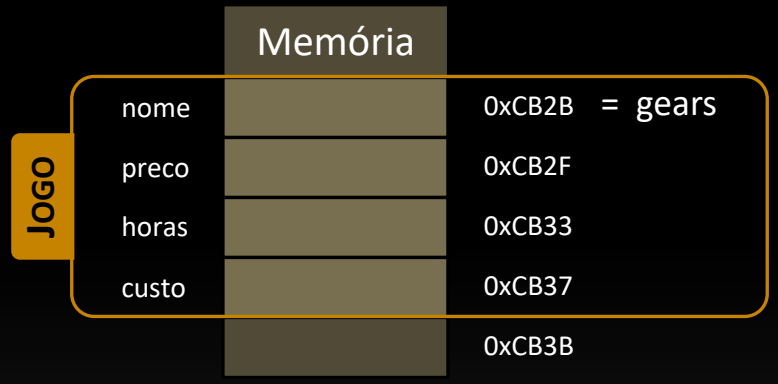
# Objetos e Instâncias

- **Memória é alocada** apenas com a criação de objetos
  - Processo também é chamado de instanciação
  - A declaração da classe não cria nada

```
class Jogo
{
private:
   string nome;
   float preco;
   int horas;
   float custo;
   ...
};
```

```
int main()
{
   Jogo gears ;
   ...
}
```

Objeto / Instância
da Classe Jogo
(variável tipo Jogo)



# Controle de Acesso

- As palavras-chave `public` e `private` fornecem controle de acesso aos membros da classe
  - Public: qualquer programa pode acessar diretamente os membros públicos através de um objeto da classe

```
int main()
{
   Jogo gears;
   gears.nome = "Gears of War";
   gears.preco = 100.0f;
   gears.horas = 340;
   gears.custo = gears.preco / gears.horas;
}
```

Equivalente

```
class Jogo
{
public:
   string nome;
   float preco;
   int horas;
   float custo;
   ...
};
```

```
struct Jogo
{
   string nome;
   float preco;
   int horas;
   float custo;
};
```

# Controle de Acesso

- As palavras-chave `public` e `private` fornecem controle de acesso aos membros da classe
  - Private: membros privados só podem ser acessados através dos métodos da classe

```
int main()
{
    Jogo gears;
✗   gears.nome = "Gears of War";
✗   gears.horas = 340;

✓   gears.adquirir("Gears of War");
✓   gears.jogar(340);
}
```

```
class Jogo
{
private:
    string nome;
    float preco;
    int horas;
    float custo;

public:
    void adquirir(const string & titulo);
    void atualizar(float valor);
    void jogar(int tempo);
    void exibir();
};
```

# Controle de Acesso

- Os membros privados ficam ocultos ao mundo exterior
  - Isso é chamado de *Data Hiding*
    - É uma boa prática de programação
    - Preserva a integridade dos dados

```
// apenas modifica as horas
gears.horas = 340;

// modifica horas e garante
// que o custo será atualizado
gears.jogar(340);
```

```
class Jogo
{
private:
    string nome;
    float preco;
    int horas;
    float custo;

public:
    void adquirir(const string & titulo);
    void atualizar(float valor);
    void jogar(int tempo);
    void exibir();
};
```

# Controle de Acesso

- Funções também podem ser ocultadas

```
class Jogo
{
private:
   string nome;
   float preco;
   int horas;
   float custo;

   void calcular()
   { if (horas > 0) custo = preco/horas; }

 public:
   void adquirir(const string & titulo, float valor);
   void atualizar(float valor);
   void jogar(int tempo);
   void exibir();
};
```

Tipicamente lidam com detalhes de implementação que não fazem parte da interface pública da classe.

# Controle de Acesso

- Os **métodos** podem acessar funções ou dados privados

```
// atualiza valor do jogo
void Jogo::atualizar(float valor)
{
    preco = valor;
    calcular();
}

// adiciona tempo de jogo
void Jogo::jogar(int tempo)
{
    horas = horas + tempo;
    calcular();
}
```

```
class Jogo
{
private:

   string nome;
   float preco;
   int horas;
   float custo;

   void calcular()
   {
      if (horas > 0)
         custo = preco/horas;
   }

   ...
};
```

# Encapsulamento

- O encapsulamento se manifesta através da:
  - Ocultação de dados
  - Ocultação de funções

- Mas em C++ a separação de código também é uma forma de encapsulamento:
  - Interface: arquivo .h
  - Implementação: arquivo .cpp

# Encapsulamento

- Benefícios do encapsulamento
  - Não é preciso conhecer os detalhes de implementação
    - O custo do jogo é armazenado?
    - Ou calculado na hora da exibição?

```
// exibe os dados de um jogo
void Jogo::exibir()
{
    cout << nome << " R$"
         << preco << " "
         << horas << "h = R$"
         << custo << "/h\n";
}
```

```
class Jogo
{
private:

    string nome;
    float preco;
    int horas;
    float custo;

    ...
};
```

# Encapsulamento

- Benefícios do encapsulamento
  - Poder alterar a implementação sem mudar a interface
    - Só é preciso conhecer:
      - O que os métodos fazem
      - Que dados eles recebem e o que eles retornam

```
// exibe os dados de um jogo
void Jogo::exibir()
{
    cout << nome << " R$"
         << preco << " "
         << horas << "h = R$"
         << horas > 0 ? preco/horas : preco << "/h\n";
}
```

```
class Jogo
{
private:

    string nome;
    float preco;
    int horas;

    ...
};
```

# Classes e Registros

- E se as designações public e private forem omitidas?
  - Membros de classes são privados por padrão
  - Membros de registros são públicos por padrão

Membros Privados

```
class Jogo
{
    string nome;
    float preco;
    int horas;
};

Jogo gow;
gow.horas = 5;
```

✗

Membros Públicos

```
struct Jogo
{
    string nome;
    float preco;
    int horas;
};

Jogo gow;
gow.horas = 5;
```

✓

# Classes e Registros

- Registros podem conter funções
  - C++ trata registros e classes de forma similar
  - Exceto pelo **nível padrão** de acesso aos membros

```
struct Jogo
{
   string nome;
   float preco;
   int horas;

   void adquirir(const string & nome);
   void atualizar(float valor);
   void exibir();
   void jogar(int tempo);
};
```

Mas tipicamente registros são usados como um POD (Plain-Old Data)

# Classes e Registros

- Classes não precisam ter uma seção privada explícita
  - Mas é uma boa prática criá-las

Membros Privados

```
class Jogo
{
   string nome;
   float preco;
   int horas;
   float custo;

public:
   void adquirir(const string & nome);
   void atualizar(float valor);
   void exibir();
   void jogar(int tempo);
};
```

A seção private reforça a ocultação de dados e que os membros não fazem parte da interface pública da classe.

# Resumo

- A criação de objetos aloca memória
  - Objetos são como variáveis
  - Classes são como tipos
- O controle de acesso aos membros
  - Permite ocultar/encapsular
    - O formato de armazenamento
    - Os detalhes da implementação
  - Estabelece uma interface para a classe
    - Métodos ditam as operações possíveis