Centro
Integrado de
Tecnologia da
Informação

C com introdução a OO

Centro Integrado de Tecnologia da Informação

Aula 9

Ronald Dener - Instrutor
Matheus Soares - Monitor

17 / outubro

- Orientação a Objetos
- C++
- Herança

17 / outubro

- Polimorfismo
- Duvidas
- Exercícios

Orientação a Objetos

- Paradigma de programação em que tudo deve ser tratado com um "objeto", com seu estado(atributos) e comportamentos(métodos).
- Atributos são como variáveis internas do objeto, e métodos são como funções que o objeto pode executar.
- O modelo para um objeto é a "classe", que define seus atributos e métodos.
- Um objeto particular na memória é denominado "instância" de sua classe, e o ato de criar um objeto é chamado de "instanciar" sua classe.

C++

- Criado em 1983 por Bjarne Stroustrup sob o nome de "C com classes", com o propósito de introduzir conceitos de alto nível à velocidade de uma linguagem de baixo nível.
- Por ser, no seu início, uma extensão da linguagem C, permite também o uso dos paradigmas disponíveis nessa outra.
- Códigos escritos em C são quase sempre compatíveis com compiladores de C++.

C++ - Classes

- Em C++, para definir uma classe, usamos a palavra reservada class.
- Como numa struct, definimos os seus atributos e seus métodos. Por convenção, o nome da classe começa com letra maiúscula.

```
class Conta
{
    private:

    unsigned int numero;
    double saldo;
    string titular;

    public:

    double getSaldo();
    unsigned int getNumero();
    string getTitular();
    void deposita(float saldo);
    void retira(float saldo);
    Conta(unsigned int numero, string titular);
    Conta::~Conta();
};
```

C++ - Classes

- Usamos as palavras reservadas public, protected e private para restringir o acesso à membros da classe:
  - Membros **public** podem ser acessados em qualquer lugar.
  - Membros **protected** podem ser acessados por membros da mesma classe, "amigos" e classes derivadas.
  - Membros **private** só podem ser acessados por membros da mesma classe e por classes derivadas.
- Isso é feito para permitir que a abstração seja forçada: impedir que um programador que irá usar sua classe acesse diretamente alguns membros e use somente alguns métodos, para que você possa mudar o funcionamento interno da classe sem prejudicar o trabalho dele.
- Isso, junto à criação de métodos para acessar os atributos internos, é chamado de Encapsulamento.

C++ - Classes

- Definimos os métodos dentro da classe, mas podemos implementá-los fora dela:

```
double Conta::getSaldo()
{
    return saldo;
}

void Conta::deposita(float saldo)
{
    this->saldo += saldo;
}
```

O operador :: (codeblocks) define o escopo acessado.

O ponteiro **this** aponta para aquela instância da classe, e ajuda quando há conflito de nomes.

- Isso permite que deixemos a definição da classe em um arquivo, disponível para outros programadores, e precompilemos outro arquivo, que contém as implementações dos métodos.

C++ - Classes

- Construtor é um método chamado sempre que a classe é instanciada.
- Não tem tipo de retorno.
- É executado após a alocação de memória para o objeto.

```
Conta::Conta(unsigned int numero, string titular)
{
    this->numero = numero;
    this->saldo = 0;
    this->titular = titular;
}
```

- É sempre definido implicitamente um construtor sem parâmetros, que é usado por padrão e com alocação dinâmica, mas esse pode ser definido explicitamente. Se QUALQUER construtor for definido, esse construtor padrão é perdido.
- É muito útil para inicializar certos atributos de todos as instâncias da classe (saldo de uma conta inicia com 0, pilha sempre começa vazia, etc.)

C++ - Classes

- Analogamente ao Construtor, Um Destrutor é um método chamado quando a instância sai de escopo ou é desalocada.
- Um destrutor não pode aceitar parâmetros.

```
Conta::~Conta()
{
}
```

- Um destrutor padrão é implementado, que não faz nada.
- É útil implementar um destrutor quando há objetos alocados dinamicamente dentro da classe, já que é preciso desalocá-los antes de destruir o objeto.
- Normalmente não é preciso chamar o destrutor explicitamente, pois ele é chamado automaticamente. Chamar um destrutor duas vezes causa erro de execução.

C++ - Alocação Dinâmica

- Em C++ temos o operador new para alocar memória. Ele é preferível ao malloc e afins, pois chama o construtor da classe.

```
//chama o construtor padrão, sem parâmetros
Conta* contaDinamica = new Conta;
Conta contaEstatica;
int tamanho = 20;
Conta* ContaArray = new Conta[tamanho];

//Chama o contrutor não-padrão.
Conta contaEstaticaInicializada(4321,"Tiago");
Conta* contaDinamicaInicializada = new Conta(1234,"Adriano");

//Conta* ContaArrayInicializado = new Conta[10](1357,"vetor");
//Proibido pela ISO C++, mas possível.
```

C++ - Alocação Dinâmica

- Para desalocar, usamos o operador delete:

```
delete contaDinamica;
delete contaDinamicaInicializada;
delete [] ContaArray;
```

- É preciso usar o delete para desalocar memória alocada com o new.

- É preciso tomar cuidado, pois chamar delete duas vezes no mesmo objeto causará erro de execução (como acontece com o free()).

- Se for usado new, use delete.
- Se for usado malloc(), use free().

C++ - Referência

- Além de variáveis estáticas e ponteiros, temos em C++ as referências.

```
int a = 10;
int& b = a;
b = 20; /// a = 20
```

- Usando o operador &, podemos criar referências, que se ligam à variáveis e passam a ser "outro nome" para ela.

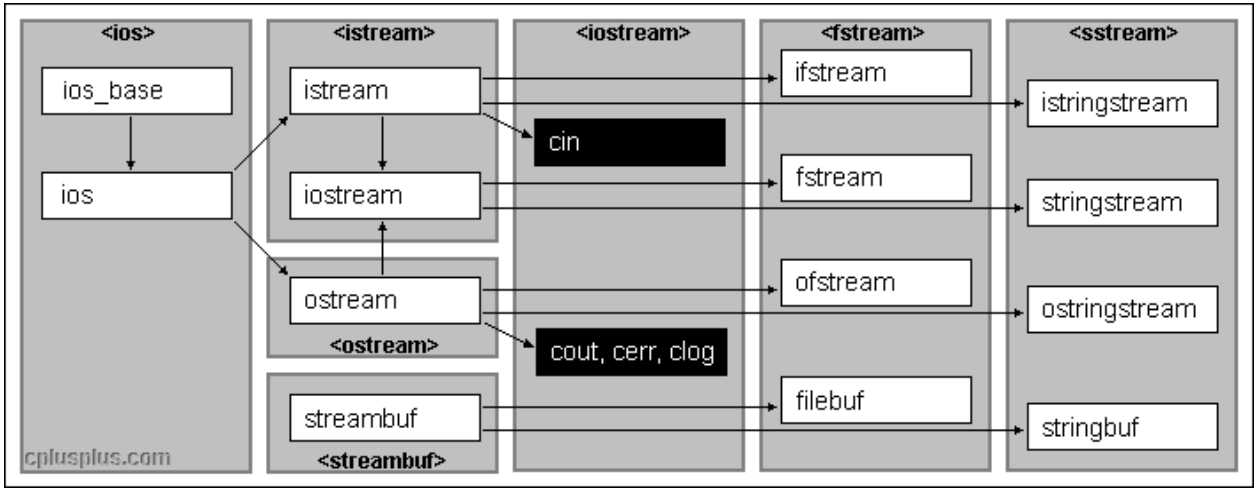
- Referências só podem ser ligadas na sua inicialização.

- Isso pode ser usado em funções para passar parâmetros por referencia sem precisar se preocupar com ponteiros.

```
void swap(int& a, int& b )
{
    int aux = a;
    a = b;
    b = aux;
}
```

C++ - Streams

- Em C++, a entrada e saída de dados é efetuada através da biblioteca iostream
- Um "stream" representa um dispositivo de entrada ou saída, tal como o teclado e o console, ou arquivos.



C++ - Streams

- A entrada padrão, cin, é usada com o operador >>:

```
cin >> b;
```

- Pode ser usado encadeado:

```
cin >> idade >> peso >> altura;
```

- A saída padrão, cout, é usada com o operador <<:

```
cout << a;
```

- Também pode ser usado encadeado:

```
cout << idade << peso << altura << endl;
```

- O endl é a quebra de linha ('\n').

C++ - Streams

- Na biblioteca fstream temos as classes:
  - **ifstream**, para arquivos abertos em modo leitura.
  - **ofstream**, para arquivos abertos em modo escrita.
  - **fstream**, para arquivos abertos em modo escrita e leitura.

```
int main () {

    int a;
    ofstream arquivoSaida;
    ifstream arquivoEntrada;

    arquivoSaida.open ("exemplo.txt");
    arquivoEntrada.open ("exemplo2.txt");

    arquivoEntrada >> a;
    arquivoSaida << a;

    arquivoEntrada.close();
    arquivoSaida.close();

    return 0;
}
```

Herança

- Mecanismo da Orientação a Objeto que permite criar novas classes aproveitando as características existentes em uma classe a ser estendida.
- A nova classe é chamada de subclasse (ou classe derivada) e a classe existente é chamada de superclasse (ou classe base).
- Uma subclasse pode servir como superclasse, e assim por diante, permitindo a criação de uma hierarquia de classes relacionadas através da herança.
- Herança permite que código seja reutilizado, e ajuda as classes a representar melhor a realidade.

Herança – Em C++

- Em C++, temos suporte à herança através da seguinte sintaxe:

```
class Base
{
    ///Membros
};

class Derivada : /**public,private,protected*/ Base
{
    ///Outros Membros
};
```

- O acesso aos membros da classe base será no máximo o definido na herança.

- Se o modificador de acesso for omitido, será usado private.

Herança - Exemplo

- Por exemplo:

```
class Carro{
private:
    string dono;
protected:
    int vagas;
public:
    void setDono(string nome);
    string getDono();
    virtual void setVagas(int num);
    int getVagas();
};
```

```
class PickUp : public Carro{
    int carga;
public:
    void setCarga(int num);
    int getCarga();
    void setVagas(int num);
};
```

Herança - Construtores

- Uma classe derivada pode usar os construtores da classe base. O construtor padrão é usado, mas isso pode ser modificado:

```
class Base
{
    public:
    Base(int);
};
```

```
class Derivada : /**public,private,protected*/ Base
{
    Derivada(int a);
};

Derivada::Derivada(int a) : Base(a)
{
    ///Implementacao do Construtor
}
```

- Note que o construtor de Base é chamado com parâmetro a, recebido na chamada do construtor de Derivada.
- O construtor da classe base sempre será executado antes do construtor da derivada.

Herança – Sobrecarga de métodos

- Uma classe derivada pode redefinir métodos da classe base, simplesmente declarando um novo método com a mesma assinatura. O método da classe base ainda estará disponível para ela através do operador :: . Os métodos são ditos sobrecarregados.

```
class Base
{
    public:
    Base(int);
    void funcao(); ///metodo da base.
};

class Derivada : /**public,private,protected*/ Base
{
    public:
    Derivada(int a);
    void funcao(); /// redefinida na derivada.
};

void Derivada::funcao()
{
    Base::funcao();
}
```

Herança – Herança Multipla

- Classes derivadas podem herdar de mais de uma classe base:

```
class Base
{
    public:
    Base(int);
};

class OutraBase
{
    public:
    OutraBase();
};

class Derivada : /**public,private,protected*/ Base,OutraBase
{
    Derivada(int a);
};

Derivada::Derivada(int a) : Base(a),OutraBase()
{
    ///Implementacao do Construtor
}
```

Se houver conflitos entre atributos e métodos das classes bases, o operador :: pode ser usado para resolvê-los.

Polimorfismo - Definição

- Capacidade de assumir formas diferentes.
- Em OO, se refere à capacidade de um objeto de uma classe base poder armazenar um objeto de uma de suas classes derivadas.
- Em C++, isso é feito através de ponteiros: Um ponteiro para uma classe base pode armazenar o endereço de uma classe derivada.

Polimorfismo – Metodos virtuais

- Quando chamamos um método sobrecarregado de um ponteiro de classe base que armazena um objeto de uma classe derivada, o método chamado é o da classe base. Mas podemos usar a diretiva virtual no método da base para forçar a checagem em tempo de execução de qual método deve ser usado:

```
class Base
{
    public:
    Base(int);
    virtual void funcao(); ///metodo da base.
};

class Derivada : /**public,private,protected*/ Base
{
    public:
    Derivada(int);
    void funcao(); /// redefinida na derivada.
};

int main()
{
    Base* objetoB = new Base(10);
    Base* objetoD = new Derivada(10);
    objetoB->funcao(); /// Chama o método da base.
    objetoD->funcao(); /// Chama o método da derivada.
}
```

Polimorfismo – Classes virtuais puras

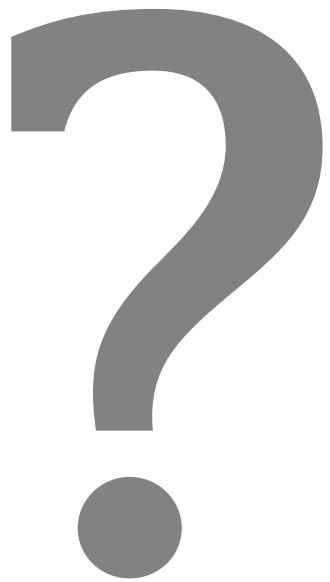
- Podemos, em C++, definir classes que nunca serão instanciadas, definindo somente a sua interface, a ser implementada nas sua classes derivadas.
- Isso é feito declarando métodos virtuais puros:

```
class Base
{
    public:
    Base(int);
    virtual void funcao() = 0; ///metodo da base.
};

class Derivada : /**public,private,protected*/ Base
{
    public:
    Derivada(int);
    void funcao(); /// redefinida na derivada.
};

int main()
{
    //Base* objetoB = new Base(10); // Não será mais possível,
                                    // pois a classe Base é virtual pura
    Base* objetoD = new Derivada(10);
    objetoD->funcao(); /// Chama o método da derivada.
}
```

Dúvidas



Exercício 1

1- Implemente a classe pilha, com métodos push(int), pop(), top() e size().

2 - Implemente a classe Fila, com métodos push(int), pop(), front() e size().