C++

Prof. Adriano Caminha

adriano@puvr.uff.br

ICEx/UFF

Volta Redonda-RJ

Introdução a C++

→ Os programas mais simples em C++ possuem a mesma estrutura básica que os programas em C.

→ A programação orientada a objetos inclui também estruturas avançadas como classes e objetos. Mas é preciso inicialmente conhecer comandos básicos específicos de C++, notações específicas, etc.

Exemplo 1: (`ex1.cpp`)

```
#include <iostream>
using namespace std;
main()
{
    cout << "Teste" << endl;

}
```

Onde:

`#include <iostream>` → instrução de preprocessamento para inclusão do arquivo com a biblioteca de entrada e saída do C++.
`use namespace std`→ indica o uso da biblioteca *std* (standard) de entrada e saída.
`int main` → *main* é o nome da função principal.
`()` → parâmetros: neste caso, sem parâmetros.
`{` → início da função principal.
`cout` → imprime na tela a mensagem. Usar o operador "<<" indicando que a string vai para a saída (tela).
`endl` → indicação de final de linha (adiciona quebra de linha).
`return 0` → indicação de fim de execução (0 = sem erro).
`}` → fim da função principal.

O Compilador `g++` (Linux GNU C++ Compiler)

→ Para compilar e executar os códigos dos exercícios e exemplos, utilizar o compilador `g++`, disponível em quase todas as distribuições Linux.

Utilização

→ Editar o programa com um editor de texto simples, como o **`gedit`** (aplicativos/acessórios/editor de textos).

→ Criar uma nova pasta (será a pasta de projetos para C++), que deverá ser copiada para um pendrive ao final de cada aula, como backup de segurança.

→ Abrir o *Terminal* (Ctrl+Alt+T) e usar o comando "`cd`" do Linux para ir para a pasta do projeto. Exemplo (se a pasta foi criada no desktop):

...$ **cd Área**[TAB]
→ TAB para auto complementar (Ex: "Área\ de\ Trabalho").
...$ **cd suapasta**

→ Salvar o arquivo com este nome, `ex1.cpp`, e extensão (`cpp` = c plus plus), isso ativará as cores do editor **`gedit`** para a linguagem C++.

- Compilar: **g++ ex1.cpp**
- Executar: **./a.out**

- Ou: **g++ *-o ex1.out* ex1.cpp** (isso define o nome do executável)
- E executar com: **./ex1.out**

O Dev-C++ (Ferramenta Windows para C e C++)

– Para os usuários do Windows, está disponível a ferramenta gratuita Dev-C++, que possibilita compilar e executar projetos em C e C++.

Utilização

– Criar um projeto C++: **(para cada exemplo ou exercício)**

  – Arquivo/Novo/Projeto (ou use o botão correspondente).

  – Console Application/Projeto C++/Nome (o nome do exemplo).

  – Escolha a pasta (se necessário crie uma para seu projeto).

  – Altere o código padrão fornecido.

  – Salve conforme adiciona seus comandos (use o nome do exemplo com extensão ".cpp" em vez do nome padrão "main.cpp").

  – Pressione a tecla F9 para compilar e executar.

Comando de Entrada via Console

– O comando `cout` mostra uma string na console ou terminal.

– O comando `cin` lê do terminal e armazena na variável indicada após o operador ">>".

Exemplo 2: `ex2.cpp`

```
#include <iostream>
using namespace std;
main()
{     char nome[80];
      int idade;

      cout<<"Nome? ";
      //seta da string para a saida <<
      cin>>nome;
      //seta da entrada para a variavel >>

      cout<<"Idade? ";
      cin>>idade;

      cout << "\nOla, " << nome << ", tudo OK aos ";
      cout << idade << " anos?\n" << endl;
}
```

Funções em C++

-Funções em C++ são semelhantes a funções em C.
-Em P.O.O., são adicionados alguns operadores e modificadores de acesso na sintaxe das funções, o que será mostrado mais tarde.

Exemplo 3: ex3_funcao1.cpp

→ Utilização de uma função simples e sua chamada a partir de outra função, para mostrar uma mensagem na tela.

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

//funcao mensagem
void mensagem()
{     cout<<"\nFuncao funcionando...\n\n";
}

//funcao main
main()
{     mensagem(); //chamada da funcao
}
```

Exemplo 4: ex4_funcoessomasub.cpp

→ Utilização de funções simples com operadores aritméticos, semelhantes aos operadores da linguagem C, para mostrar resultados de somas e subtrações na tela.

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

int soma(int a, int b)
{     int r = a + b;
      return r; //ou return a+b;
}

int subtr(int a, int b)
{     int r = a - b;
      return r; //ou return a-b;
}

main()
{     int z, x=5, y=3;

      z = soma(x,y);
      cout<<"\n5+3 = "<<z<<"\n";
      z = subtr(x,y);
      cout<<"5-3 = "<<z<<"\n";

      //ou, chamando dentro do cout: (mais um exemplo)
      cout<<"\nNo cout:\n5+3 = "<<soma(x,y)<<"\n";
      cout<<"5-3 = "<<subtr(x,y)<<"\n"<<endl;
}
```

Exercício

– Escreva um programa que contenha funções para calcular a soma, a subtração, o produto e a divisão entre dois números inteiros.

– No programa principal leia dois inteiros x e y e imprima o resultado da soma, da subtração, do produto e da divisão entre os dois inteiros lidos através das chamadas das funções.

– Mostre também o resultado das seguintes expressões (usando as funções, chamando uma dentro da chamada da outra):

```
((x + y) * (x - y))
((x * y) + (x / y))
```

POO – Programação Orientada a Objetos

– A POO foi criada para aproximar o mundo real do mundo virtual.

– A ideia fundamental é simular o mundo real dentro do computador, por isso utiliza-se o conceito de objetos (nosso mundo é composto de objetos).

– Na POO, o programador é responsável por moldar o mundo dos objetos e definir como eles devem interagir entre si.

– Definição de Análise e Projeto POO: “Os objetos conversam uns com os outros através de mensagens (chamadas de funções) e o papel principal do programador é especificar que mensagens cada objeto pode receber e qual ação aquele objeto deve realizar ao receber cada mensagem.”

Principais Conceitos da POO

– **CLASSES** → são *modelos* para os objetos que compõem o sistema. São *novos tipos* definidos pelo programador com declarações de *atributos* (propriedades) e *comportamentos* (funções), para gerar entidades na memória chamadas de *objetos*.

– **ATRIBUTOS** → *propriedades* ou *dados* dos objetos.

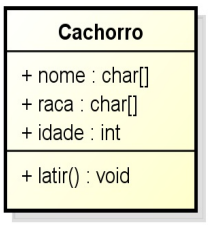
– **MÉTODOS** → funções declaradas nas classes, que serão executadas *através* dos objetos. Os métodos do objeto devem ser os únicos capazes de alterar os valores das propriedades do próprio objeto.

– **OBJETO** → entidades na memória, criadas a partir das classes, que devem receber valores para os atributos e devem conter métodos para manipular estes valores e executar tarefas diversas usando estes valores.

Diagrama de Classes

– **UML** → (*Unified Modeling Language*) linguagem usada pra projetar sistemas OO, que possui símbolos pra definir classes, objetos, etc.

Representação Gráfica



→ Nome da classe

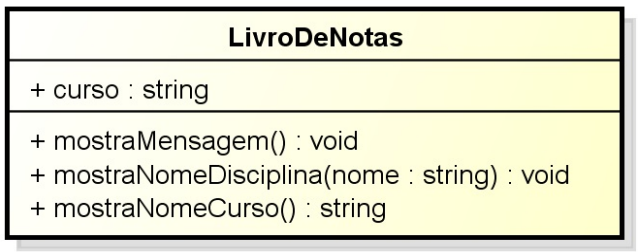
→ Atributos
+ → público (visível)
: → tipo

→ Método

Exemplo "LivroDeNotas" com um método [Deitel]:


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

class LivroDeNotas
{     public: //especificador de acesso
            void mostraMensagem()  //método
            {     cout<<"Livro de Notas"<<endl;
            }
}; //fim da classe

main()
{     LivroDeNotas livro; //cria um objeto livro
      livro.mostraMensagem();  //chama o método
}
```

Exemplo "LivroDeNotas" com um atributo:




```
#include <iostream>
#include <string> //para usar o tipo string
using namespace std;
class LivroDeNotas
{     public:
      string curso; //atributo

      public: //especificador de acesso
            void mostraMensagem()  //método
            {     cout << "Livro de Notas" << endl;
            }

            //método recebendo um parâmetro
            void mostraNomeDisciplina(string nome)
            {     cout << "Disciplina: " << nome << endl;
            }

            //método que retorna o valor de um atributo
            string mostraNomeCurso()
            {     return curso;
            }
}; //fim da classe
```

```
main()
{     LivroDeNotas livro; //cria um objeto livro

      //valor para o atributo
      livro.curso = "Ciencias da Computacao";

      livro.mostraMensagem();  //chama o método

      //mostra valor atributo
      cout<<"Curso: "<< livro.mostraNomeCurso() << endl;

      livro.mostraNomeDisciplina("Progr 2"); //com parâmetro
}
```

Exercícios

1. Desenhe o diagrama de classe e escreva o código para uma classe que representa um CD, com atributos titulo, artista e ano, com um método para mostrar os valores dos atributos do objeto. Na *main*, crie três objetos da classe CD, atribua valores para os atributos de cada objeto e mostre esses valores utilizando o método.

2. Desenhe um diagrama e escreva um código para uma classe que representa uma equação, com atributos inteiros x e y, métodos para calcular a soma, a subtração, o produto e a divisão entre dois números inteiros (os atributos x e y) e um método para calcular a seguinte expressão (usando os outros métodos da classe):

```
((x + y) + (x – y)) - ((x * y) + (x / y))
```

Na *main*, crie um objeto da classe, leia dois inteiros x e y, atribua os valores lidos aos atributos do objeto, imprima os resultados de todos os métodos do objeto.

Funções Membro

– Funções ou métodos *membro* são métodos declarados dentro da classe. Por exemplo, o método `mostraNomeDisciplina`, do exemplo anterior, pode ser classificado como método membro:

```
//método recebendo um parâmetro
void mostraNomeDisciplina(string nome)
{     cout << "Disciplina: " << nome << endl;
}
```

– Em geral possuem tarefas de cálculos e se diferenciam dos outros métodos da classe (construtores, acessadores, etc), recebendo esta denominação.

– **Variáveis locais** → são variáveis declaradas dentro do corpo da função e só podem ser utilizadas na função (mesma denominação utilizada na linguagem C).

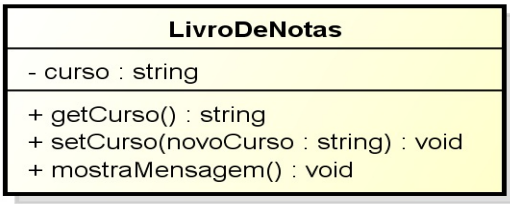
Membros de Dados

- Uma classe possui, em geral, um ou mais **métodos membros**, usados para manipular os atributos.
- Esses atributos são também chamados de **membros de dados** e são declarados dentro da classe, mas fora dos métodos membro.
- Todo *objeto* possui sua própria cópia de atributos *com seus próprios valores*.
- Os métodos membros devem ser declarados como `public`.
- O especificador de acesso *padrão* para os atributos é `private`, mas é boa prática declarar sempre explicitamente os especificadores buscando manter sempre a clareza do código.

Getters & Setters

- Encapsulamento, um conceito importante em P.O.O., determina que os atributos dos objetos pertencem apenas a estes e somente estes devem os manipular. Ou seja, deve-se declarar os atributos como privados (especificador `private`) sempre que possível. O atributo fica então “oculto” ou “encapsulado”.
- Uma classe possui, em geral, um ou mais **métodos acessadores**, ou ***getters***, usados para ***obter*** os valores dos respectivos atributos privados. Estes métodos possuem o mesmo tipo do atributo correspondente e apenas retornam o valor deste atributo.
- Uma classe possui, em geral, um ou mais **métodos modificadores**, ou ***setters***, usados para ***alterar*** os valores dos respectivos atributos privados. Estes métodos recebem um novo valor por parâmetro, substituem o valor do atributo por este novo e nada retornam.

Exemplo "LivroDeNotas" com getters & setters:



← Atenção ao sinal “-” de *private*

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

class LivroDeNotas
{     private: //especificador de acesso encapsulamento
      string curso; //atributo privado
      public:
            //getter = retorna o valor do atributo
            string getCurso()
            {     return curso; }

            //setter = altera o valor do atributo
            void setCurso(string novoCurso)
            {     curso = novoCurso; }
```

```
            //método membro
            void mostraMensagem()
            {     cout << "Livro de Notas" << endl;
            }

};  //fim da classe

main()
{     LivroDeNotas livro; //cria um objeto livro

      //altera valor do atributo (valor inicial)
      livro.setCurso("Ciencias da Computacao");

      //método membro
      livro.mostraMensagem();

      //mostra o valor do atributo
      cout<<"Curso: "<< livro.getCurso() << endl;
}
```

Exercícios

3. Desenhar uma classe que represente **uma pessoa para uma agenda** de e-mails, fones e outros dados que forem interessantes para o usuário. Escrever um programa em C++ para a classe, incluindo atributos privados e os respectivos métodos getters & setters. No método `main()`, criar três objetos da classe, atribuir valores aos atributos e os imprimir.

4. Desenhar uma classe que represente **um veículo da sua escolha (um carro, uma moto ou outro) para um cadastro de veículos**. Utilize como atributos os dados que forem interessantes para o usuário do cadastro. Escrever um programa em C++ para a classe, incluindo atributos privados e os respectivos métodos getters & setters. No método `main()`, criar três objetos da classe, atribuir valores aos atributos e os imprimir.

5. Desenhar uma classe que represente **uma fruta para um menu de saladas de frutas**. Utilize como atributos os dados que forem interessantes para o usuário do cadastro. Escrever um programa em C++ para a classe, incluindo atributos privados e os respectivos métodos getters & setters. No método `main()`, criar três objetos da classe, atribuir valores aos atributos e os imprimir.

Métodos Contrutores

- Métodos que auxiliam na criação do objeto, em geral recebendo argumentos para serem atribuídos aos atributos do objeto logo na sua instanciação (criação), já os inicializando.

  Exemplo:

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

class LivroDeNotas
{
      private:
            string nomeCurso;


      public:
            //metodo construtor – usar o mesmo nome da classe
            LivroDeNotas (string nome)
            {     setNomeCurso(nome);     }

            //setter
            void setNomeCurso(string nome){     nomeCurso=nome;}

            //getter
            string getNomeCurso() { return nomeCurso; }

            //funcao membro mensagem
            void mostraMensagem()
            { cout << "Bem vindo ao Livro de Notas do curso "
                        << getNomeCurso() << "!" << endl;
                  //uso do getter para obter o valor
                  //(convencao, pois poderia ser usado "nomeCurso"
                  //(atributo) diretamente)
            }

}; //fim da classe

main()
{
      //cria 2 objetos usando o construtor
      LivroDeNotas livro1 ("c++");
      LivroDeNotas livro2 ("Java");

      //exibe valores inicializados pelo construtor
      cout << "Livro de Notas 1:"
             << livro1.getNomeCurso()
             << "\n Livro de Notas 2:"
             << livro2.getNomeCurso()
             << endl;

}
```

Classes em Arquivos Separados

- Esta técnica facilita na reutilização dos módulos ao copiar o(s) arquivo(s) para um novo projeto.
- Facilita também a manutenção e correção de erros.

Exemplo:

Arquivo "`LivroDeNotas.h`"
→ Essa extensão ".h" significa *header* ou cabeçalho.
→ Este arquivo deve estar na mesma pasta (ou no mesmo projeto Dev-C++) do arquivo principal, mostrado a seguir.

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

class LivroDeNotas
{     private:
            string nomeCurso;

      public:
            //metodo construtor
            LivroDeNotas (string nome){ setNomeCurso(nome);}

            //setter
            void setNomeCurso(string nome){    nomeCurso=nome;}

            //getter
            string getNomeCurso() { return nomeCurso; }

            //funcao membro mensagem
            void mostraMensagem()
            { cout << "Bem vindo ao Livro de Notas do curso "
                        << getNomeCurso() << "!" <<endl;
            }
}; //fim da classe
```

Arquivo "`MainLivroDeNotas.cpp`"

```
#include <iostream>
#include <string>
#include "LivroDeNotas.h" //aspas, pois não está na biblioteca padrão
using namespace std;

main()
{
      //cria 2 objetos usando o construtor
      LivroDeNotas livro1 ("c++");
      LivroDeNotas livro2 ("Java");

      //exibe valores inicializados pelo construtor
      cout  <<"Livro de Notas 1:" << livro1.getNomeCurso()
            <<"\nLivro de Notas 2:" << livro2.getNomeCurso()
            << endl;

}
```

Exercício

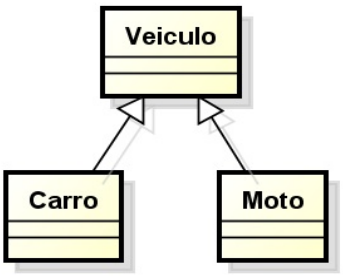
1. Reescrever os programas em C++ dos exercícios 1 a 5 anteriores utilizando métodos construtores e colocando as classes e o *main* em arquivos separados.

Herança

- Herança acontece quando **uma nova classe absorve atributos e métodos de uma classe existente**. É um dos pontos principais da POO.

- Uma classe que **herda** de outra é chamada de **classe derivada** ou **subclasse** ou ainda **classe filha**. A classe que possui subclasses é chamada de **classe base** ou **superclasse** ou ainda **classe mãe**.

- **Exemplo**: Um carro é um veiculo (quaisquer propriedades e comportamentos de um veículo também são propriedades de um carro).



- A subclasse herda **membros** (atributos e métodos) **não *private*** da superclasse. Isso significa que ao escrever o código da subclasse, **não é necessário repetir o código já escrito na superclasse** definido como *public* e também que é preciso usar getters para acessar atributos *private*.

- Mas ***objetos* da classe filha possuem todos os membros da classe mãe, além dos definidos na própria classe filha**. Os objetos são montados a partir de algo como uma união das classes na hierarquia de herança.

- Uma classe derivada representa um grupo mais **especializado** de objetos.

- O relacionamento **É UM** da UML representa a herança.

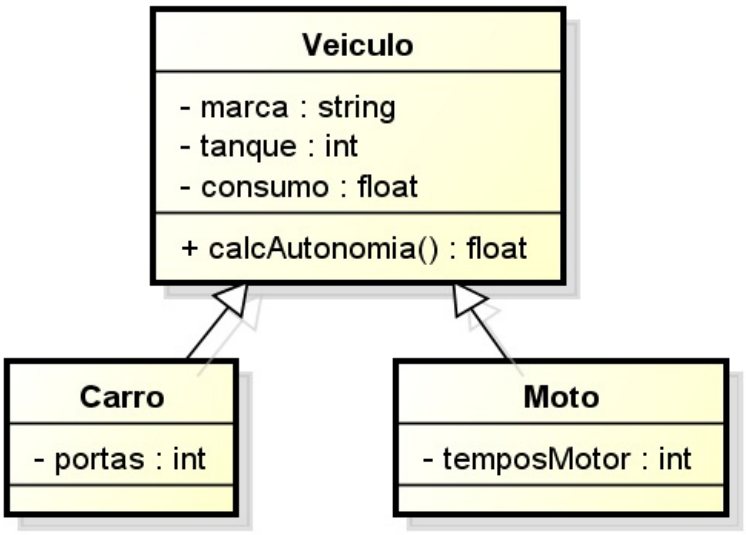
- Em um relacionamento **É UM**, **um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe básica**.

Exemplo:

- Objetos de Carro ou de Moto possuem atributos e métodos não private de Veiculo, além dos definidos em cada respectiva subclasse.

- Objetos de Carro e Moto são objetos especializados de Veiculo.

- E objetos de Carro e Moto também podem ser classificados como objetos de Veiculo.



Arquivo "Veiculo.h":

```
//includes só no main neste exemplo... faça o teste...
class Veiculo
{     private:
               string marca;
               int tanque;
               float consumo;

      public:
              string getMarca() { return marca; }
              int getTanque() { return tanque; }
              float getConsumo() { return consumo; }

              void setMarca(string m) { marca = m; }
              void setTanque(int t) { tanque = t; }
              void setConsumo(float c) { consumo = c; }

              float calcAutonomia()
              { return tanque * consumo; }

}; //fim da classe Veiculo
```

Arquivo "Carro.h":

```
class Carro: public Veiculo //Carro herda de Veiculo
{     private:
               int portas;
      public:
              int getPortas() { return portas; }
              void setPortas(int p) { portas = p; }
};
```

Arquivo "MainVeiculoCarro.cpp":

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

//depois do using namespace, incluir as classes, usando aspas
#include "Veiculo.h"
#include "Carro.h"

main()
{   int p, t;
    float con;
    string m;
    Carro c;

    cout << "Portas: ";
    cin >> p;
    c.setPortas(p);
    cout << "Tanque: ";
    cin >> t;
    c.setTanque(t);
    cout << "Consumo: ";
    cin >> con;
    c.setConsumo(con);

    cout << "Marca: ";

    cin.clear(); //ATENÇÃO: limpar buffer
    cin.sync(); //getline não funciona em todos os compiladores!
    getline(cin, m); //nomes compostos

    //OU, de forma simples e que funciona em todos os compiladores:
    //cin >> m; //nomes simples (sem espaços)

    c.setMarca(m); //metodos herdados

    cout << "\n\nCarro: " << c.getMarca()
         << "\nAutonomia: " << c.calcAutonomia()
         << endl;

}
```

Interfaces de Objetos

- Representam um *planejamento da funcionalidade* do sistema;
- Definem e padronizam o modo como as interações acontecem no sistema, *como os objetos do sistema se comunicam*.
- A interface de uma classe descreve os serviços (métodos) que os clientes (objetos e outros) de uma classe podem utilizar e como esses serviços devem ser solicitados, mas não define *como* a classe executa os serviços (internamente).
- Consistem em métodos *public* da classe, declarados como *protótipos de métodos* que terminam sempre com o sinal **;** (ponto e vírgula)

Exemplo "`LivroDeNotas`" com Interface de Objetos

- **1° passo**: editar o arquivo "`LivroDeNotas.h`"
  - Definição da classe `LivroDeNotas` contendo os protótipos dos métodos. Esta definição representa a interface da classe.
  - Em um sistema produzido para um cliente no mercado de software, este arquivo pode ser mostrado em discussões sobre o funcionamento do sistema. Já o próximo arquivo, com implementações dos métodos, poderia ser, por exemplo, segredo de produção e não seria mostrado ao cliente.

Arquivo "LivroDeNotas.h":

```
#include <string>
class LivroDeNotas
{     private:
            string nomeCurso;

      public:
            LivroDeNotas (string); //metodo construtor
            void setNomeCurso(string);
            string getNomeCurso();
            void mostraMensagem();
}; //fim da classe
```

- **2° passo**: editar o arquivo "`LivroDeNotas.cpp`"
  - Os nomes dos arquivos ".h" e ".cpp" devem ser os mesmos.
  - Define a implementação da classe `LivroDeNotas` contendo as implementações dos corpos dos métodos.
  - Em um sistema produzido para um cliente no mercado de software, este arquivo não seria mostrado ao cliente.
  - **::** → operador *resolução de escopo binário* usado para amarrar cada método membro à definição da classe na interface `LivroDeNotas.h`.

Arquivo "LivroDeNotas.cpp":

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

#include "LivroDeNotas.h" //interface a ser implementada

            //metodo construtor
            LivroDeNotas::LivroDeNotas (string nome)
            { setNomeCurso(nome);   }

            //setter
            void LivroDeNotas::setNomeCurso(string nome)
            {     nomeCurso=nome;}

            //getter
            string LivroDeNotas::getNomeCurso()
            {     return nomeCurso; }

            //funcao membro mensagem
            void LivroDeNotas::mostraMensagem()
            { cout << "Bem vindo ao Livro de Notas do curso "
                      << getNomeCurso() << "!" <<endl;
            }

//fim das implementações
```

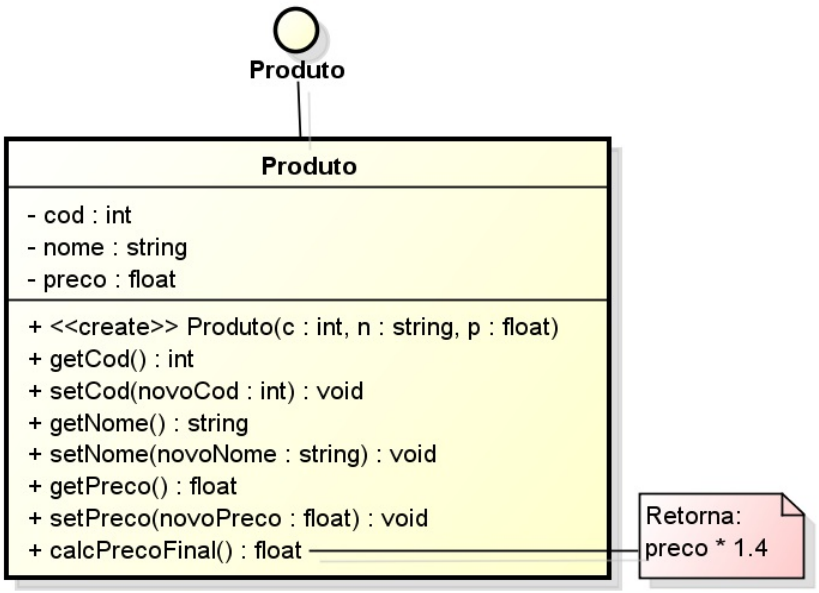
- **3° passo**: editar o arquivo que contém o `main()`.

Arquivo "MainLivroDeNotas.cpp"

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

#include "LivroDeNotas.h"

main()
{     //cria 2 objetos usando o construtor
      LivroDeNotas livro1 ("c++");
      LivroDeNotas livro2 ("Java");

      //exibe valores inicializados pelo construtor
      cout <<"Livro de Notas 1:" << livro1.getNomeCurso()
             <<"\nLivro de Notas 2:" << livro2.getNomeCurso()
             << endl;

}
```

Exercícios

1. Escrever um programa em C++ para implementar a seguinte classe, incluindo sua interface. No main(), criar dois objetos passando valores quaisquer para seus atributos através do método construtor e em seguida os imprimir.



2. Criar uma classe com atributos representando um objeto do mundo real, a partir do qual se possa definir pelo menos três atributos. Desenhar a classe com sua interface de objetos e implementar seus códigos em C++. No main(), criar dois objetos passando valores quaisquer para seus atributos através do método construtor e em seguida os imprimir.

3. Reescrever os programas em C++ dos exercícios 1 a 4 anteriores utilizando interfaces de objetos e colocando as classes e o *main* em arquivos separados. Não utilizar métodos construtores na Herança (esta parte será mostrada em seguida).

Métodos Contrutores em Herança

– Na herança, o método construtor da subclasse deve receber todos os valores para inicializar todos os atributos do objeto e repassar para sua superclasse imediatamente acima todos os valores que devem ser recebidos pelo método construtor dessa superclasse.

Exemplo:

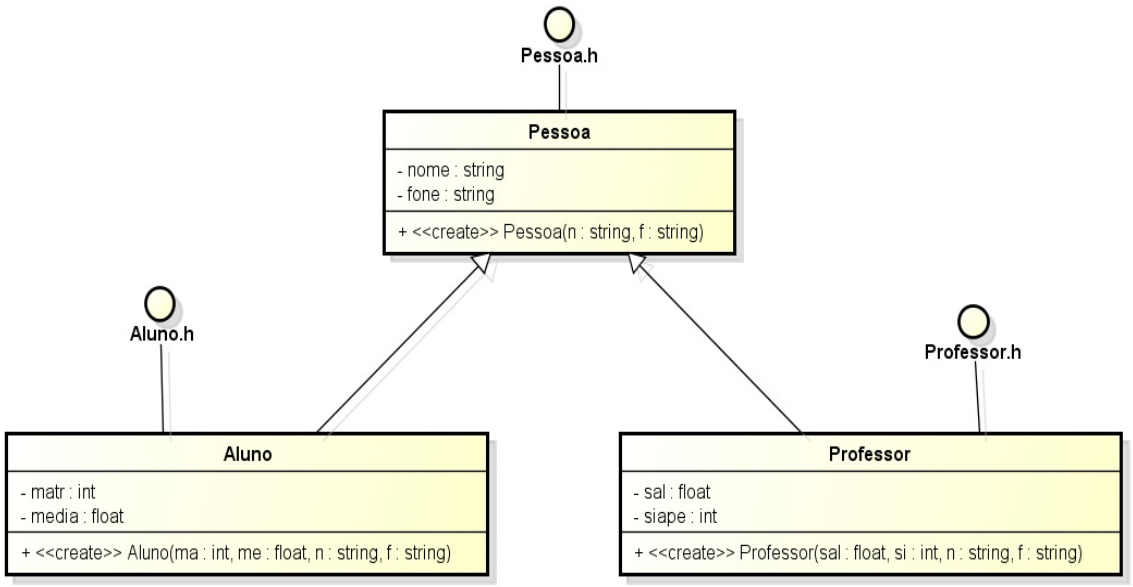
- Na superclasse Pessoa:

```
//metodo construtor
Pessoa (string n, string f);
```

- Na subclasse Aluno:

```
//metodo construtor com repasse
Aluno (int ma, float me, string n, string f):Pessoa(n, f);
```

Detalhando o exemplo:



Arquivo "`Pessoa.h`":

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

class Pessoa
{
 private:
  string nome;
  string fone;

 public:
  Pessoa(string, string); //metodo construtor
  void setNome(string);
  string getNome();
  void setFone(string);
  string getFone();
};
```

Arquivo "`Pessoa.cpp`":

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

#include "Pessoa.h"

    Pessoa::Pessoa(string n, string f) {
     setNome(n);
     setFone(f);
    }

    string Pessoa::getNome() { return nome; }
    string Pessoa::getFone() { return fone; }
    void Pessoa::setNome(string n) { nome = n; }
    void Pessoa::setFone(string f) { fone = f; }

//fim da implementação de Pessoa.h
```

Arquivo "Aluno.h":

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

class Aluno:public Pessoa
{
 private:
  int matr;
  float media;

 public:
  Aluno(int, float, string, string); //metodo construtor com repasse
  void setMedia(float);
  float getMedia();
  void setMatr(int);
  int getMatr();
};
```

Arquivo "Aluno.cpp":

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

#include "Pessoa.h"
#include "Aluno.h"

    Aluno::Aluno(int ma, float me, string n, string  f)
       :Pessoa(n, f) { //metodo construtor com repasse
     setMedia(me);
     setMatr(ma);
    }

    int Aluno::getMatr() { return matr; }
    float Aluno::getMedia() { return media; }
    void Aluno::setMatr(int ma) { matr = ma; }
    void Aluno::setMedia(float me) { media = me; }

//fim da implementação de Aluno.h
```

Arquivo "main.cpp":

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

#include "Pessoa.h"
#include "Aluno.h"

main()
{   Aluno aluno(12345, 8.5, "maria", "(24) 9999-9999");

    cout << "\n\nAluno:"
         << "\nMatricula: " << aluno.getMatr()
         << "\nNome: " << aluno.getNome()
         << "\nMedia: " << aluno.getMedia()
         << "\nFone: " << aluno.getFone()
         << endl;

}
```

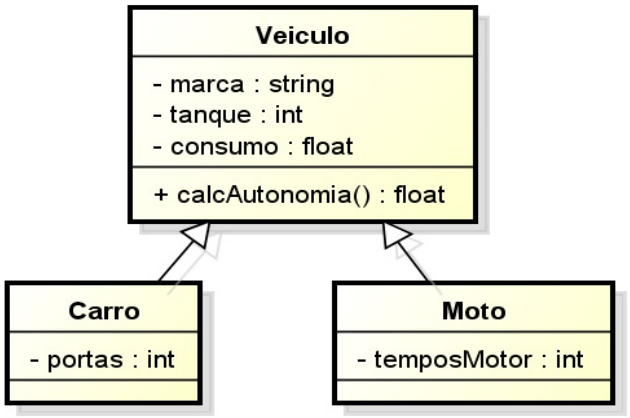
Exercícios

1. Completar o exemplo acima incluindo os arquivos da classe Professor. No main(), acrescentar um objeto “professor” passando valores quaisquer para seus atributos através do método construtor e em seguida imprimir os dois objetos.

2. Escrever um programa em C++ utilizando uma superclasse e duas subclasses com pelo menos dois atributos cada. Criar um tema para as classes e desenhar o UML incluindo também as interfaces de objetos e os construtores. Escrever o código das classes. No main(), criar dois objetos passando valores quaisquer para seus atributos através do método construtor e em seguida os imprimir.

3. Escrever um programa em C++ utilizando a classe Produto da lista de exercícios anterior como uma superclasse. Acrescentar duas subclasses de produtos quaisquer, com pelo menos dois atributos cada. Desenhar o UML incluindo também as interfaces de objetos e os construtores. Escrever o código das classes. No main(), criar dois objetos passando valores quaisquer para seus atributos através do método construtor e em seguida os imprimir.

4. Reescrever o programa em C++ do exemplo abaixo, utilizando interfaces de objetos e métodos construtores na herança. No main(), criar dois objetos passando valores quaisquer para seus atributos através do método construtor e em seguida os imprimir.



Métodos Destrutores

- Análogos aos construtores, os destrutores também são métodos membro chamados pelo sistema, só que são chamados quando o objeto sai de escopo (sua chave "}" é fechada) ou, em alocação dinâmica, tem seu ponteiro desalocado.

- Ambos (construtor e destrutor) não possuem valor de retorno.

- Não se pode chamar o destrutor, o que se faz é fornecer ao compilador o código a ser executado quando o objeto é destruído, apagado.

- Os destrutores são muito úteis para “limpar a casa” quando um objeto deixa de ser usado no escopo de um método em que foi criado, ou mesmo num bloco de código.

- Quando usados em conjunto com alocação dinâmica eles fornecem uma maneira muito prática e segura de organizar o uso do “heap” de memória.

- A importância dos destrutores em C++ é aumentada pela ausência de “garbage collection” ou coleta automática de lixo.

- A sintaxe do destrutor é simples, ele também tem o mesmo nome da classe só que precedido pelo sinal til "**~**". Seu argumento é `void` sempre:

```
~NomeDaClasse() { /* Codigo do destrutor */ }
```

Exemplo:

Arquivo "Contador.h":

```
#include <iostream.h>
class Contador
{           private:
                  int num;
            public:
                  Contador(int); //construtor
                  ~Contador(); //destrutor
                  void incrementa();
                  int getNum();
                  void setNum(int);
};
```

Arquivo "Contador.cpp":

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

      Contador::Contador(int n) { setNum(n); } //construtor
      Contador::~Contador(void) { //destrutor
            cout << "Contador destruido: " << num << endl;
      }
      void Contador::incrementa() { num++; }
      int Contador::getNum() { return num; }
      void Contador::setNum(int n) { num = n; }
//fim da implementação de Contador.h
```

Arquivo "main.cpp":

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

```
main()
{
    { //inicio de um novo bloco de codigo para chamar
            //implicitamente o destrutor ao final dele

      Contador minutos(0);
      minutos.incrementa();
      cout << "Minutos: " << minutos.getNum() << endl;

      {     //inicio de outro novo bloco de codigo

                  Contador segundos(10);
                  segundos.incrementa();
                  cout  << "Segundos: " << segundos.getNum() <<endl;
      } //fim do segundo novo bloco de codigo

      minutos.incrementa();
   } //fim do primeiro novo bloco de codigo

} //fim do main()
```

Saída do programa:

```
Minutos: 1
Segundos: 11
Contador destruido, valor: 11
Contador destruido, valor: 2
```

Considerações:

- No escopo de `main`, um novo bloco de código é declarado e dentro dele é criado o objeto `minutos`, com valor inicial `0` (zero).
- `minutos` é incrementado, agora `minutos.getNum()` retorna 1.
- O valor de `num` em `minutos` é impresso na tela.
- Um segundo novo bloco de código é criado.
- O objeto `segundos` é criado com valor inicial `10`.
- `segundos` é incrementado, agora `segundos.getNum()` retorna `11`.
- O valor de `num` em `segundos` é impresso na tela.
- **O bloco de código em que foi criado `segundos` é finalizado, então esse objeto sai de escopo, é apagado, mas antes de ser apagado o sistema chama automaticamente o destrutor.**
- `minutos` é novamente incrementado.
- **É finalizado o primeiro novo bloco de código, agora todas as variáveis declaradas dentro dele (incluindo o objeto `minutos`) saem de escopo, mas antes o sistema chama o destrutor de `minutos`.**

Métodos *Friend*

- *friend* é um modificador especial de acesso a um método ou a uma classe. Ao se declarar que um método **fora** de uma classe é **amigo** desta classe, **permite-se** a este método amigo **ler e manipular membros** (atributos e métodos) tanto `private` quanto `protected`.

- Declara-se o protótipo do método externo *dentro da classe*, com seu nome precedido da palavra `friend`.

- CUIDADO: O uso dos métodos amigos deve ser evitado sempre que possível, pois diminui a coesão e a robustez da orientação a objetos. O uso deste mecanismo representa uma quebra no conceito Encapsulamento.

Exemplo:

Arquivo "Retangulo.h":

```
#include <iostream>
using namespace std;
class Retangulo
{     private:
            int largura;
            int altura;

      public:
            int getLargura();
            void setLargura(int);
            int getAltura();
            void setAltura(int);
            int area();
            //metodo amigo (implementado fora da classe,
            //mas que possui acesso direto aos atributos do
            //objeto recebido por parâmetro)
            friend Retangulo duplicado(Retangulo);
            //retorna um objeto Retangulo, mas isso não é obrigatório
};
```

Arquivo "Retangulo.cpp":

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

      int Retangulo::getLargura() { return largura; }
      void Retangulo::setLargura(int l) { largura=l; }
      int Retangulo::getAltura() { return altura; }
      void Retangulo::setAltura(int a) { altura=a; }
      int Retangulo::area() { return(altura*largura); }
```

```
      //neste friend não é utilizado o operador "::", ou seja,
      //ele está fora da classe
      Retangulo duplicado(Retangulo r) {
            Retangulo result;
            result.largura = r.largura*2; //sem getters & setters
            result.altura = r.altura*2; //sem getters & setters
            return result;
      }
//fim da implementação de Retangulo.h
```

Arquivo "main.cpp":

```
#include <iostream>
using namespace std;
#include "Retangulo.h"
main()
{     Retangulo ret1, ret2;
      ret1.setLargura(4);
      ret1.setAltura(5);

      cout << "Area ret 1:" << ret1.area() << endl;
      ret2 = duplicado(ret1); //chamada a friend
      cout  << "Area ret 2 (duplicado de ret 1): "
                  << ret2.area() << endl;
}
```

Classes *Friend*

- Da mesma forma que se pode declarar métodos como amigos de uma determinada classe, pode-se também declarar *uma classe como amiga de outra*.
- Este artifício faz com que os membros da classe onde foi feita a declaração sejam acessíveis à classe indicada na declaração.
- Assim, **a segunda classe passa a ter possibilidade de manipulação livre dos membros da outra**, ou seja, todas as funções da primeira têm poder de acesso aos membros da segunda.

Exemplo:

Arquivo "Retangulo.h":

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

//predefinição (não é "include" pois a classe ainda não "existe")
class Quadrado;

class Retangulo
{     private:
            int largura;
            int altura;
```

```
      public:
            void setLargura(int);
            void setAltura(int);
            int area();
            void converte(Quadrado);
};
```

Arquivo "Retangulo.cpp":

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

//não é herança então não importa a ordem
#include "Retangulo.h"
#include "Quadrado.h"

      void Retangulo::setLargura(int l) { largura = l; }
      void Retangulo::setAltura(int a) { altura = a; }
      int Retangulo::area() { return (largura * altura); }
      void Retangulo::converte (Quadrado q) { //acesso
            largura = q.lado;
            altura = q.lado;
      }
//fim da implementação de Retangulo.h
```

Arquivo "Quadrado.h":

```
#include <iostream>
using namespace std;
class Quadrado
{     private:
            int lado;
      public:
            void setLado(int);
            friend class Retangulo;  //classe amiga tem acesso
};
```

Arquivo "Quadrado.cpp":

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

      void Quadrado::setLado(int s) { lado = s; }

//fim da implementação de Quadrado.h
```

Arquivo "main.cpp":

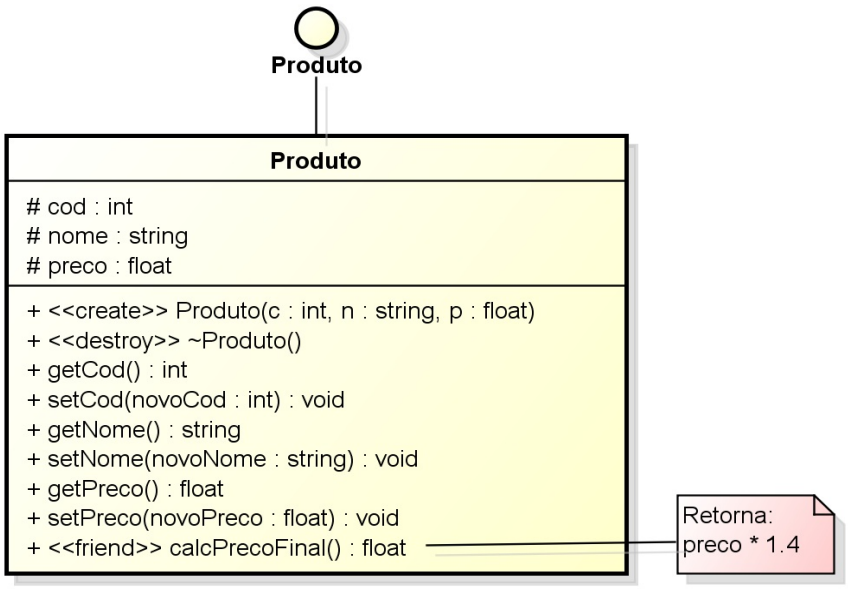
```
#include <iostream>
using namespace std;
#include "Quadrado.h"
#include "Retangulo.h"

main()
{     Quadrado quad;
      Retangulo ret;
      quad.setLado(4);
      ret.converte(quad);
      cout << "area: " << ret.area() << endl;
}
```

Membros *Protected*

- Membros declarados como `protected` (em vez de `private`) em uma superclasse podem ser acessados diretamente pela subclasse, ou seja, na herança, `protected` é igual a `public`.
- Mas para classes *fora* da hierarquia da herança, `protected` funciona como `private`.
- Em UML, o símbolo "`#`" indica um membro `protected`.

Exercícios

1. Escrever os programas em C++ da classe `Produto` abaixo. Atenção para a utilização de método destrutor (com uma mensagem qualquer de despedida do objeto) e método `friend`, além dos membros `protected`.



1. No `main()`, criar um bloco de código para limitar a existência de objetos. Criar um objeto de `Produto` dentro dele passando parâmetros para o construtor, imprimir os valores dos atributos, imprimir o preço final utilizando o método `friend`. Fechar o bloco antes de fechar o `main()` para permitir observar a mensagem de despedida do método destrutor.

2. Acrescentar três subclasses a `Produto`. Incluir métodos destrutores. No `main()`, dentro de um bloco de código limitador, criar um objeto para cada subclasse, passando valores quaisquer para os construtores. Em seguida imprimir os valores dos atributos.

3. Acrescentar a `Produto` outra subclasse, `Presunto`. Esta subclasse deve ter os atributos `marca` e `peso`. Crie uma classe `Queijo`, fora da herança, com o atributo `peso`. Declarar `Presunto` como classe `friend` de `Queijo`. Criar em `Presunto` o método `calcPeso(Queijo)`, que deve retornar o seguinte valor: 2/3 do valor do atributo `peso` do objeto de `Queijo`.

Empacotador de Preprocessador

- Inserido em cada arquivo de interface de objeto (".h") para impedir que o código da interface seja incluído no mesmo arquivo de código-fonte mais de uma vez, impedindo erros de múltiplas definições (às vezes, interfaces incluem várias vezes o mesmo arquivo de outra interface, por exemplo).
- Também é chamado de "*Include Guard*".
- Em programas maiores, outras definições e declarações também serão colocadas em arquivos de cabeçalhos. O empacotador de preprocessador impede que o código entre **#IFNDEF** ("se não definido") e **#ENDIF** seja incluído novamente.
- Se não foi incluído anteriormente, a diretiva **#DEFINE** irá incluí-lo identificando-o com um nome. Se já foi incluído (o nome é verificado), não o será novamente.

Sintaxe:

```
#ifndef NOME_H //convenção para o arquivo "Nome.h"

#define NOME_H

/*
Todo código que define o conteúdo do arquivo Nome.h: includes,
assinaturas de métodos, atributos, etc...
* /

#endif
```

Exemplo (solução do exercício anterior):

Arquivo "Produto.h":

```
#ifndef PRODUTO_H
#define PRODUTO_H

#include <iostream>
using namespace std;

class Produto
{     protected:
           int cod;
           string nome;
           float preco;

      public:
           Produto(int, string, float);
           ~Produto();
           int getCod();
           string getNome();
           float getPreco();
           void setCod(int);
           void setNome(string);
           void setPreco(float);

           friend float calcPrecoFinal(Produto);
};

#endif
```

Arquivo "Produto.cpp":

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

#include "Produto.h"
      Produto::Produto(int c, string n, float p)
      {  setCod(c);
            setNome(n);
            setPreco(p);
      }
      Produto::~Produto()
      {  printf("\nFim de objeto de Produto...\n");
      }

      int Produto::getCod() { return cod; }
      string Produto::getNome() { return nome; }
      float Produto::getPreco() { return preco; }
      void Produto::setCod(int c) { cod = c; }
      void Produto::setNome(string n) { nome = n; }
      void Produto::setPreco(float p) { preco = p; }
```

```
      float calcPrecoFinal(Produto prod)
      { return prod.preco * 1.4; }

//fim da implementação de Produto.h
```

Arquivo "Presunto.h":

```
#ifndef PRESUNTO_H
#define PRESUNTO_H

#include <iostream>
using namespace std;

class Queijo;

class Presunto:public Produto
{
      protected:
           string marca;
           float peso;

      public:
           Presunto(string, int, string, float);
           ~Presunto();
           string getMarca();
           float getPeso();
           void setMarca(string);
           void setPeso(float);

           float calcPeso(Queijo);

};

#endif
```

Arquivo "Presunto.cpp":

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

#include "Produto.h"
#include "Presunto.h"
#include "Queijo.h"

         Presunto::Presunto
                        (string m, int c, string n, float p)
                        : Produto(c, n, p)
         {  setMarca(m); //peso será setado com calcPeso
         }

         Presunto::~Presunto()
         {  printf("\nFim de objeto de Presunto...\n");
         }
```

```
         string Presunto::getMarca() { return marca; }
         float Presunto::getPeso() { return peso; }
         void Presunto::setMarca(string m) { marca= m; }
         void Presunto::setPeso(float p) { peso = p; }

         float Presunto::calcPeso(Queijo q)
         { return q.peso * 2/3; }

//fim da implementação de Presunto.h
```

Arquivo "Queijo.h":

```
#ifndef QUEIJO_H
#define QUEIJO_H

#include <iostream>
using namespace std;

class Queijo
{     protected:
           float peso;
      public:
           Queijo(float);
           ~Queijo();
           float getPeso();
           void setPeso(float);
           friend class Presunto; //classe amiga
};

#endif
```

Arquivo "Queijo.cpp":

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

#include "Queijo.h"

         Queijo::Queijo(float pe)
         {  setPeso(pe);
         }

         Queijo::~Queijo()
         {  printf("\nFim de objeto de Presunto...\n");
         }

         float Queijo::getPeso() { return peso; }
         void Queijo::setPeso(float p) { peso = p; }

//fim da implementação de Presunto.h
```

Arquivo "main.cpp":

```
#include <iostream>
using namespace std;
#include "Produto.h"
#include "Presunto.h"
#include "Queijo.h"

main()
{
   { //bloco limitador 1

            Produto p1 (101, "Camisa", 35.5);

            cout  << "\n\nProduto: "
                        << p1.getCod() << " - "
                        << p1.getNome() << " - R$ "
                        << calcPrecoFinal(p1)
                        << endl;
    }

    { //bloco limitador 2

            Queijo q1 (0.5);
            Presunto pr1 ("Especial", 103, "Sem Gordura", 9.5);

            pr1.setPeso( pr1.calcPeso( q1 ));

            cout  << "\n\nProduto: "
                        << pr1.getCod() << " - "
                        << pr1.getMarca() << " - "
                        << pr1.getNome() << " - "
                        << pr1.getPeso() << " Kg - R$ "
                        << calcPrecoFinal(pr1)
                        << endl;

    }

}
```

Introdução a Ponteiros

- Um ponteiro é uma variável que contém o endereço de memória de outra variável.
- Os ponteiros devem ser inicializados quando forem declarados ou em uma atribuição.
- O ponteiro pode ser inicializado como 0 (zero) ou NULL ou como um endereço.
- O ponteiro com valor NULL não aponta, é ainda um ponteiro nulo.

Sintaxe: O operador ***** é usado na declaração do ponteiro:

```
tipo *nome-da-variavel;
```

Exemplos de Declarações:

```
int *p;
float *a;
double *q;
```

Operadores:

& → é um operador unário que contém a *referência* da variável. É usado para se obter o *endereço* da variável.

Ex: `a = &m;  //atribui à "a" o endereço de "m"`

***** → é um operador unário que contém o *conteúdo* da variável que aponta. Quando se quer obter o *valor* da variável pra onde o ponteiro aponta, usa-se "*" (* = valor da variável apontada).

Exemplo:

```
int a=8;
int *c;
int b;
c = &a; //"c" aponta para o ENDEREÇO de "a"
b= *c;  //"b" recebe o VALOR do APONTADO POR "c"
```

Memória:

```
_____________________
      c = 108       |100
____________________|
                    |104
____________________|
      a = 8         |108  ← c
____________________|
                    |112
____________________|
      b = 8         |116
____________________|
```

]

Exemplo:

```
#include <iostream>
using namespace std;
main()
{     int y;
      int *p;

      p = &y;  //recebe o endereço de memoria de y
      *p = 30; //atribui 30 ao apontado por p (y=30)

      cout  << "o endereco de y eh:" << &y //endereço
                  << "\no valor de p eh" << p
                  << endl;

      cout  << "o valor de y eh:" << y
                  << "\no valor de *p eh:" << *p //apontado
                  << endl;

}
```

Passagem de Parâmetros por Referência

- Existem duas formas de passar parâmetros para um método: por valor ou por referência.
- Quando o argumento é passado por valor, uma cópia do valor do argumento é armazenada na memória e passada em seguida para o método chamado.
- Em C++, a passagem de parâmetros por referência pode ser realizada de duas formas: usando a referência (endereço) ou usando o ponteiro.
- Usa-se o operador **&**(endereço) no protótipo do método e também na assinatura do método, na implementação. Vejamos no exemplo a seguir.

Exemplo:

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

void reajustar(double &preco, double &reajuste); //ref

main()
{     double valorPreco, valorReajuste;
      do {  cout << "Insira o preco atual:";
            cin >> valorPreco;
            reajustar(valorPreco, valorReajuste);

                  cout  << "Novo preco:" << valorPreco << endl;
            cout  << "aumento foi de:" << valorReajuste
                              << endl;
  } while ( valorPreco != 0 );
}

//recebe os endereços das variáveis
void reajustar(double &preco, double &reajuste)
{
      reajuste = preco * 0.25;
      preco = preco * 1.25;
}
```

Passagem de Parâmetros para Ponteiros

- Utiliza-se o operador ***** no protótipo e na assinatura do método.
- Na chamada ao método, é preciso usar o operador **&** para passar para o método o *endereço* de cada variável.
- Dentro do método, o operador * denota o *conteúdo* da área apontada por cada ponteiro usado.

Exemplo 1:

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

//protótipo do método usando ponteiros
void reajustar(double *ppreco, double *preajuste);
```

```
main()
{     double valorPreco, valorReajuste;

      do {  cout << "Insira o preco atual:";
            cin >> valorPreco;

                  //chamada
            reajustar(&valorPreco, &valorReajuste);

                  cout  << "Novo preco:" << valorPreco << endl;
            cout  << "aumento foi de:" << valorReajuste
                              << endl;
      } while ( valorPreco != 0 );
}

//para ponteiros
void reajustar(double *ppreco, double *preajuste)
{
      *preajuste = *ppreco * 0.25; //valores
      *ppreco = *ppreco * 1.25;
}
```

Exemplo 2:

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

void troca(int *px, int *py)
{
  int aux;
  aux = *px;
  *px = *py;
  *py = aux;
}

main()
{  int x=10, y=20;
   troca(&x, &y);
   cout << "x = " << x << "\ny = " << y << endl;
}
```

Exemplo 3:

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

void cubo(int *p);

main()
{     int num=5;
      cout << "\nValor inicial: " << num << endl;
      cubo(&num);
      cout << "\nNovo valor: " << num << endl;
}

void cubo(int *p)
{
      *p = (*p) * (*p) * (*p);
}
```

Ponteiro não Const para Dados não Const

- Há quatro maneiras de passar um ponteiro para um método: a primeira é passar um ponteiro não constante que aponta para dados não constantes.
- Os dados podem ser modificados pelo ponteiro "desreferenciado" (usando *) e o ponteiro pode ser modificado para apontar para outros dados.
- A declaração não inclui o modificador `const`. Este modificador será comentado mais a frente.
- Este ponteiro, por exemplo, pode ser utilizado para receber uma string terminada pelo caractere nulo '`\0`' em um método que altera o valor do ponteiro para processar cada caractere da string.

Exemplo:

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

void converte(char *);
main()
{     //vetor de caracteres frase do tipo char
      char frase[]="programa orientado a objetos";
      cout << "frase antes da conversao" << frase << endl;

      converte(frase); //metodo altera o conteudo
      cout << "frase apos a conversao:" << frase << endl;
}

//recebe ponteiro não const para objeto não const
void converte(char *s) {
      while(*s!='\0') {

        if( islower(*s) )  //testa o que é minusculo de a a z
            *s = toupper(*s);  //transforma em maiuscula

        s++;  //anda com o ponteiro dentro do vetor
      }
}
```

- A função `islower()` recebe um argumento caractere e retorna verdadeiro se o caractere for uma letra minúscula ou falso, caso contrário.
- Os caracteres a-z são convertidos para maiúsculas correspondentes pela função `toupper()`.
- O ponteiro `s` é incrementado em 1 (só é possível porque `s` não foi declarado como `const`).
- Quando o operador ++ é aplicado a um ponteiro que aponta pra um vetor, o endereço de memória armazenado no ponteiro é modificado pra apontar para o próximo elemento do vetor (aritmética de ponteiros).

Ponteiro não Const para Dados Const

- É um ponteiro que pode ser modificado para apontar para qualquer item de dados do tipo apropriado, mas os dados para os quais ele aponta **não podem ser modificados por esse ponteiro**.

Exemplo:

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

//ponteiro apenas de leitura dos dados
void imprimeCaracteres(const char *);

main()
{     //dados não podem ser modificados (const)
      const char frase[]="poo – programacao 2";
      cout << "a string eh:" << "\n";
      imprimeCaracteres(frase);
      cout << endl;
}

void imprimeCaracteres(const char *s) {
 //*s aponta para um vetor
 //o valor inicial está no índice 0 do vetor
 //(vale pra dados não constantes também)
 for( ; *s != '\0'; s++)
   cout << *s;
}
```

- O nome do vetor `frase` é um ponteiro para o primeiro caractere no vetor.
- `s` é um ponteiro para uma constante do tipo `char`.
- `s` pode ser modificado (pode apontar pra outro endereço), mas não pode modificar o caractere para o qual ele aponta, isto é, `s` é um ponteiro de leitura.

Exemplo:

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

void funcao(const int *);

main() {
 int y;

 funcao(&y);
}

void funcao(const int *x)
{
      *x = 100;
      //ERRO: não se pode alterar um valor constante
}
```

Ponteiro Const para Dados não Const

- É um **ponteiro que sempre aponta para a mesma posição de memória**. Os dados nessa posição **podem ser modificados** pelo ponteiro.
- É o padrão para declarações de vetores. **Um nome de um vetor é um ponteiro constante para o começo do vetor**. Todos os dados podem ser modificados e acessados utilizando o nome do vetor.
- Os ponteiros que são declarados como `const` devem ser inicializados quando são declarados.

Exemplo:

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

main()
{

      int x, y;

      int *const p = &x;
      //inicializacao do ponteiro constante para um inteiro
      //nao constante que pode ser modificado por p,
      //mas p sempre aponta para a mesma posição de memoria

      *p = 10;
      //possivel, pois o valor da variavel x, para onde
      //o ponteiro p aponta, pode ser alterado

      p = &y;
      //ERRO: não pode alterar a posição para onde p aponta

}
```

Ponteiro Const para Dados Const

- É um **ponteiro que sempre aponta para a mesma posição de memória** e os dados nessa posição **não podem ser modificados** pelo ponteiro.
- Essa é a maneira como um vetor deve ser passado para um método que somente lê o vetor e não o modifica.

Exemplo:

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

main()
{     int x=5, y;
      const int *const p = &x;
      //"int * const p" significa "*p constante",
      //ou seja, apontador constante para valores int.
      //"const int * const" é como "const *p constante",
      //ou seja, apontador constante para um int constante

      cout << *p << endl;
```

```
      *p = 10;
      //ERRO: não é permitido mudar o valor de *p,
      //pois *p é constante e aponta pra x (que neste
      //contexto tem valor constante)

      p = &y;
      //ERRO: não é permitido mudar a posição de memoria
      //pra onde p aponta, pois p foi declarado constante

}
```

Exemplo:

```
#include <iostream>
#include <iomanip>  //manipulação de saída (formatação)
using namespace std;

void selectionSort(int *const, const int);
void troca(int *const,int *const);

main()
{     int i;
      const int tam = 10;
      int vetor[10] = {2,8,4,6,10,12,20,22,16,18};

      cout << "dados não ordenados:" << endl;
      for(i=0; i<tam; i++)
      cout << setw(4) << vetor[i];

      selectionSort(vetor, tam);
      cout << "\n dados ordenados crescentemente:\n";
      for(i=0;i<tam;i++)
            cout << setw(4) << vetor[i];
      cout<<endl;
}

void selectionSort(int *const v, const int n)
{  int menor;
      for(int j=0; j < n-1; j++) {
            menor = j;
            for(int ind = j+1; ind<n; ind++)
      {     if( v[ind] < v[menor] )
                  {     menor = ind;
            }
            }
            troca(&v[j], &v[menor]);
      }
}

void troca(int *const pt1, int *const pt2)
{     int aux = *pt1;
      *pt1 = *pt2;
      *pt2 = aux;
}
```

Alocação Dinâmica de Memória com New e Delete

- O operador new cria uma nova variável dinâmica de um tipo especificado e retorna um ponteiro que aponta para essa nova variável.

Exemplo: (criar uma nova variável dinâmica de "MeuTipo" e fazer com que a variável ponteiro "p" aponte para essa nova variável)

```
MeuTipo *p;
p = new MeuTipo;
```

- Se o tipo for uma classe, o construtor padrão (sem parâmetros) é chamado para a variável dinâmica recém criada.

- Ou, pode-se também chamar um construtor diferente incluindo argumentos, da seguinte forma:

```
MeuTipo *p;
p = new MeuTipo (32, 17);
```

- Uma notação semelhante pode ser usada para tipos primitivos:

```
int *n;
n = new int(17);
//inicializa *n como 17
//(valor do apontado por n é 17)
```

- Em resumo, `new` cria automaticamente um objeto de tamanho apropriado (número de bytes), chama o construtor do objeto e retorna um ponteiro para o tipo correto.

- Caso não haja espaço disponível na memoria, o operador `new` retorna 0 (zero).

- O operador `delete` é usado para destruir o objeto e liberar a memória ocupada por ele:

Exemplo:

```
int *p;
p = new int;  //aloca memoria
delete p;  //libera a memoria referenciada por p
```

Exemplo:

```
int *vetor = new int[5];
//aloca memoria para 5 elementos inteiros
delete[] vetor;  //o vetor eh apagado
```

Exemplo:

```
#include <iostream>
using namespace std;
main()
{     int *p, *vetor;
      p = new int;
      *p = 10;  //apontado por p recebe 10

      //*p = 10 e p = endereço do apontado por p
      cout  << "valor de *p = " << *p
                  << "\nvalor de p = " << p << endl;

      //libera a area apontada por p
      delete p;
      cout  << "\nApos delete p, valor de p = "
                  << p << endl;
      p = NULL;  //equivale a p = 0
      cout << "\nvalor de p = " << p <<endl;

      //aloca vetor
      vetor = new int[5];
      cout << "\n";

      //preencher vetor
      for(int i=0; i<5; i++) {
            cout << "digite um numero inteiro:";
            cin >> vetor[i];
      }

      cout << "\n imprimindo vetor:\n";
      for(int i=0; i<5; i++)
            cout << vetor[i] << "\t";


      cout << "\n imprimindo vetor com outra notacao\n";
      for(int i=0; i<5; i++)
            cout << *(vetor+i) << "\t";

      cout << "\n";

      vetor++;
      cout  << "apos incrementar ponteiro vetor, *vetor = "
                  << *vetor << endl;

      vetor--;
      cout  << "apos decrementar ponteiro vetor, *vetor = "
                  << *vetor << endl;

      delete[] vetor;
      vetor = 0;  //encerra o ponteiro vetor

}
```

Objetos e Métodos Const

- O modificador `const` impede modificações no objeto. Neste caso, os métodos membro também devem ser declarados como `const`, mesmo os *getters* que não modificam valores. Os modos de qualificação atuam em conjunto, para assegurar que um objeto `const` não será modificado. C++ só permite que sejam chamados para este objeto métodos membro qualificadas como `const`.

- `const` pode qualificar:

  - um **parâmetro de método** (assegurando que este não será modificado),

  - um **método membro** (assegurando que este não modifica os dados membro de sua classe), ou

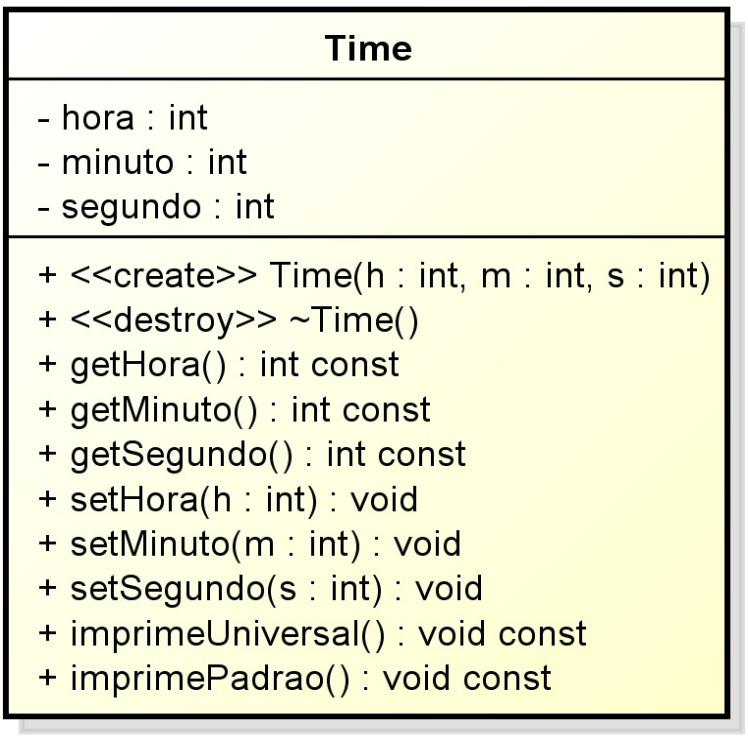
  - uma instância de objeto (assegurando que este não será modificado).

- É possível criar sobrecargas (versões alternativas com listas de parâmetros diferentes) *não const* dos métodos, o funcionamento será automático dependendo dos argumentos passados nas chamadas.

- O modificador `const` não é usado no construtor (nem no destrutor), o objeto se torna `const` após o fim da execução do construtor.

- Um construtor deve ter permissão de modificar um objeto para que o objeto possa ser inicializado adequadamente.

Exemplo:



Arquivo "`Time.h`":

```
#include <iostream>
using namespace std;
class Time {
 private:
  int hora;
  int minuto;
  int segundo;
 public:
  Time(int,int,int);
  ~Time();
  //normalmente getters são declarados como const
  int getHora() const;
  int getMinuto() const;
  int getSegundo() const;
  void setHora(int);
  void setMinuto(int);
  void setSegundo(int);
  void imprimeUniversal() const;
  void imprimePadrao() const;
};
```

Arquivo "`Time.cpp`":

```
#include <iostream>
#include <iomanip>
#include "Time.h"
using namespace std;
      Time::Time(int h,int m, int s) {
       setHora(h);
       setMinuto(m);
       setSegundo(s);
      }
      Time::~Time() { cout << "Fim objeto Time." << endl; }

      void Time::setHora(int h) {
       hora = ( h >= 0 && h < 24 ) ? h : 0; //valida horas
      }
      void Time::setMinuto(int m) {
       minuto = ( m >= 0 && m < 60 ) ? m : 0; //valida min
      }
      void Time::setSegundo(int s) {
       segundo = ( s >= 0 && s < 60 ) ? s : 0; //valida seg
      }
      int Time::getHora() const {  return hora; }
      int Time::getMinuto() const {  return minuto; }
      int Time::getSegundo() const {  return segundo; }
      void Time::imprimeUniversal() const {
       cout << setfill('0') << setw(2) << hora
                  << ":" << setw(2) << minuto
                  << ":" << setw(2) << segundo << endl;
      }
      void Time::imprimePadrao() const {
       cout << ((hora == 0 || hora == 12) ? 12 : hora % 12)
                  << ":" << setfill('0') << setw(2) << minuto
                  << ":" << setw(2) << segundo
                  << (hora < 12 ? "AM" : "PM") << endl;
      }
```

Testes: Para os métodos `main()` abaixo, verifique que tipo de método funciona com que tipo de objeto (`const` ou não `const`).

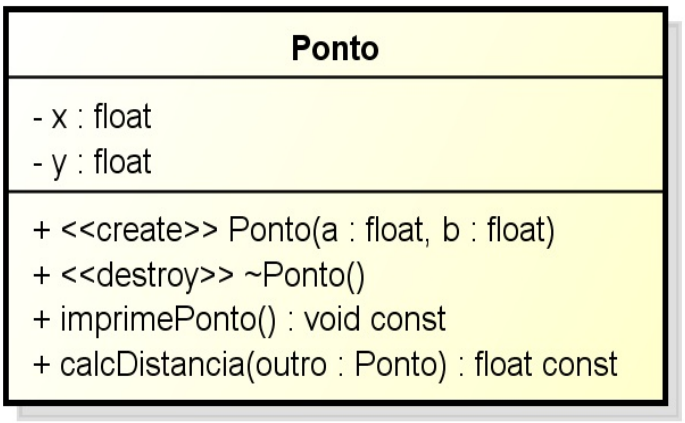
Arquivo "main.cpp": (versão 1)
```
#include <iostream>
using namespace std;
#include "Time.h"
main() {
  Time noite(18,0,0);  //objeto não constante
  const Time dia(5,45,0);  //objeto constante

  //objeto constante e métodos constantes
  cout << dia.getHora() << endl;
  dia.imprimeUniversal();
  dia.imprimePadrao();

}
```

Arquivo "main.cpp": (versão 2)
```
#include <iostream>
using namespace std;
#include "Time.h"
main() {
  Time noite(18,0,0);  //objeto não constante
  const Time dia(5,45,0);  //objeto constante

  //objeto não constante
  noite.imprimePadrao(); //método const
  noite.imprimeUniversal(); //método const
  noite.setHora(20); //método não const
  noite.imprimeUniversal(); //método const
  noite.imprimePadrao(); //método const
}
```

Arquivo "main.cpp": (versão 3)
```
#include <iostream>
using namespace std;
#include "Time.h"
main() {
  Time noite(18,0,0);  //objeto não constante
  const Time dia(5,45,0);  //objeto constante

  //objeto não constante
  noite.setHora(20); //método não const
  cout << noite.getHora() << endl; //método const
  cout << noite.getMinuto() << endl; //método const

  //objeto constante não pode ser modificado
  //(comente para compilar)
  dia.setHora(7);
}
```

Exercícios

1. Escrever um programa C++ para a classe `Ponto` abaixo, incluindo os *getters & setters*:



2. No `main()`, declarar um objeto constante origem com coordenadas `0` e `0` (zero e zero). Declarar um segundo objeto com coordenadas quaisquer. Mostrar os pontos usando o método correspondente. Mostrar a distância entre eles.

   Observação 1: Executar testes declarando objetos constantes e outros não constantes para verificar as chamadas aos métodos, inclusive os *getters & setters* (declarados como constantes em alguns testes e como não constantes em outros testes).

   Observação 2: Comentar o código conforme os testes forem sendo executados. Exemplo:

```
//p1.setX(10); //ERRO: p1 é objeto const
```

Inicializadores

– Todos os atributos *podem* ser inicializados conforme a sintaxe abaixo, mas os declarados como `const` *obrigatoriamente* terão que ser inicializados dessa forma:

Exemplo:

```
//os inicializadores dos atributos const nome e preco
//com valores n e pr
Produto::Produto(string n, double pr)
                  : nome(n), preco(pr)
{
      //vazio
}
```

Exercícios

1. Escrever um programa C++ para a classe `Ponto` do exercício anterior, incluindo os *getters & setters*, utilizando atributos `const` e *inicializadores* no construtor. Fazer testes com o `main()`, da mesma forma que no exercício 2 anterior.

2. Escrever um programa C++ para a classe `Time` do exemplo anterior, utilizando atributos `const` e *inicializadores* no construtor. Fazer testes com o `main()`, da mesma forma que no exercício 2 anterior.

Polimorfismo

- A palavra "polimorfismo" em POO está relacionada à várias formas de um mesmo método. Na herança, subclasses diferentes podem apresentar implementações diferentes de um método de mesma assinatura, retornando resultados adequados a cada uma, mas permitindo a mesma chamada.

- Técnica muito utilizada para permitir percorrer coleções de objetos e executar uma mesma chamada de método. Nestes casos é necessário utilizar métodos virtuais e sobrecarga de métodos, apresentados mais tarde.

- Cada objeto executa sua versão do método ao serem requisitados em um laço com uma mesma chamada de método, sem ser necessário testar se cada objeto pertence a uma determinada classe.

- Por exemplo, um certo método "`calcImposto()`" pode ter implementações diferentes para diferentes objetos de subclasses de uma certa classe "`Produto`".

Exemplo:

Arquivo "Poligono.h":
```
#ifndef POLIGONO_H
#define POLIGONO_H

#include <iostream>
using namespace std;

class Poligono {
 protected:
  double largura;
  double altura;
 public:
  void setLargura(double);
  void setAltura(double);
};

#endif
```

Arquivo "Poligono.cpp":
```
#include <iostream>
using namespace std;

#include "Poligono.h"
```

```
    void Poligono::setLargura(double l) {
     largura=l;
    }
    void Poligono::setAltura(double a) {
     altura=a;
    }
```

Arquivo “Retangulo.h”:

```
#ifndef RETANGULO_H
#define RETANGULO_H

#include <iostream>
using namespace std;

#include "Poligono.h"

class Retangulo : public Poligono {
 public:
  double area();
};

#endif
```

Arquivo “Retangulo.cpp”:

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

#include "Retangulo.h"

    double Retangulo::area() {
     return (altura*largura);
    }
```

Arquivo “Triangulo.h”:

```
#ifndef TRIANGULO_H
#define TRIANGULO_H

#include <iostream>
using namespace std;

#include "Poligono.h"

class Triangulo : public Poligono {
 public:
  double area();
};

#endif
```

Arquivo “Triangulo.cpp”:

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

#include "Triangulo.h"

    double Triangulo::area() {
      return (largura*altura/2);
    }
```

Arquivo “main.cpp”:

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

```
#include "Retangulo.h"
#include "Triangulo.h"
main()
{//objetos
 Retangulo ret;
 Triangulo tri;
 //ponteiro da superclasse para polimorfismo
 Poligono *pol1=&ret;
 Poligono *pol2=&tri;
 //operador seta
 pol1->setLargura(4);
 pol1->setAltura(5);
 pol2->setLargura(4);
 pol2->setAltura(5);
 //mostra polimorfismo area()
 cout << "area do retangulo:" << ret.area() << endl;
 cout << "area do triangulo:" << tri.area() << endl;
}
```

Métodos Virtuais

- Métodos virtuais são declarados colocando-se a palavra `virtual` no início do protótipo do método na superclasse.
- Quando o compilador encontra uma instrução com chamada a um método virtual, ele não tem como identificar qual é o método associado em tempo de *compilação*. Por esta razão, a instrução é avaliada em tempo de *execução*, quando é possível identificar que tipo de objeto é apontado pelo ponteiro.
- Esta situação é chamada **vinculação dinâmica tardia** e seu uso permite acomodar o surgimento de novas classes no sistema.

Exemplo 1:

Arquivo "Poligono.h":

```
#ifndef POLIGONO_H
#define POLIGONO_H

class Poligono {
 protected:
  double largura;
  double altura;
 public:
  void setLargura(double);
  void setAltura(double);
  //todo metodo virtual é criado na superclasse
  virtual double area();
};

#endif
```

Arquivo "Poligono.cpp":

```
#include "Poligono.h"

    void Poligono::setLargura(double l) {
     largura=l;
    }
```

```
    void Poligono::setAltura(double a) {
     altura=a;
    }
       double Poligono::area() { //metodo virtual
     return 0;
    }
```

Arquivo “Retangulo.h”:

```
#ifndef RETANGULO_H
#define RETANGULO_H

#include "Poligono.h"

class Retangulo : public Poligono {
 public:
  double area();
};

#endif
```

Arquivo “Retangulo.cpp”:

```
#include "Retangulo.h"

    double Retangulo::area() {
     return (altura*largura);
    }
```

Arquivo “Triangulo.h”:

```
#ifndef TRIANGULO_H
#define TRIANGULO_H

#include "Poligono.h"
class Triangulo : public Poligono {
 public:
  double area();
};

#endif
```

Arquivo “Triangulo.cpp”:

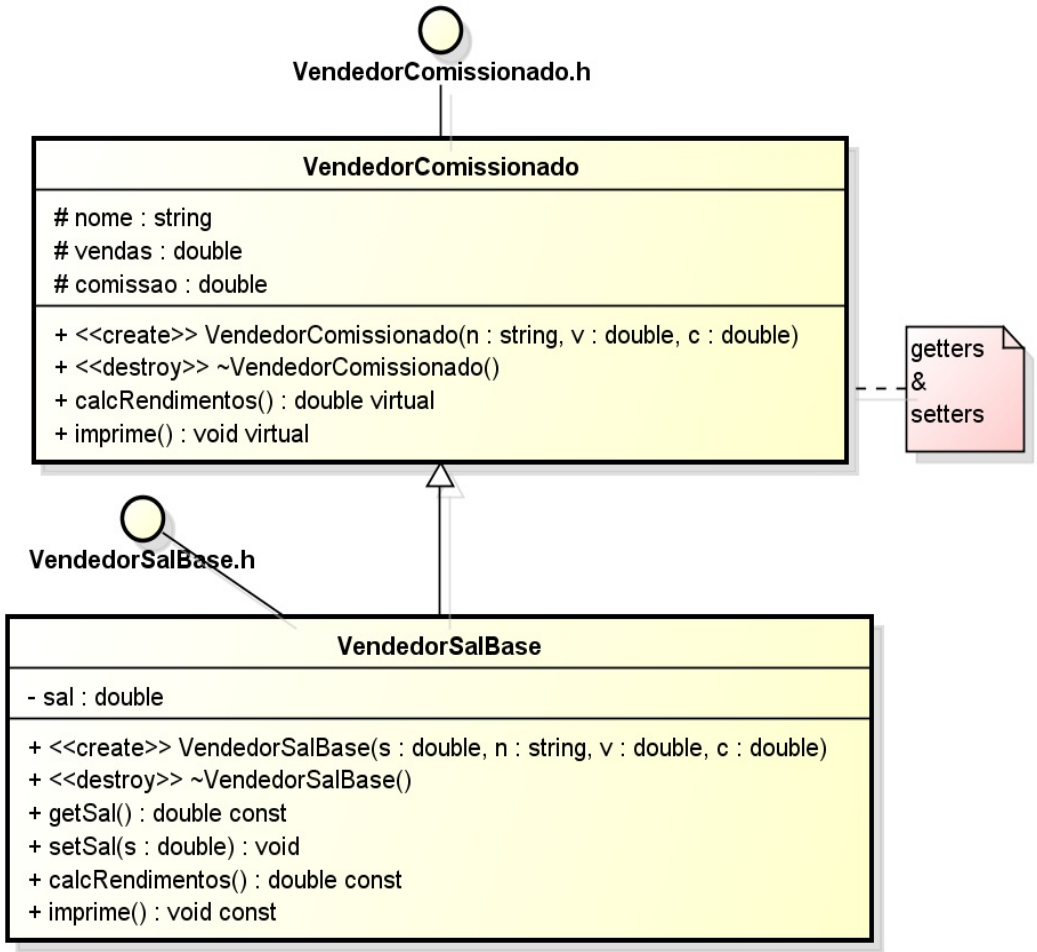
```
#include "Triangulo.h"

    double Triangulo::area() {
      return (largura*altura/2);
    }
```

Arquivo “main.cpp”:

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

#include "Poligono.h"
#include "Retangulo.h"
#include "Triangulo.h"

main()
{
 //objetos
 Retangulo ret;
 Triangulo tri;
 Poligono pol; //para mostrar a virtual
```

```
 //ponteiro da superclasse para polimorfismo
 Poligono *pol1=&ret;
 Poligono *pol2=&tri;
 Poligono *pol3=&pol;

 //operador seta
 pol1->setLargura(4);
 pol1->setAltura(5);
 pol2->setLargura(4);
 pol2->setAltura(5);
 pol3->setLargura(4);
 pol3->setAltura(5);

 //possivel usar o operador seta, pois
 //um objeto Poligono "enxerga" o método area()
 cout << "area do retangulo:" << pol1->area() << endl;
 cout << "area do triangulo:" << pol2->area() << endl;
 cout << "area do poligono:" << pol3->area() << endl;
}
```

Exemplo 2:



Arquivo "`VendedorComissionado.h`":

```
#ifndef VENDEDORCOMISSIONADO_H
#define VENDEDORCOMISSIONADO_H

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

```
class VendedorComissionado {
 protected:
  string nome;
  double vendas;
  double comissao;
 public:
  VendedorComissionado(const string &, double, double);
  ~VendedorComissionado();
  void setNome(const string &);
  string getNome() const;
  void setVendas(double);
  double getVendas() const;
  void setComissao(double);
  double getComissao() const;
  virtual double calcRendimentos() const;
  virtual void imprime() const;
};
#endif
```

Arquivo "VendedorComissionado.cpp":

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

VendedorComissionado::VendedorComissionado
      (const string &n, double v, double c)
      {
            nome = n;   //constantes podem ser atribuidas diretamente
            setVendas(v);
            setComissao(c);
      }
VendedorComissionado::~VendedorComissionado() { }

void VendedorComissionado::setNome(const string &n) { nome=n; }

string VendedorComissionado::getNome() const { return nome; }

void VendedorComissionado::setVendas(double v) {
  vendas = (v < 0.0) ? 0.0 : v;
}

double VendedorComissionado::getVendas() const { return vendas; }

void VendedorComissionado::setComissao(double c) {
  comissao = (c > 0.0 && c < 1.0) ? c : 0.0;
}

double VendedorComissionado::getComissao() const { return comissao; }

double VendedorComissionado::calcRendimentos() const {
  return (comissao * vendas);
}

void VendedorComissionado::imprime() const {
  cout << "\nComissionado: " << nome
       << "\nVendas: " << vendas
       << "\nComissao: " << comissao << endl;
}
```

Arquivo "VendedorSalBase.h":
```
#ifndef VENDEDORSALBASE_H
#define VENDEDORSALBASE_H

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

class VendedorSalBase : public VendedorComissionado {
 private:
  double sal;
 public:
  VendedorSalBase(double, const string &, double, double);
  ~VendedorSalBase();
  void setSal(double);
  double getSal() const;
  double calcRendimentos() const;
  void imprime() const;
};

#endif
```

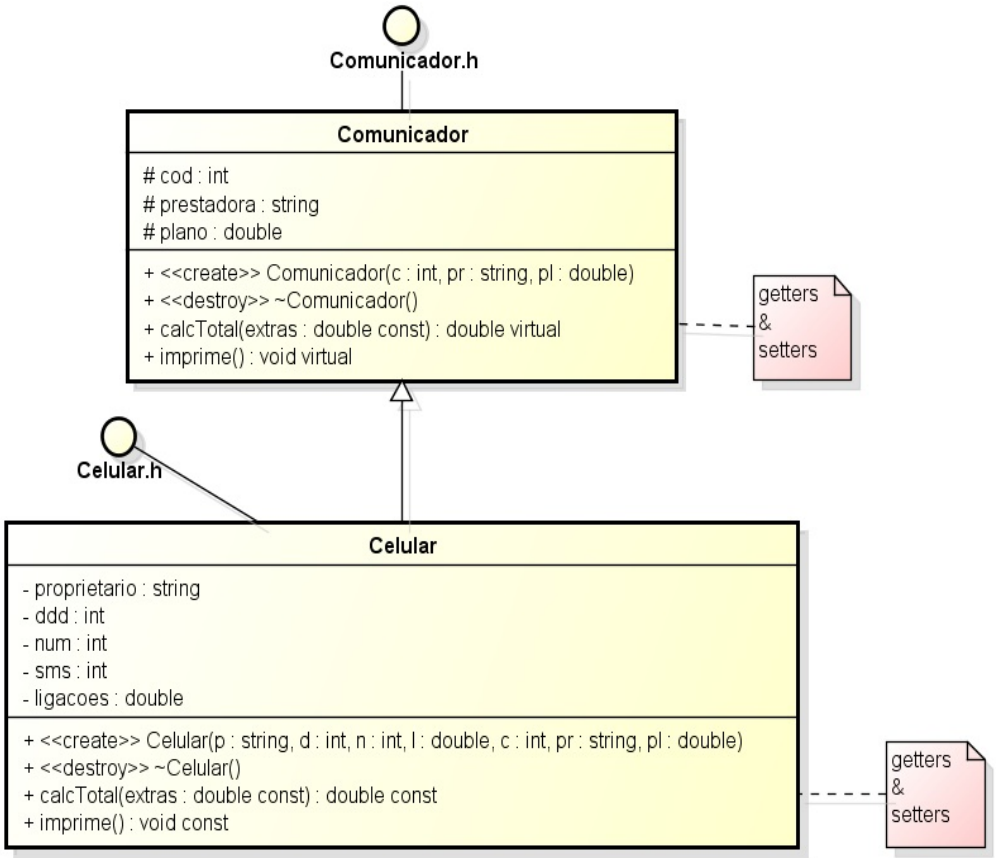
Arquivo "VendedorSalBase.cpp":
```
#include <iostream>
#include <string>
using namespace std;
#include "VendedorSalBase.h"

VendedorSalBase::VendedorSalBase
      (double s, const string &n, double v, double c)
      :VendedorComissionado(n, v, c)
      {  setSal(s);
      }
VendedorSalBase::~VendedorSalBase() { }
void VendedorSalBase::setSal(double s) {
  sal = (s < 0.0) ? 0.0 : s;
}
double VendedorSalBase::getSal() const {
  return sal;
}
double VendedorSalBase::calcRendimentos() const {
  return (sal + (comissao * vendas));
}
void VendedorSalBase::imprime() const {
  cout << "\nVendedor com salario base: " << nome
       << "\nVendas: " << vendas
       << "\nComissao: " << comissao
       << "\nSalario base:" << sal << endl;
}
```

Arquivo "main.cpp":
```
#include <iostream>
#include <string>
using namespace std;
#include "VendedorComissionado.h"
#include "VendedorSalBase.h"
main()
{ VendedorComissionado vcom("ana", 3500, 0.07);
  VendedorSalBase vsal(5000, "joao", 300, 0.04);
    //cria um ponteiro para a superclasse e o inicializa
  VendedorComissionado *vcp = 0;
  //cria um ponteiro para a subclasse
```

```
  VendedorSalBase *vsp = 0;

  cout<<"Vinculacao Estatica:\n";
  vcom.imprime(); //gera a saída utilizando vinculação estatica
  vsal.imprime(); //gera saída usando vinculação estatica
  cout<<"----------------------------------------";
  cout<<"\nVinculacao Dinamica:\n";
  //aponta o ponteiro da superclasse para o objeto da superclasse
  //(aponta para o endereço de vcom)
  vcp = &vcom;
  vcp->imprime(); //gera a saida utilizando vinculação dinamica
  //aponta o ponteiro da subclasse para o objeto da subclasse
  vsp = &vsal;
  vsp->imprime();
  cout<<"----------------------------------------";
  cout<<"\nPolimorfismo:\n";
  //aponta o ponteiro da superclasse para o objeto da subclasse
  vcp = &vsal;
  vcp->imprime(); //polimorfismo: chama imprime() da subclasse
  cout<<endl;
}
```

Exercícios

1. Escrever um programa C++ para as classes `Comunicador` e `Celular` abaixo, incluindo os *getters & setters*, utilizando atributos `const` e *inicializadores* no construtor. Fazer testes com o main(), da mesma forma que no exemplo anterior.



2. Acrescentar um método `friend` para a classe `Celular` do exemplo anterior. Acrescentar uma classe `Modem` que tenha como atributo seu próprio consumo e relacionar como `friend` de `Celular`, ou `Celular` como `friend` de `Modem`. No cálculo do total, acrescente o consumo de `Modem`. Fazer testes com o `main()`, da mesma forma que nos exemplos que utilizam `friend`.

Sobrecarga de Métodos

- Vários métodos podem ser redefinidos com o mesmo nome, mas com listas *diferentes* de argumentos (em tipo, número e ordem).
- Quando um método sobrecarregado é chamado, o compilador seleciona o método adequado examinando os parâmetros de acordo com a quantidade, os tipos e a ordem passados na chamada.
- A ideia é criar opções de chamadas para um mesmo método, passando argumentos diferentes para executar tarefas semelhantes, mas com dados diferentes.

Exemplo:

```
//soma de dois inteiros
int soma(int x, int y) { return x + y; }

//soma de três inteiros
int soma(int x, int y, int z) { return x + y + z; }

main()
{     cout << soma (7, 8) << endl; //chama o primeiro método
      cout << soma (7, 8, 5) << endl; //chama o segundo método
}
```

Sobreposição de Métodos Herdados

- Subclasses podem redefinir (sobrepor, sobrescrever) métodos herdados reescrevendo-os com o mesmo nome e listas *iguais* de argumentos (em tipo, número e ordem).
- A ideia é melhorar ou tornar mais específico o comportamento dos objetos da subclasse pra aquele determinado método.
- Obs.: Quando as listas de argumentos são diferentes, caracteriza-se como sobrecarga do método herdado.
- Sobrecarga (*overloading*) e Sobreposição (*overriding*) são classificados como Polimorfismo.

Exemplo:

```
//Arquivo: “Ponto.h”
class Ponto
{ private:
           float x;
           float y;
  public:
            Ponto(float, float);
            float getX();
            float getY();
            void setX(float);
            void setY(float);
            void mostra();  //será redefinido
};

//Arquivo: “Ponto.cpp”
#include <iostream>
#include "Ponto.h"
using namespace std;
Ponto::Ponto(float a, float b)
{ setX(a);
  setY(b);
}
float Ponto::getX() { return x; }
float Ponto::getY() { return y; }
void Ponto::setX(float a) { x = a; }
void Ponto::setY(float b) { y = b; }
void Ponto::mostra()
{ cout << "(" << x << "," << y << ")" << endl;
}

//Arquivo: “Reflete.h”
class Reflete:public Ponto
{ public:
          Reflete(float, float);
          void refletePonto();
          void mostra(); //sobreposicao
};

//Arquivo: “Reflete.cpp”
#include <iostream>
#include "Ponto.h"
#include "Reflete.h"
using namespace std;
Reflete::Reflete(float a, float b):Ponto(a,b) { }
void Reflete::refletePonto()
{ setX( getX() + 1 ); //apenas soma 1
  setY( getY() + 1 );
}
void Reflete::mostra() //sobreposicao
{ cout << "X:" << getX(); //muda o formato
  cout << " Y:" << getY() << endl;
}


//Arquivo: “Move.h”
class Move:public Ponto //sem sobreposicao de "mostra()"
{ public:
          Move(float, float);
          void movePonto(float, float);
};
```

```
//Arquivo: “Move.cpp”
#include <iostream>
#include "Ponto.h"
#include "Move.h"

Move::Move(float a, float b):Ponto(a,b) { }

void Move::movePonto(float m1, float m2)
{ setX(getX() + m1);
  setY(getY() + m2);
}

//Arquivo: “main.cpp”
#include <iostream>
#include "Ponto.h"
#include "Reflete.h"
#include "Move.h"
using namespace std;
main()
{
 Reflete p1(5.74, 6.66);
 p1.refletePonto();
 cout << "p1: ";
 p1.mostra();

 Move p2(2.0,3.0);
 p2.movePonto(0.66, 0.88);
 cout << "p2: ";
 p2.mostra();
}
```

Exercício

1. Escrever programas C++, alterando as classes dos exercícios anteriores, incluindo métodos representando sobrecarga e métodos representando sobreposição. Altere os métodos main() de cada exercício para testar estes métodos que representam Polimorfismo. Utilize vários valores e mensagens de indicação para identificar que método foi executado em cada chamada.