Computação e Linguagem de Programação

Aula 5 parte 6

Professores
Sandro Fonseca de Souza
Dilson de Jesus Damião

Aula Anterior

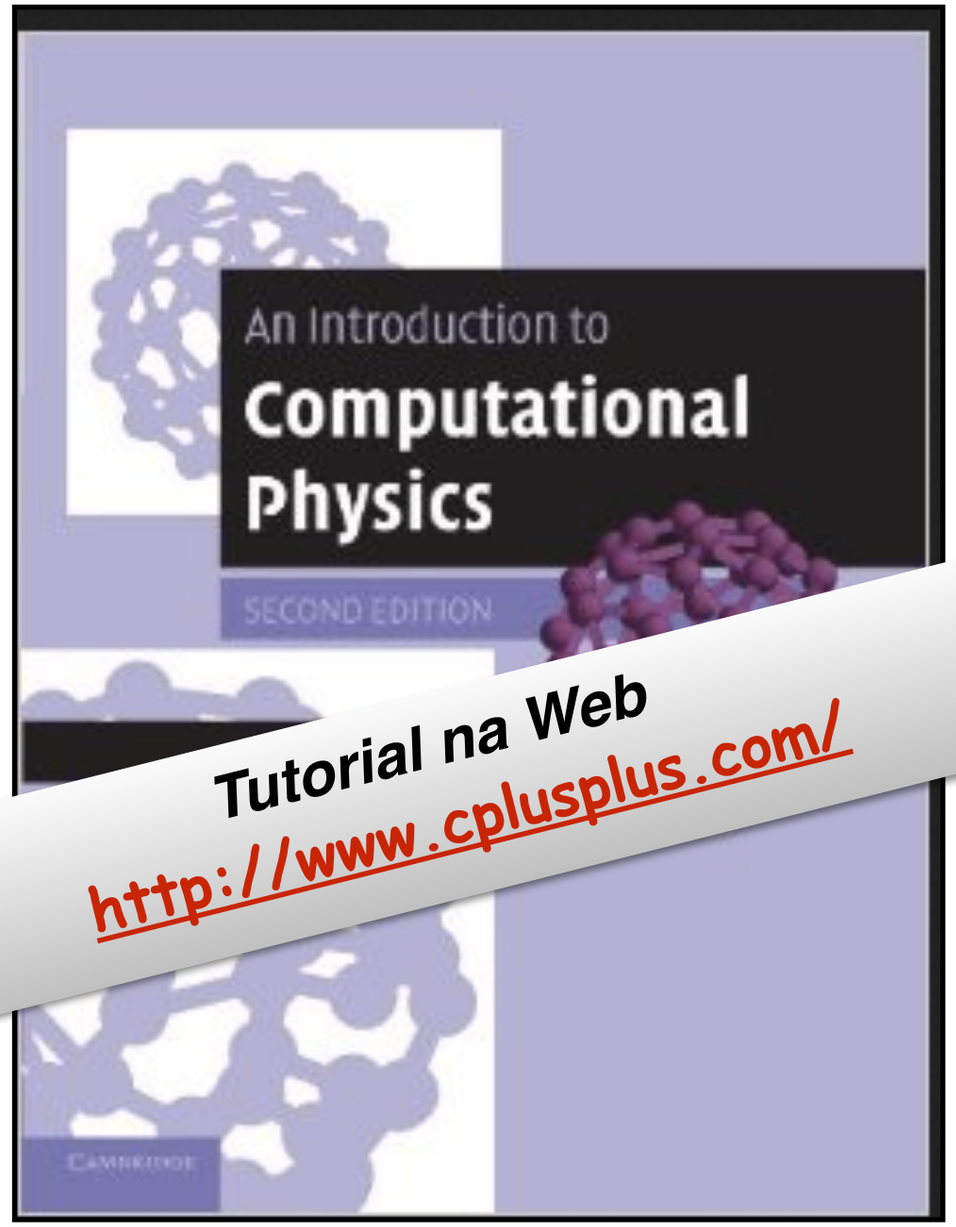
- Arrays
- Ponteiros e Referencias
- Arrays e Funções
- Gerenciamento de memória dinâmica
- Vetores em C++

Sumário

- Classes e Orientação a Objeto em C++

Bibliografia Sugerida

Bibliotecas padrão do C++
http://www.cplusplus.com/reference/



Tutorial na Web
http://www.cplusplus.com/

Introdução

In this lecture we will learn basic classes in C++.

C and C++ allow you to define your own data types.
These *user-defined* data types are created using the
`struct` or the `class` keywords.

In C++, a class is like an array: *it is a derived type*.
But unlike an array, the elements of a class may have different types. Furthermore, some elements of a class may be functions and operators.

Introdução

- The `struct` keyword is mostly used in the C programming language. In C, the elements of `struct` can be ordinary data types and/or other structures. To remain compatible with the C language, C++ maintains the `struct` keyword. However, in C++, a `struct` and a `class` have the same meaning and functionality.

- Although any storage region in RAM is referred to as an *object*, the word is usually used to describe variables whose data type is a class. Thus *object-oriented programming* involves programs that use classes.

Estruturas em C/C++

- A data structure (or derived data type) is a set of data elements grouped together under one name.
- These data elements, known as members, can have different types and different lengths.

```
struct name {
 type1 member_name1;
 type2 member_name2;
 .
 .
} object_names;
```

```
struct Student{
 string name;
 int mt1, mt2, fin;
 double avr;
} s1, s2;
```

- Here, **Student** is a new valid type name like the fundamental ones **int** or **double**. **s1** and **s2** are objects (or variables) derived from this new type.

Estruturas em C/C++

```
// A basic use of the stucure
#include <iostream>
#include <iomanip>
using namespace std;

struct Fruit{
 double weight;
 double price;
};

int main(){
 Fruit orange, apricot;

 orange.price  = 2.50; // TL/kg
 apricot.price = 3.25; // TL/kg

 cout << "Input the amount of orange in kg: ";
 cin >> orange.weight;
 cout << "Input the amount of apricot in kg: ";
 cin >> apricot.weight;

 cout << "\nTotal prices (TL):\n";
 cout << setprecision(2) << fixed;
 cout << "Orange = "  << orange.price * orange.weight << endl;
 cout << "Apricot = " << apricot.price * apricot.weight << endl;
}
```

```
Input the amount of orange in kg: 2
Input the amount of apricot in kg: 1.5

Total prices (TL):
Orange = 5.00
Apricot = 4.88
```

Classes Básicas

- A ***class*** is an expanded concept of a data structure in C. instead of holding only data, **a class can hold both data and functions.**

- An ***object*** is an instantiation of a class. In terms of variables, a class would be the *type*, and an object would be the *variable*.

- Classes are decelerated by using **`class`** keyword.

```
class class_name {
    access_specifier_1:
    member1;
    access_specifier_2:
    member2;
    ...
} object_names;
```

Classes Básicas

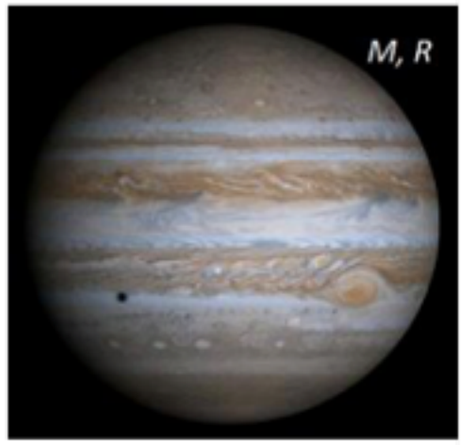
- An access specifier is one of the followings:
  - **`private`**
    members of a class are accessible only from within other members of the same class
  - **`public`**
    members are accessible from anywhere where the object is visible
  - **`protected`**
    members are accessible from members of their same class but also from members of their derived classes

By default, all members of a class declared with the `class` keyword have `private` access for all its members.

Classes Básicas

The following class can be used to represent a planet whose mass is *M* and radius is *R*.

```
// Example Class
class Planet{
  public:
    void SetMassRadius(double, double);
    double Density();
    double Gravity();
  private:
    double M, R, G;
};
```



- declares a class (i.e. a type) called **`Planet`**
- The functions:
  `SetMassRadius()`
  `Density()`
  `Gravity()`
  ⇄ *member functions* or *methods*.
- Member **`M`**,**`R`** and **`G`** have (default) **`private`** access and member functions have **`public`** access.

Planets and Pluto

This table contains selected physical characteristics of the planets and Pluto.

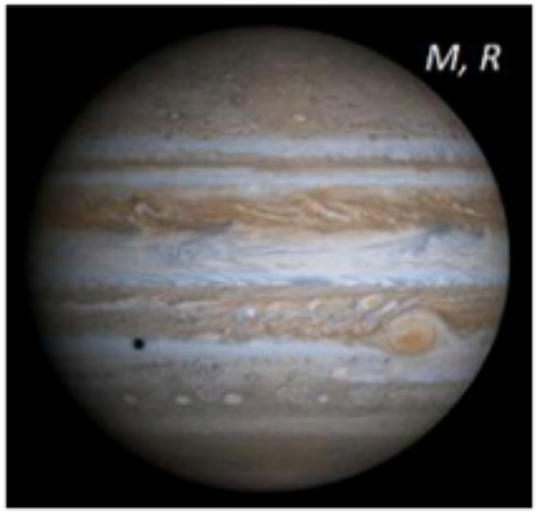
Planet	Equatorial Radius	Mean Radius	Mass	Bulk Density	Sidereal Rotation Period	Sidereal Orbit Period	V(1,0)	Geometric Albedo	Equatorial Gravity	Escape Velocity
	(km)	(km)	(x 10^{24} kg)	(g cm^{-3})	(d)	(y)	(mag)		($m\ s^{-2}$)	($km\ s^{-1}$)
Mercury	2439.7 [D] ±1.0	2439.7 [D] ±1.0	0.330104 [F] ±.000036	5.427 [*] ±.007	58.6462 [D]	0.2408467 [B]	-0.60 [E] ±0.10	0.106 [B]	3.70 [*]	4.25 [*]
Venus	6051.8 [D] ±1.0	6051.8 [D] ±1.0	4.86732 [G] ±.00049	5.243 [*] ±.003	-243.018 [D]	0.61519726 [B]	-4.47 [E] ±0.07	0.65 [B]	8.87 [*]	10.36 [*]
Earth	6378.14 [D] ±.01	6371.00 [D] ±.01	5.97219 [H] ±.00060	5.5134 [*] ±.0006	0.99726968 [B]	1.0000174 [B]	-3.86 [B]	0.367 [B]	9.80 [*]	11.19 [*]
Mars	3396.19 [D] ±.1	3389.50 [D] ±.2	0.641693 [I] ±.000064	3.9340 [*] ±.0008	1.02595676 [D]	1.8808476 [B]	-1.52 [B]	0.150 [B]	3.71 [*]	5.03 [*]
Jupiter	71492 [D] ±4	69911 [D] ±6	1898.13 [J] ±.19	1.3262 [*] ±.0004	0.41354 [D]	11.862615 [B]	-9.40 [B]	0.52 [B]	24.79 [*]	60.20 [*]
Saturn	60268 [D] ±4	58232 [D] ±6	568.319 [K] ±.057	0.6871 [*] ±.0002	0.44401 [D]	29.447498 [B]	-8.88 [B]	0.47 [B]	10.44 [*]	36.09 [*]
Uranus	25559 [D] ±4	25362 [D] ±7	86.8103 [L] ±.0087	1.270 [*] ±.001	-0.71833 [D]	84.016846 [B]	-7.19 [B]	0.51 [B]	8.87 [*]	21.38 [*]
Neptune	24764 [D] ±15	24622 [D] ±19	102.410 [M] ±.010	1.638 [*] ±.004	0.67125 [D]	164.79132 [B]	-6.87 [B]	0.41 [B]	11.15 [*]	23.56 [*]
Pluto	1151 [C] ±6	1151 [C] ±6	.01309 [N] ±.00018	2.05 [*] ±.04	-6.3872 [D]	247.92065 [B]	-1.0 [B]	0.3 [B]	0.66 [*]	1.23 [*]

Implementação da Classe Planet

- Consider a planet of mass *M* and equatorial radius *R*. The mean mass density *d* and equatorial gravity *g* of the planet are given respectively by

$$g = \frac{GM}{R^2}$$

$$d = \frac{M}{4\pi R^3/3}$$



- where *G* is the universal gravitational constant and has the value 6.67428×10^{-11} $m^3/kg/s$.

Implementação da Classe Planet

```
// A basic use of classes
#include <iostream>
#include <cmath>
using namespace std;

class Planet{
   public:
      void SetMassRadius(double, double);
      double Density();
      double Gravity();
   private:
      double M, R, G;
};

int main(){
  Planet Mars;
  Mars.SetMassRadius(6.4e23, 3.4e6);
  cout << "Density = " << Mars.Density() << endl;
  cout << "Gravity = " << Mars.Gravity() << endl;
}

// continue ...
```

Implementação da Classe Planet

```
// Set the mass (kg) and
// equatorial radius (m) of the planet
void Planet::SetMassRadius(double mass, double radius){
   M = mass;
   R = radius;
   G = 6.67428e-11;
}

// Mass density in g/cm3
double Planet::Density(){
 double d = M/(4.0*M_PI*R*R*R/3);
 return d * 1.0e-3;
}

// Surface gravity in m/s2
double Planet::Gravity(){
 double g = G*M/(R*R);
 return g;
}
```

```
Density = 3.88736
Gravity = 3.6951
```

Implementação da Classe Planet

- Here **`Mars`** is declared to be an object of the **`Planet`** class.
- Consequently, **`Mars`** has its own internal data members **`M`**, **`R`**, and **`G`** and has also ability call member functions.
- The mass and radius of **`Mars`** are supplied via the **`SetMassRadius()`** method.
- Its density and surface gravity are evaluated and output .
- Notice one must use the specifier **`Planet::`** before each member function to indicate that these functions are the members of the Planet class.
- The output shows that the density of the Mars is about 3.9 g/cm^3 and its surface gravity is 3.7 m/s^2.

Implementação da Classe Planet

- public members are accessible from outside the class but private members are not.

- Therefore, the following accesses are forbidden:

```
cout << Mars.M << endl; // forbidden
cout << Mars.R << endl; // forbidden
```

```
// Self contained implementation in a class
#include <iostream>
#include <cmath>
using namespace std;

class Planet{
 public:
   void SetMassRadius(double mass, double radius){
    M = mass; R = radius; G = 6.67428e-11;
   }
   double Density(){
     return 1.0e-3 * M/(4.0*M_PI*R*R*R/3);
   }
   double Gravity(){ return G*M/(R*R); }
 private:
   double M, R, G;
};

int main(){
 Planet Mars;
 Mars.SetMassRadius(6.4e23, 3.4e6);

 cout << "Density = " << Mars.Density() << endl;
 cout << "Gravity = " << Mars.Gravity() << endl;
}
```

Construtores e Destrutores

- The **`Planet`** class uses the **`SetMassRadius()`** function to initialize its objects. However, you can initialize the values when the object is declared like ordinary variables

```
int p = 35;
string name = "Bjarne";
```

- This is done by means of a *constructor* function which is a member function called automatically when an object is declared.

- A constructor function must have the same name as the class name and have no return type.

```
// A basic use of class constructor
#include <iostream>
#include <cmath>
using namespace std;

class Planet{
   public:
      Planet(double, double);
      double Density();
      double Gravity();
   private:
      double M, R, G;
};

int main(){
 Planet Mars(6.4e23, 3.4e6), Jupiter(1.9e27, 7.0e7);

   cout << "Mars Density = " << Mars.Density() << endl;
   cout << "Mars Gravity = " << Mars.Gravity() << endl;

   cout << "Jupiter Density = " << Jupiter.Density() << endl;
   cout << "Jupiter Gravity = " << Jupiter.Gravity() << endl;
}
// continue ...
```

```
// Set the mass (kg) and
// equatorial radius (m) of the planet
Planet::Planet(double mass, double radius){
    M = mass;
    R = radius;
    G = 6.67428e-11;
}

// Mass density in g/cm3
double Planet::Density(){
 double d = M/(4.0*M_PI*R*R*R/3);
 return d * 1.0e-3;
}

// Surface gravity in m/s2
double Planet::Gravity(){
 double g = G*M/(R*R);
 return g;
}
```

```
Mars Density = 3.88736
Mars Gravity = 3.6951
Jupiter Density = 1.32242
Jupiter Gravity = 25.8799
```

Ponteiros para Classes

It is perfectly valid to create pointers that point to classes.

For example:

```
Planet *p;
```

is a pointer to an object of class **`Planet`**.

In order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection.

```cpp
// Pointer to a class
#include <iostream>
#include <cmath>
using namespace std;

class Planet{
 public:
   Planet(double mass, double radius){
     M = mass; R = radius; G = 6.67428e-11;
   }
   double Density(){ return 1.0e-3 * M/(4.0*M_PI*R*R*R/3); }
   double Gravity(){ return G*M/(R*R); }
 private:
   double M, R, G;
};

int main(){
 Planet *gezegen = new Planet(6.4e23, 3.4e6);

 cout << "Density = " << gezegen->Density() << endl;
 cout << "Gravity = " << gezegen->Gravity() << endl;
}
```

Incluindo Classes de um Arquivo

The contents of the main program, and of the class(es), can be placed into separate files.

Then, using the **`#include`** directive you can use the class(es) required.

In general, the files containing classes (or functions) are called *header files*. Usually headers have the extension ".h" or ".hpp".

```
// Planet.h
#ifndef PLANET_H
#define PLANET_H

class Planet{
  public: Planet(double, double);
  double Density();
  double Gravity();
  private:
  double M, R, G;
};
// Constructor function to set the mass and radius of the planet
// By default the planet is assumed to be Earth
Planet::Planet(double mass = 6.0e24, double radius = 6.4e6){
  M = mass; R = radius;
  G = 6.67428e-11;
}
// Mass density in g/cm3
double Planet::Density(){
   return M/(4.0*M_PI*R*R*R/3) * 1.0e-3;
}
// Surface gravity in m/s2
double Planet::Gravity(){
 return G*M/(R*R);
}
#endif
```

```
// Including a class from a file
#include <iostream>
#include <cmath>
using namespace std;

#include "Planet.h"

int main(){
  Planet Mars(6.4e23, 3.4e6), Jupiter(1.9e27, 7.0e7);

  cout << "Mars Density = " << Mars.Density() << endl;
  cout << "Mars Gravity = " << Mars.Gravity() << endl;
  cout << "Jupiter Density = " << Jupiter.Density() << endl;
  cout << "Jupiter Gravity = " << Jupiter.Gravity() << endl;
}
```

Exercícios

Exercício 1

In the x-y plane, the general equation of a circle of radius r is given by: $(x - a)^2 + (y - b)^2 = r^2$.
Implement a `Circle` class. Each object of this class will represent a circle, storing its radius (r) and the a and b coordinates of its center as doubles. The class must include

- a default constructor function whose prototype is
  `Circle(double radius, double centerX, double centerY);`
  to set (initialize) radius and center coordinates.
- a member function named `double area()` that returns the area of the circle.
- a member function named `double circ()` that returns circumference.
- a member function named `bool isInside(double x, double y)` that returns `true` if the given point (x, y) is inside the circle and returns `false` otherwise.

Assume that the class decleration and its members/methods are stored in the file `Circle.h`. An example usage of the `Circle` is given below:

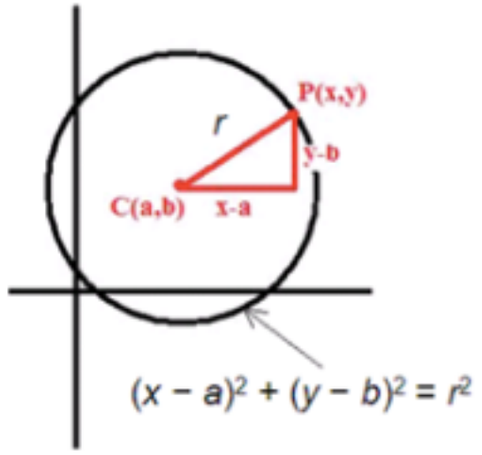
```
#include <iostream>
using namespace std;

#include "Circle.h"

int main(){
   // a circle whose center is origin
   Circle guzelCember(10.0, 0.0, 0.0);

   cout << guzelCember.area() << endl;
   cout << guzelCember.circ() << endl;
   cout << guzelCember.isInside(1.5, 2.7) << endl;

   return 0;
}
```



Exercício 2

Implement an `RCcircuit` class. Each object of this class will represent a simple charging RC circuit.

The class must include

- a default constructor function whose prototype is
  `RCcircuit(double R, double C, double V0);`
  to initialize the values of resistance (R) in Ohms, capacitor (C) in Farads and the potential difference across DC voltage source (V0) in Volts.
- a member function named `double current(double t)` that returns the current in the circuit at given time (in seconds) where t > 0.
- a member function named `double VC(double t)` that returns potential across the capacitor at given time (in seconds) where t > 0 .
- a member function named `double VR(double t)` that potential across the capacitor at given time (in seconds) where t > 0.
- a member function named `double tau()` that returns the time constant of the circuit defined by $T = R*C$.

Assume that the class decleration and its members/methods are stored in the file `RCCircuit.h`.

Example usage of the `RCCircuit` class is given below:

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

#include "RCCircuit.h"

int main(){
  RCcircuit *Devrem = new RCcircuit(2.2e+6, 1.0e-6, 12.);
  double time = 0.0;

  cout << "time constant: " << Devrem->tau() << endl;

  do{
      cout << Devrem->current(time) << "\t"
           << Devrem->VC(time)      << "\t"
           << Devrem->VR(time)      << endl;
      time += 0.1;

  }while(time < 5*Devrem->tau());

  return 0;
}
```