Topic 10
Basic Classes

Department of Engineering Physics
University of Gaziantep

Course web page
www.gantep.edu.tr/~bingul/ep241

Sep 2013
Introduction

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.
Structures in 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;
```

- 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.
// A basic use of the structure

#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):
";
    cout << setprecision(2) << fixed;
    cout << "Orange = " << orange.price * orange.weight << endl;
    cout << "Apricot = " << apricot.price * apricot.weight << endl;
}
Basic Classes

- 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.

```cpp
class class_name {
    access_specifier_1:
    member1;
    access_specifier_2:
    member2;
...
} object_names;
```
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.
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 \( M, R \) and \( G \) have (default) \texttt{private} access and member functions have \texttt{public} access.
### Planets and Pluto: Physical Characteristics

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

<table>
<thead>
<tr>
<th>Planet</th>
<th>Equatorial Radius</th>
<th>Mean Radius</th>
<th>Mass (x 10^{24} kg)</th>
<th>Bulk Density (g cm^{-3})</th>
<th>Sidereal Rotation Period (d)</th>
<th>Sidereal Orbit Period (y)</th>
<th>V(1,0) (mag)</th>
<th>Geometric Albedo</th>
<th>Equatorial Gravity (m s^{-2})</th>
<th>Escape Velocity (km s^{-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercury</td>
<td>2439.7 ±1.0</td>
<td>2439.7 ±1.0</td>
<td>0.330104 ±0.000036</td>
<td>5.427 ±0.007</td>
<td>58.6462 [D]</td>
<td>0.2408467 [B]</td>
<td>-0.60 ±0.10</td>
<td>0.106 [B]</td>
<td>3.70 [M]</td>
<td>4.25 [M]</td>
</tr>
<tr>
<td>Venus</td>
<td>6051.8 ±1.0</td>
<td>6051.8 ±1.0</td>
<td>4.86732 ±0.00049</td>
<td>5.243 ±0.003</td>
<td>-243.018 [D]</td>
<td>0.61519726 [B]</td>
<td>-4.47 ±0.07</td>
<td>0.65 [B]</td>
<td>8.87 [M]</td>
<td>10.36 [M]</td>
</tr>
<tr>
<td>Earth</td>
<td>6378.14 ±0.01</td>
<td>6371.00 ±0.01</td>
<td>5.97219 ±0.0060</td>
<td>5.5134 ±0.0006</td>
<td>0.99726968 [B]</td>
<td>1.0000174 [B]</td>
<td>-3.86 [B]</td>
<td>0.367 [B]</td>
<td>9.80 [M]</td>
<td>11.19 [M]</td>
</tr>
<tr>
<td>Mars</td>
<td>3396.19 ±1.1</td>
<td>3389.50 ±2</td>
<td>0.641693 ±0.00064</td>
<td>3.9340 ±0.0008</td>
<td>1.02595676 [D]</td>
<td>1.8808476 [B]</td>
<td>-1.52 [B]</td>
<td>0.150 [B]</td>
<td>3.71 [M]</td>
<td>5.03 [M]</td>
</tr>
<tr>
<td>Saturn</td>
<td>60268 ±4</td>
<td>58232 ±6</td>
<td>568.319 ±0.057</td>
<td>0.6871 ±0.0002</td>
<td>0.44401 [D]</td>
<td>29.447498 [B]</td>
<td>-8.88 [B]</td>
<td>0.47 [B]</td>
<td>10.44 [M]</td>
<td>36.09 [M]</td>
</tr>
<tr>
<td>Uranus</td>
<td>25559 ±4</td>
<td>25362 ±7</td>
<td>86.8103 ±0.0087</td>
<td>1.270 ±0.001</td>
<td>-0.71833 [D]</td>
<td>84.016846 [B]</td>
<td>-7.19 [B]</td>
<td>0.51 [B]</td>
<td>8.87 [M]</td>
<td>21.38 [M]</td>
</tr>
<tr>
<td>Neptune</td>
<td>24764 ±15</td>
<td>24622 ±19</td>
<td>102.410 ±0.010</td>
<td>1.638 ±0.004</td>
<td>0.67125 [D]</td>
<td>164.79132 [B]</td>
<td>-6.87 [B]</td>
<td>0.41 [B]</td>
<td>11.15 [M]</td>
<td>23.56 [M]</td>
</tr>
<tr>
<td>Pluto</td>
<td>1151 ±6</td>
<td>1151 ±6</td>
<td>0.01309 ±0.00018</td>
<td>2.05 ±0.04</td>
<td>-6.3872 [D]</td>
<td>247.92065 [B]</td>
<td>-1.0 [B]</td>
<td>0.3 [B]</td>
<td>0.66 [M]</td>
<td>1.23 [M]</td>
</tr>
</tbody>
</table>
Implementation of the Planet Class

- 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 $6.67428 \times 10^{-11}$ m$^3$/kg/s.
// 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 ...
// 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
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/cm³ and its surface gravity is 3.7 m/s².
- public members are accessible from outside the class but private members are not.

- Therefore, the following accesses are forbidden:

```cpp
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;
}
Constructors and Destructors

- 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/cm^3
double Planet::Density(){
    double d = M/(4.0*M_PI*R*R*R/3);
    return d * 1.0e-3;
}

// Surface gravity in m/s^2
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
Pointers to Classes

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

For example:

```c
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.
// 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;
}
Including a Class from a File

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

Then, using the \texttt{#include} directive you can use the class(es) required.

In general, the files containing classes (or functions) are called \textit{header files}. Usually headers have the extension ".h" or ".hpp".
```cpp
#ifndef PLANET_H
#define PLANET_H

class Planet{
    public: Planet(double mass, double radius);
    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;
}
Example: ‘A Cat class’

Each object of this class will represent a cat. The class includes

* a constructor function whose prototype is
  
  ```cpp
  Cat(int Age=1, double Mass=2.0);
  ```
  
  to set (initialize) the age and weight of the cat.

* a member function named `void speak()` that outputs a "meow" message.

* a member function named `void kill()` that reduces the cat's lives by one (the cat has nine lives).

* a member function named `double getMass()` to get the mass of the cat.

* a member function named `int getAge()` to get the age of the cat.

* a member function named `int getLife()` to get the remaining life(s) of the cat.

Example: ‘A Point Class’

Each object of this class will represent a Point in x-y plane.

The class includes

* a constructor function whose prototype is
  \[
  \text{Point(double xx=0, double yy=0);}
  \]
  to set (initialize) the coordinate.

* a member function named \text{double distance()}
  that returns the distance of the point to the origin.

* a member function named \text{double angle()}
  that returns the angle w.r.t x-axis
Homeworks

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

```cpp
#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;
}
```
Implement an RC circuit class. Each object of this class will represent a simple charging RC circuit.

The class must include

- a default constructor function whose prototype is
  
  \texttt{RCircuit(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 \texttt{double \ current(double t)} that returns the current in the circuit at given time (in seconds) where \( t > 0 \).
- a member function named \texttt{double \ VC(double t)} that returns potential across the capacitor at given time (in seconds) where \( t > 0 \).
- a member function named \texttt{double \ VR(double t)} that potential across the capacitor at given time (in seconds) where \( t > 0 \).
- a member function named \texttt{double \ tau()} that returns the time constant of the circuit defined by \( T = R \times C \).

Assume that the class declaration and its members/methods are stored in the file \texttt{RCircuit.h}.

Example usage of the RCircuit class is given below:

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

#include "RCircuit.h"

int main(){
    RCircuit *Devrem = new RCircuit(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;
}
```
Implement a Square class

Each object of this class will represent a square of given side.

The class includes

* a constructor function whose prototype is

  \[ \text{Square(double side = 1);} \]

  to set (initialize) the side.

* a member function named \text{double area()} that returns the area of the square.

* a member function named \text{double circ()} that returns the circumference of the square.

* a member function named \text{double diag()} that returns the diagonal length of the square.
**Implement a Cube class**

Each object of this class will represent a cube of given side.

The class includes

* a constructor function whose prototype is
  
  ```cpp
  Cube(double side = 1);
  ```

  to set (initialize) the side.

* a member function named `double area()`
  that returns the total surface area of the cube.

* a member function named `double volume()`
  that returns the volume of the cube.

* a member function named `double diag()`
  that returns the longest diagonal length of the cube.