Object Oriented Programming Development

تطوير البرمجة موجهة الأهداف
What are we doing today?

Introduction to:
- Objects
- Basic Terminology
- C++
- the module
What is Object Oriented Programming?

An object is like a black box. The internal details are hidden.

- Identifying objects and assigning responsibilities to these objects.

- Objects communicate to other objects by sending messages.

- Messages are received by the methods of an object.
What is an object?

- **Tangible Things** as a car, printer, ...
- **Roles** as employee, boss, ...
- **Incidents** as flight, overflow, ...
- **Interactions** as contract, sale, ...
- **Specifications** as colour, shape,
So, what are objects?

- an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.

Or

- An "object" is anything to which a concept applies.

Etc.
Why do we care about objects?

- **Modularity** - large software projects can be split up in smaller pieces.

- **Reuseability** - Programs can be assembled from pre-written software components.

- **Extensibility** - New software components can be written or developed from existing ones.
The two parts of an object

Object = Data + Methods

or to say the same differently:

An object has the responsibility to *know* and the responsibility to *do*.
Basic Terminology

Abstraction is the representation of the essential features of an object. These are ‘encapsulated’ into an abstract data type.

Encapsulation is the practice of including in an object everything it needs hidden from other objects. The internal state is usually not accessible by other objects.
Basic Terminology: Inheritance التوريث

Inheritance means that one class inherits the characteristics of another class. This is also called a “is a” relationship:

A car *is a* vehicle

A dog *is an* animal

A teacher *is a* person
Inheritance

Mechanism for deriving new classes from existing classes
A tandem bicycle is a kind of bicycle

Bicycle with two seats

A mountain bicycle is a kind of bicycle

Bicycle with shocks

A racing bicycle is a kind of bicycle

Lightweight aerodynamic construction

Tandem, mountain, and racing bicycles are specialized bicycles
ألا يكون جيداً أن ننشئ كينونات متخصصة دون أن نبدأ من الصفر عن طريق تعديل الشكل أو إضافة صورة أو تغيير الأبعاد أو تغيير التخصيص.

Be able to create specialized program objects without starting from scratch.

- Blinking rectangles
- Moving bitmaps
- Arbitrary precision numbers

Inheritance is the object-oriented programming mechanism for specialization.
Inheritance التوارث

Ability to define new classes of objects using existing classes as basis
الفصائل الجديدة هي اصدارات خاصة من الفصيلة الأم وترث خصائصها

Bicycle
Mountain Bikes
Racing Bikes
Tandem Bikes

is-a relationships
**Polymorphism** means “having many forms”. It allows different objects to respond to the same message in different ways, the response specific to the type of the object.

E.g. the message `displayDetails()` of the Person class should give different results when send to a Student object (e.g. the enrolment number).
Basic Terminology: Aggregation

Aggregation describes a “has a” relationship. One object is a part of another object.

We distinguish between composite aggregation (the composite “owns” the part) and shared aggregation (the part is shared by more than one composite).

A car has wheels.
The most important aspect of an object is its behaviour (the things it can do). A behaviour is initiated by sending a message to the object (usually by calling a method).
The two steps of Object Oriented Programming

- **Making Classes**: Creating, extending or reusing abstract data types.

- **Making Objects interact**: Creating objects from abstract data types and defining their relationships.
Inheritance

A natural way to reuse code

Programming by extension rather than reinvention

Object-oriented paradigm is well suited for this style of programming

Terminology

Base class (superclass)

Derived class (subclass)

Bicycle

Mountain Bikes

Racing Bikes

Tandem Bikes

is-a relationships
Object Oriented Programming

- Programmer *thinks* about and defines the attributes and behavior of objects.
- Often the objects are modeled after real-world entities.
- Very different approach than *function-based* programming (like C).
Object Oriented Programming

- Object-oriented programming (OOP)
  - Encapsulates data (attributes) and functions (behavior) into packages called classes.

- So, Classes are user-defined (programmer-defined) types.
  - Data (data members)
  - Functions (member functions or methods)

- In other words, they are structures + functions
Classes in C++

- A class definition begins with the keyword `class`.
- The body of the class is contained within a set of braces, `{   } ;` (notice the semi-colon).

```c++
class class_name {
    ...
    ...
    ...
    ...
};
```

Any valid identifier

Class body (data member + methods)
Classes in C++

- Within the body, the keywords `private:` and `public:` specify the access level of the members of the class.
  - The default is `private`.

- Usually, the data members of a class are declared in the `private:` section of the class and the member functions are in `public:` section.
Classes in C++

class class_name
{
    private:
    ...
    ...
    ...

    public:
    ...
    ...
    ...

};
Classes in C++

Member access specifiers

public:
- Can be accessed outside the class directly.
  - The public stuff is *the interface*.

private:
- Accessible only to member functions of class
- Private members and methods are for internal use only.
Class

Data

data1
data2
data3

Functions

func1()
func2()
func3()
Functions Are Public, Data Is Private

Private and public.
class foo
{
    private:
    int data;
    public:
    void memfunc (int d) 
    {
        data = d;
    }
};
Two objects of class smallobj
#include <string>
#include <iostream>

class Person {
    char name[20];
    int yearOfBirth;

public:
    void displayDetails() {
        cout << name << " born in " << yearOfBirth << endl;
    }
};
Defining the Class

class smallobj //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
{ somedata = d; }
void showdata() //member function to display data
{ cout << "\nData is " << somedata; }
};
#include <iostream>
using namespace std;

class part //define class
{
private:
int modelnumber; //ID number of widget
int partnumber; //ID number of widget part
float cost; //cost of part

public:
void setpart(int mn, int pn, float c) //set data
{
    modelnumber = mn;
partnumber = pn;
cost = c;
}

void showpart() //display data
{
    cout << "Model " << modelnumber;
cout << " , part " << partnumber;
cout << " , costs $" << cost << endl;
}
```cpp
#include <iostream>
using namespace std;

class smallobj { //define a class
private:
    int somedata,

public:
    void setdata(int d)
    { somedata = d; }

    void showdata() //member function to display data
    { cout << "Data is " << somedata << endl; }

    int main()
    {
        smallobj s1, s2; //define two objects of class smallobj
        s1.setdata(1066); //call member function to set data
        s2.setdata(1776);
        s1.showdata(); //call member function to display data
        s2.showdata();
        return 0;
    }
};
```
class part  //define class
{
private:
    int modelnumber;  //ID number of widget
    int partnumber;   //ID number of widget part
    float cost;       //cost of part

public:
    void setpart(int mn, int pn, float c)  //set data
    {
        modelnumber = mn;
        partnumber = pn;
        cost = c;
    }

    void showpart()  //display data
    {
        cout << "Model " << modelnumber;
        cout << ", part " << partnumber;
        cout << ", costs $" << cost << endl;
    }
};

int main()
{
    part part1;          //define object
    part1.setpart(6244, 373, 217.55F);  //call member function
    part1.showpart();     //call member function
    return 0;
}
// circles.cpp
// circles as graphics objects
#include "msoftcon.h" // for graphics functions
class circle // graphics circle
{
protected:
    int xCo, yCo; // coordinates of center
    int radius;
    color fillcolor; // color
    fstyle fillstyle; // fill pattern
public: // sets circle attributes
    void set(int x, int y, int r, color fc, fstyle fs)
    {
        xCo = x;
        yCo = y;
        radius = r;
        fillcolor = fc;
        fillstyle = fs;
    }
    void draw() // draws the circle
    {
        set_color(fillcolor); // set color
        set_fill_style(fillstyle); // set fill
        draw_circle(xCo, yCo, radius); // draw solid circle
    }
};
int main()
{
    init_graphics(); //initialize graphics system

circle c1; //create circles
circle c2;
circle c3;
    //set circle attributes
c1.set(15, 7, 5, cBLUE, X_FILL);
c2.set(41, 12, 7, cRED, O_FILL);
c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);

c1.draw(); //draw circles
c2.draw();
c3.draw();
set_cursor_pos(1, 25); //lower left corner return 0;
}

// engobj.cpp
// objects using English measurements
#include <iostream>
using namespace std;

class Distance //English Distance class
{
private:
  int feet;
  float inches;
public:
  void setdist(int ft, float in) //set Distance to args
  {  feet = ft; inches = in; }

  void getdist() //get length from user
  {
    cout << "\nEnter feet: "; cin >> feet;
    cout << "Enter inches: "; cin >> inches;
  }

  void showdist() //display distance
  {  cout << feet << "\'-" << inches << '\""; }
};

int main()
{
  Distance dist1, dist2; //define two lengths
  dist1.setdist(11, 6.25); //set dist1
  dist2.getdist(); //get dist2 from user

  //display lengths
  cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << endl;
  return 0;
}
This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r);
        double getDiameter();
        double getArea();
        double getCircumference();
};

No need for others classes to access and retrieve its value directly. The class methods are responsible for that only.

They are accessible from outside the class, and they can access the member (radius)
Creating an object of a Class

- Declaring a variable of a class type creates an **object**. You can have many variables of the same type (class).
  - **Instantiation**

- Once an object of a certain class is instantiated, a new memory location is created for it to store its data members and code.

- You can instantiate many objects from a class type.
  - **Ex**) Circle c; Circle *c;
#include <iostream>
using namespace std;

class Counter
{
    private:
        unsigned int count;       //count
    public:
        Counter() : count(0)      //constructor
        {
            /*empty body*/
        }
        void inc_count()           //increment count
        {
            count++;
        }
        int get_count()            //return count
        {
            return count;
        }
};

int main()
{
    Counter c1, c2;       //define and initialize

cout << "\nc1=" << c1.get_count();   //display
cout << "\nc2=" << c2.get_count();

c1.inc_count();          //increment c1
C2.inc_count();           //increment c2
C2.inc_count();           //increment c2

cout << "\nc1=" << c1.get_count();   //display again
cout << "\nc2=" << c2.get_count();

cout << endl;
return 0;
}
// cirtor.cpp
// circles use constructor for initialization
#include "msoftcon.h" // for graphics functions

class circle //graphics circle
{
protected:
   int xCo, yCo; //coordinates of center
   int radius;
   color fillcolor; //color
   fstyle fillstyle; //fill pattern

public:
   //constructor
   circle(int x, int y, int r, color fc, fstyle fs) :
       xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
   { }

   void draw() //draws the circle
   {
      set_color(fillcolor); //set color

      set_fill_style(fillstyle); //set fill
      draw_circle(xCo, yCo, radius); //draw solid circle
   }
};

int main()
{
   init_graphics(); //initialize graphics system
      //create circles
   circle c1(15, 7, 5, cBLUE, X_FILL);
   circle c2(41, 12, 7, cRED, 0_FILL);
   circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);

   c1.draw(); //draw circles
   c2.draw();
   c3.draw();
   set_cursor_pos(1, 25); //lower left corner
   return 0;
}
Destructors

We’ve seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

class Foo
{
    private:
        int data;
    public:
        Foo() : data(0) //constructor (same name as class)
        {
        }
        ~Foo() //destructor (same name with tilde)
        {
        }
};
dist3.add_dist(dist1, dist2)

Member functions of dist3 can refer to its data directly.

Data in objects passed as arguments is referred to with the dot operator.
Objects as Function Arguments

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

class Distance //English Distance class
{
private:
    int feet;
    float inches;
public:
    //constructor (no args)
    Distance() : feet(0), inches(0.0)
    {
    }
    //constructor (two args)
    Distance(int ft, float in) : feet(ft), inches(in)
    {
    }

    void getdist() //get length from user
    {
        cout << "Enter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }

    void showdist() //display distance
    {
        cout << feet << "-" << inches << "\n";
    }

    void add_dist( Distance, Distance ); //declaration
};

//add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
```
```c++

inches = d2.inches + d3.inches; // add the inches
feet = 0;                     // (for possible carry)
if(inches >= 12.0)            // if total exceeds 12.0,
{
    inches -= 12.0;          // then decrease inches
    feet++;                  // by 12.0 and
}                          // increase feet
feet += d2.feet + d3.feet;   // add the feet

int main()
{
    Distance dist1, dist3;   // define two lengths
    Distance dist2(11, 6.25); // define and initialize dist2
    dist1.getdist();         // get dist1 from user
    dist3.add_dist(dist1, dist2); // dist3 = dist1 + dist2
    // display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
```
**Figure 6.5**
The scope resolution operator.
Special Member Functions

- **Constructor:**
  - Public function member
  - called when a new object is created (instantiated).
  - Initialize data members.
  - Same name as class
  - No return type
  - Several constructors
    - Function overloading
class Circle
{
private:
    double radius;
public:
    Circle();
    Circle(int r);
    void setRadius(double r);
    double getDiameter();
    double getArea();
    double getCircumference();
};
Implementing class methods

- Class implementation: writing the code of class methods

There are two ways:

1. Member functions defined outside class
   - Using Binary scope resolution operator (::)
   - “Ties” member name to class name
   - Uniquely identify functions of particular class
   - Different classes can have member functions with same name

- Format for defining member functions

```cpp
ReturnType ClassName::MemberFunctionName() {
}
```
Implementing class methods

2. Member functions defined inside class

- Do not need scope resolution operator, class name;

```cpp
class Circle
{
    private:
        double radius;
    public:
        Circle() { radius = 0.0; }
        Circle(int r);
        void setRadius(double r) { radius = r; }
        double getDiameter() { return radius * 2; }
        double getArea();
        double getCircumference();
};
```
class Circle {
    private:
        double radius;
    public:
        Circle() { radius = 0.0; }
        Circle(int r);
        void setRadius(double r) { radius = r; }
        double getDiameter() { return radius * 2; }
        double getArea();
        double getCircumference();
};

Circle::Circle(int r) {
    radius = r;
}

double Circle::getArea() {
    return radius * radius * (22.0/7);
}

double Circle::getCircumference() {
    return 2 * radius * (22.0/7);
}
Accessing Class Members

- Operators to access class members
  - Identical to those for structs
    - Dot member selection operator (.)
      - Object
      - Reference to object
    - Arrow member selection operator (->)
      - Pointers
class Circle
{
  private:
    double radius;
  public:
    Circle() { radius = 0.0; }
    Circle(int r);
    void setRadius(double r) { radius = r; }
    double getDiameter() { return radius * 2; }
    double getArea();
    double getCircumference();
};
Circle::Circle(int r)
{
  radius = r;
}
double Circle::getArea()
{
  return radius * radius * (22.0/7);
}
double Circle::getCircumference()
{
  return 2 * radius * (22.0/7);
}

void main()
{
  Circle c1, c2(7);
  cout<<"The area of c1:"
    <<c1.getArea()<<"\n"
    //c1.radius = 5;//syntax error
  c1.setRadius(5);
  cout<<"The circumference"
    << c1.getCircumference()
    <<"\n"
  //c2.radius = 5; //syntax error
  cout<<"The Diameter of c2"
    <<c2.getDiameter()<<"\n";
}
class Circle {
    private:
        double radius;
    public:
        Circle() { radius = 0.0; }
        Circle(int r);
        void setRadius(double r) { radius = r; }
        double getDiameter() { return radius * 2; }
        double getArea();
        double getCircumference();
    }

    Circle::Circle(int r) {
        radius = r;
    }

    double Circle::getArea() {
        return radius * radius * (22.0/7);
    }

    double Circle::getCircumference() {
        return 2 * radius * (22.0/7);
    }

    void main() {
        Circle c(7);
        Circle *cp1 = &c;
        Circle *cp2 = new Circle;

        cout<<"The area of cp1 is 
        <<cp1->getArea();

    }
}
Destructors

- Special member function
- Same name as class
  - Preceded with tilde (~)
- No arguments
- No return value
- Cannot be overloaded
- Before system reclaims object’s memory
  - Reuse memory for new objects
  - Mainly used to de-allocate dynamic memory locations
Another class Example

This class shows how to handle time parts.

class Time
{
    private:
        int *hour,*minute,*second;
    public:
        Time();
        Time(int h,int m,int s);
        void printTime();
        void setTime(int h,int m,int s);
        int getHour(){return *hour;}
        int getMinute(){return *minute;}
        int getSecond(){return *second;}
        void setHour(int h){*hour = h;}
        void setMinute(int m){*minute = m;}
        void setSecond(int s){*second = s;}
        ~Time();
};
Time::Time()
{
    hour = new int;
    minute = new int;
    second = new int;
    *hour = *minute = *second = 0;
}

Time::Time(int h, int m, int s)
{
    hour = new int;
    minute = new int;
    second = new int;
    *hour = h;
    *minute = m;
    *second = s;
}

void Time::setTime(int h, int m, int s)
{
    *hour = h;
    *minute = m;
    *second = s;
}
void Time::printTime()
{
    cout<<"The time is : ("<<*hour<<":"<<*minute<<":"<<*second<<")"
        "endl;
}

Time::~Time()
{
    delete hour; delete minute; delete second;
}

void main()
{
    Time *t;
    t = new Time(3, 55, 54);
    t->printTime();

    t->setHour(7);
    t->setMinute(17);
    t->setSecond(43);

    t->printTime();
    delete t;
}
Reasons for OOP

1. Simplify programming

2. Interfaces
   - Information hiding:
     - Implementation details hidden within classes themselves

3. Software reuse
   - Class objects included as members of other classes
class RectangleShape {
    public:
        RectangleShape(SimpleWindow &W, 
                        float XCoord, float YCoord, const color &Color, 
                        float Width, float Height);
        void Draw();
        color GetColor() const; 
        void GetSize(float &Width, float &Height) const; 
        void GetPosition(float &x, float &y) const; 
        float GetWidth() const; 
        float GetHeight() const;  
        SimpleWindow & GetWindow() const; 
        void SetColor(const color &Color); 
        void SetPosition(float x, float y); 
        void SetSize(float Width, float Height);

    private:
        SimpleWindow &Window;  
        float XCenter; 
        float YCenter;   
        color Color; 
        float Width;  
        float Height;
};
Before Inheritance

```cpp
class CircleShape {
    public:
        CircleShape(SimpleWindow &W, float x, float y, const color &Color, float Diameter);
        void Draw();
        color GetColor() const;
        float GetSize() const;
        void GetPosition(float &x, float &y) const;
        SimpleWindow& GetWindow() const;
        void SetColor(const color &Color);
        void SetPosition(float x, float y);
        void SetSize(float Diameter);
    private:
        SimpleWindow &Window;
        float XCenter;
        float YCenter;
        color Color;
        float Diameter;
};
```
Shapes

Hierarchy

C: Shape
DM: Color
MF: GetColor(), SetColor()

C: RectangleShape
DM: Width, Height
MF: Draw(), GetWidth(), GetHeight(), SetSize()

C: TriangleShape
DM: SideLength
MF: Draw(), GetSideLength(), SetSize()

C: WindowObject
DM: Location, Window
MF: GetPosition(), GetWindow(), SetPosition()

C: EllipseShape
DM: Width, Height
MF: Draw(), GetWidth(), GetHeight(), SetSize()

C: Label
Inheritance is the process of creating new classes, called **derived classes**, from existing or **base classes**.
class WindowObject {

public:

    WindowObject(SimpleWindow &w, const Position &p);
    Position GetPosition() const;
    SimpleWindow& GetWindow() const;
    void SetPosition(const Position &p);

private:

    SimpleWindow &Window;
    Position Location;
};
WindowObject Constructor

WindowObject::WindowObject(SimpleWindow &w,
const Position &p) : Window(w),
Location(p) {
    // No body needed
}

Members are initialized in class definition order
WindowObject Inspectors

Position
   WindowObject::GetPosition() const
   {
      return Location;
   }

SimpleWindow&
   WindowObject::GetWindow() const {
      return Window;
   }
void WindowObject::SetPosition(const Position &p) {
    Location = p;
}
Defining a Derived Class

class DerivedClass : public BaseClass {
  public:
    // public section
    ...
  private:
    // private section
    ...
};

Derived class name
Access specifier (usually public)
Class name of base class
Declaring a Derived Class

```
class Shape : public WindowObject {
    public:
        Shape(SimpleWindow &w, const Position &p, const color &c = Red);
        color GetColor() const;
        void SetColor(const color &c);
    private:
        color Color;
};
```

Read this as *Shape is a kind of WindowObject*
Derived class constructor parameter list

Derived class name

Base class constructor parameter list (sublist of PList)

Base class name

DClass::DClass(PList) : BClass(BList), DMbrList {
   // Body of derived class constructor
   ...
};

Derived class data member initialization list (sublist of PList)
Implementing a Derived Class

Shape::Shape(SimpleWindow &w, const Position &p, const color &c) : WindowObject(w, p), Color(c) {
    // No body needed
}
color Shape::GetColor() const {
    return Color;
}
void Shape::SetColor(const color &c) {
    assert(c >= 0 && c < MaxColors);
    Color = c;
}
Basic Shapes

EllipseShape

RectangleShape

TriangleShape

Width

Height

SideLength
```cpp
#include "shape.h"

class TriangleShape : public Shape {
  public:
    TriangleShape(SimpleWindow &w, const Position &p, const color &c = Red, float slen = 1);
    float GetSideLength() const;
    void SetSize(float slen);
    void Draw();
  private:
    float SideLength;
};
```
#include "shape.h"

class EllipseShape : public Shape {

public:
    EllipseShape(SimpleWindow &w,
                 const Position &Center,
                 const color &c = Red, float Width = 1,
                 float Height = 2);

    float GetWidth() const;
    float GetHeight() const;
    void Draw();
    void SetSize(float Width, float Height);

private:
    float Width;
    float Height;
};
#include "shape.h"

class RectangleShape : public Shape {

public:
    RectangleShape(SimpleWindow &w, const Position &Center, const color &c = Red, float Width = 1, float Width = 2);
    float GetWidth() const;
    float GetHeight() const;
    void Draw();
    void SetSize(float Width, float Height);

private:
    float Width;
    float Height;
};
void TriangleShape::Draw() {
    const float Pi = 3.1415;
    const Position Center = GetPosition();
    const float SLength = GetSideLength();

    // Compute c, distance from center of triangle
    // to the top vertex, and a, the distance from
    // the center to the base of the triangle
    float c = SLength / (2.0 * cos(30 * Pi / 180.0));
    float a = tan(30 * Pi / 180.0) * .5 * SLength;
TriangleShape::Draw()

// Create an array containing the positions of
// the vertices of the triangle

vector Position TrianglePoints[3];
TrianglePoints[0] = Center + Position(0, -c),
TrianglePoints[1] = Center
    + Position(-.5 * SLength, a);
TrianglePoints[2] = Center
    + Position(.5 * SLength, a);

// Draw the triangle

GetWindow().RenderPolygon(TrianglePoints, 3,
    GetColor(), HasBorder());
#include "rect.h"
#include "ellipse.h"
#include "triangle.h"

SimpleWindow Window("TestShapes", 17.0, 7.0, Position(4.0, 4.0));

int ApiMain() {
    Window.Open();

    TriangleShape T(Window, Position(3.5, 3.5), Red, 3.0);
    T.Draw();

    RectangleShape R(Window, Position(8.5, 3.5), Yellow, 3.0, 2.0);
    R.Draw();

    EllipseShape E(Window, Position(13.5, 3.5), Green, 3.0, 2.0);
    E.Draw();

    return 0;
}
Fun with Shapes
Cleaning Up

```cpp
int ApiEnd()
{
    TWindow.Close();
    return 0;
}
```
class SomeClass
{
public:
    void MemberFunction();
    int MyPublicData;
protected:
    int MyProtectedData;
private:
    int MyPrivateData;
};

void SomeClass::MemberFunction()
{
Inheritance and Member Access

```c
void NonMemberFunction() {
    SomeClass C;
    C.MyPublicData = 1;  // access allowed
    C.MyProtectedData = 2;  // illegal
    C.MyPrivateData = 3;  // illegal
}
```
class BaseClass {
    public:    int MyPublicData;
    protected: int MyProtectedData;
    private:   int MyPrivateData;
};

class DerivedClass : public BaseClass {
    public: void DerivedClassFunction();
    // ...
};

void
## Controlling Inheritance

<table>
<thead>
<tr>
<th>Inheritance Type</th>
<th>Base class member access</th>
<th>Derived class member access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public</strong></td>
<td>public</td>
<td>public</td>
</tr>
<tr>
<td></td>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td>inaccessible</td>
</tr>
<tr>
<td><strong>protected</strong></td>
<td>public</td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td>inaccessible</td>
</tr>
<tr>
<td><strong>private</strong></td>
<td>public</td>
<td>private</td>
</tr>
<tr>
<td></td>
<td>protected</td>
<td>private</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td>inaccessible</td>
</tr>
</tbody>
</table>
A class is a specification or blueprint for a number of objects. Objects consist of both data and functions that operate on that data.

In a class definition, the members—whether data or functions—can be private, meaning they can be accessed only by member functions of that class, or public, meaning they can be accessed by any function in the program.
A member function is a function that is a member of a class. Member functions have access to an object’s private data, while non-member functions do not.

A constructor is a member function, with the same name as its class, that is executed every time an object of the class is created. A constructor has no return type but can take arguments.

It is often used to give initial values to object data members. Constructors can be overloaded, so an object can be initialized in different ways.
A destructor is a member function with the same name as its class but preceded by a tilde (~). It is called when an object is destroyed. A destructor takes no arguments and has no return value.

In the computer’s memory there is a separate copy of the data members for each object that is created from a class, but there is only one copy of a class’s member functions. You can restrict a data item to a single instance for all objects of a class by making it static.

One reason to use OOP is the close correspondence between real-world objects and OOP classes. Deciding what objects and classes to use in a program can be complicated. For small programs, trial and error may be sufficient. For large programs, a more systematic approach is usually needed.
Questions

1. What is the purpose of a class definition?
ماهو الهدف من تعريف الفصيلة

2. A ________ has the same relation to an ________ that a basic data type has to a variable of that type.

3. In a class definition, data or functions designated private are accessible
   a. to any function in the program.
   b. only if you know the password.
   c. to member functions of that class.
   d. only to public members of the class.

4. Write a class definition that creates a class called leverage with one private data member, crowbar, of type int and one public function whose declaration is void pry().

5. True or false: Data items in a class must be private.

6. Write a statement that defines an object called lever1 of the leverage class described in Question 4.

7. The dot operator (or class member access operator) connects the following two entities (reading from left to right):
   a. A class member and a class object
   b. A class object and a class
   c. A class and a member of that class
   d. A class object and a member of that class
1. A class declaration describes how objects of a class will look when they are created.
2. class object
3. c
4. class leverage
{"private:
int crowbar;
public:
void pry();
};
5. false; both data and functions can be private or public
6. leverage lever1;
7. d
8. Write a statement that executes the pry() function in the lever1 object, as described in Questions 4 and 6.

9. Member functions defined inside a class definition are _________ by default.

10. Write a member function called getcrow() for the leverage class described in Question 4. This function should return the value of the crowbar data. Assume the function is defined within the class definition.
11. A constructor is executed automatically when an object is ___.
12. A constructor’s name is the same as __________.
13. Write a constructor that initializes to 0 the crowbar data, a
member of the leverage class described in Question 4. Assume
that the constructor is defined within the class definition.
14. True or false: In a class you can have more than one
constructor with the same name.
15. A member function can always access the data
a. in the object of which it is a member.
b. in the class of which it is a member.
c. in any object of the class of which it is a member.
d. in the public part of its class.
16. Assume that the member function getcrow() described in
Question 10 is defined outside the class definition. Write the
declaration that goes inside the class definition.
8. lever1.pry();
9. inline (also private)
10. int getcrow()
    { return crowbar; }
11. created (defined)
12. the class of which it is a member
13. leverage() { crowbar = 0; }
14. true
15. a
16. int getcrow();
17. int leverage::getcrow()
    { return crowbar; }
18. member functions and data are, by default, public in structures but private in classes
19. three, one
20. calling one of its member functions
21. b, c, d
22. false; trial and error may be necessary
23. d
17. Write a revised version of the getcrow() member function from Question 10 that is defined outside the class definition.

18. The only technical difference between structures and classes in C++ is that __________.

19. If three objects of a class are defined, how many copies of that class’s data items are stored in memory? How many copies of its member functions?

20. Sending a message to an object is the same as ___.

21. Classes are useful because they
   a. are removed from memory when not in use.
   b. permit data to be hidden from other classes.
   c. bring together all aspects of an entity in one place.
   d. can closely model objects in the real world.