كلية الهندسة – قسم المعلوماتية

مقرر برمجة 2

إ. د. علي سليمان

محاضرات الأسبوع الثامن

الفصل الثاني 2025-2026

Virtual Functions and Polymorphism

التوابع الظاهرية وتعدد الأشكال

1. Introduction.
2. Definition and use of virtual functions.
3. Calling the virtual function through the reference parameter.
4. Inheritance of the phenotypic attribute.
5. Creating pure virtual functions.

1. مقدمة.
2. تعريف التوابع الظاهرية واستخدامها.
3. استدعاء التابع الظاهري من خلال معامل المرجع.
4. وراثة الصفة الظاهرية.
5. إنشاء التوابع الظاهرية الصرفة.

المحاضرة من المراجع :

[1]- Deitel & Deitel, C++ How to Program, Pearson; 10th Edition (February 29, 2016)

[2]- د.علي سليمان, البرمجة غرضية التوجه في لغة C++ 2009-2010

Introduction

مقدمة 1

- إن تقنية تعدد الأشكال polymorphism تساعد في تطوير برامج عامة بدلاً من كتابة برامج مخصصة لهدف معين ومحدد.

- polymorphism مدعومة في لغة C++ في وقت الترجمة compile time ووقت التنفيذ run time.
- تحقيق تعدد الأشكال في وقت الترجمة من خلال التحميل الزائد للتوابع والمعاملات.
- تحقيق تعدد الأشكال وقت التنفيذ فيتم من خلال الوراثة والتوابع الظاهرية.

- نرغب في هذا الفصل بطرح السؤال التالي: ما هو سلوك المترجم لدى محاولة أغراض صنف مشتق القيام بمناداة التوابع المعاد تعريفها ضمن الصنف المشتق؟.
- نحاول الإجابة عن هذا السؤال من خلال المثال التالي الذي يستند إلى البنية الهرمية Point/Circle التي قمنا بعرضها في الفصل السابق والتي نلخصها كما يلي:
- الملف الرأسي لتعريف الصنف Point :

```
#ifndef POINT_H
#define POINT_H
```

Introduction

مقدمة 2

```
class Point
{
public:
   Point( int = 0, int = 0 );
   void setX( int );
   int getX() const;
   void setY( int );
   int getY() const;
   void print() const;
private:
   int x;
   int y;
};
#endif
```

Introduction

مقدمة 3

ملف تعريف توابع الصنف PointF :

```
// function PointF
#include <iostream>
#include "point.h"
using namespace std;
Point::Point(int xValue,int yValue)
{
    x=xValue;
    y=yValue;
}
void Point::setX( int xValue )
{   x = xValue; }
int Point::getX() const
{   return x;     }
```

```
void Point::setY( int yValue )
{   y = yValue;   }

int Point::getY() const
{   return y;     }

void Point::print() const
{   cout << '[' << getX() << ", " << getY() << ']';}
```

الملف الرأسي لتعريف الصنف Circle.h :

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
class Circle : public Point {
```

```
public:
   Circle( int = 0, int = 0, double = 0.0 );
   void setRadius( double );
   double getRadius() const;
   double getDiameter() const;
   double getCircumference() const;
   double getArea() const;
   void print() const;

private:
   double radius;
};

#endif
```

Introduction

مقدمة 6

ملف تعريف الصنف CircleF :

```
// function CircleF

#include <iostream>
#include "circle.h"
using namespace std;
Circle::Circle( int xValue, int yValue, double radiusValue ):
Point( xValue, yValue )
{   setRadius( radiusValue );}

void Circle::setRadius( double radiusValue )
{   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );}

double Circle::getRadius() const
{   return radius;      }
```

Introduction

مقدمة 7

```
double Circle::getDiameter() const
{   return 2 * getRadius();   }

double Circle::getCircumference() const
{   return 3.14159 * getDiameter();}

double Circle::getArea() const
{   return 3.14159 * getRadius() * getRadius();}

void Circle::print() const
{  cout << "center = ";    Point::print();
   cout << "; radius = " << getRadius();
```

- لقد تم اشتقاق الصنف Circle بالوراثة العامة للصنف Point مع إضافة عضو بيانات جديد وهو نصف القطر radius وتعريف بعض التوابع الإضافية الخاصة بالتعامل مع الأغراض من هذا الصنف.

- نلاحظ أن التابع print الذي تم تعريفه في الصنف Point ووراثته ضمن الصنف المشتق Circle حيث تم إعادة تعريفه ضمن الصنف المشتق.
- لنختبر هذه البنية الهرمية وسلوك التابع print ضمن هذه البنية من خلال البرنامج البسيط التالي :

```
// VirtualFuncPolymorphism.cpp : main project file.

#include <iostream>
#include <iomanip>
#include "point.h"    // Point class definition
#include "circle.h"   // Circle class definition
using namespace std;
int main()
{
  Point point( 30, 50 );      Circle circle( 120, 89, 2.7 );
```

Introduction

مقدمة 9

```
    point.print();     cout<<endl;
    circle.print();    cout<<endl;
       system("pause");      return 0;
} // end main
```

يعطي هذه البرنامج على خرجه :

[30, 50]
center = [120, 89]; radius = 2.7
Press any key to continue . . .

- لدى استدعاء التابع print مع أغراض من الصنف Point فإن النسخة الخاصة بهذا الصنف هي التي تنفذ، وكذلك لدى استدعاء نفس التابع مع أغراض الصنف Circle فإن النسخة المحملة تحميلاً زائداً ضمن الصنف Circle هي التي تنفذ علماً أن الصنف Circle يتضمن النسخة الموروثة من الصنف Point.

لنحاول إجراء الاختبار السابق نفسه ولكن من خلال المؤشرات إلى أغراض هذه المرة. البرنامج التالي يوضح هذا الاستخدام :

```
#include <iostream>
#include <iomanip>
#include "point.h"                  // Point class definition
#include "circle.h"                 // Circle class definition
using namespace std;

int main()
{
 Point point( 30, 50 );     Circle circle( 120, 89, 2.7 );
   cout<<"Calling print function without pointers :"<<endl;
   point.print();     cout<<endl;
   circle.print();    cout<<"\n\n";
```

```
    Point *pointPtr = &point;          // base-class pointer
    Circle *circlePtr = &circle;       // derived-class pointer

    cout<<"Calling print function using pointers : "<<endl;
    pointPtr->print();                 // invokes Point's print
    cout<<endl;

    circlePtr->print();                // invokes Circle's print
    cout<<endl<<endl;

    cout<<"using a base-class pointer to reference a derivedClass object \n";
    pointPtr=&circle;   cout<<"Calling print function :"<<endl;
    pointPtr->print();                 // invokes Point's print????
    cout<<endl;    system("pause");      return 0;
}                                      // end main
```

Introduction

مقدمة 12

يعطي هذا البرنامج على خرجه :

```
Calling print function without pointers :
[30, 50]
center = [120, 89]; radius = 2.7

Calling print function using pointers :
[30, 50]
center = [120, 89]; radius = 2.7

using a base-class pointer to reference a  derivedClass object
Calling print function :
[120, 89]
Press any key to continue . . .
```

Introduction

مقدمة 13

1. تم تعريف غرضين من النوع Point و Circle على التوالي واستدعاء التابع print المرتبط بكل غرض على حدى.
2. تم تعريف مؤشرين إلى غرضين من النوع Point و Circle واستخدامهما للتأشير إلى الغرضين السابقين واستدعاء التابع print بواسطة هذين المؤشرين وكانت الاستجابة شبيهة بالاستدعاء الأول.
3. تم جعل المؤشر إلى غرض من الصنف الأساس يشير إلى غرض من الصنف المشتق (إن هذا الأمر متاح لأن كل غرض من أغراض صنف مشتق هو غرض من أغراض الصنف الأساس) ومن ثم استدعاء التابع Print إلا أن الاستدعاء في هذه الحالة كان لنسخة التابع العائدة للصنف الأساس وليس للصنف المشتق مما يعني أن التابع المستدعى يرتبط بنوع المؤشر وليس نوع الغرض الذي يؤشر إليه وانطلاقا من هذه الحقيقة فإننا لا نستطيع استدعاء توابع الصنف Circle مع هذا المؤشر.
4. الخلاصة: إن المترجم يميز النسخة التي يجب أن يتعامل معها للتابع المعاد تعريفه ضمن الصنف المشتق في حال الاستدعاء العادي أما في حال استخدام المؤشرات فإنه يعاني في بعض الحالات من بعض المشاكل في تمييز النسخة التي سيتعامل معها.

2-4- التوابع الظاهرية 1 Virtual functions

- إن التابع المستدعى يرتبط بنوع المؤشر وليس نوع الغرض الذي يؤشر إليه.
- يمكن باستخدام التوابع الظاهرية أن نتعامل مع نوع الغرض المؤشر إليه بدلاً من نوع المؤشر.
- التابع الظاهري virtual function هو تابع عضو يتم التصريح عنه ضمن الصنف الأساس ويعاد تعريفه من قبل الصنف المشتق.
- لإنشاء تابع ظاهري يسبق التصريح عن التابع في الصنف الأساس بالكلمة المفتاحية virtual.
- عندما تتم وراثة صنف يحتوي على تابع ظاهري، يقوم الصنف المشتق بإعادة تعريف التابع الظاهري ليتلاءم مع احتياجاته الخاصة.

- باختصار يمكن القول إن التوابع الظاهرية تقوم على فلسفة " واجهة واحدة، طرائق متعددة one interface multiple methods , " والتي تعتبر الأساس لتعدد الأشكال polymorphism.

- يعرف التابع الظاهري ضمن الصنف الأساس شكل الواجهة إلى ذلك التابع، وكل إعادة تعريف للتابع الظاهري من قبل صنف مشتق تعرف سلوكه (عمله) بشكل مرتبط بالصنف المشتق. وبالتالي فإن إعادة التعريف تنشئ طريقة محددة specific method.

Virtual functions

2-4- التوابع الظاهرية 2

- تتصرف التوابع الظاهرية تماماً كأي نوع آخر من توابع الصنف الأعضاء عند الوصول إليه بشكل عادي، في حين أن ما يعطي التوابع الظاهرية أهمية هو طريقة عملها عندما يتم الوصول إليها من خلال مؤشر.
- إن المؤشر إلى صنف أساس يمكن أن يستخدم للإشارة إلى أغراض من أي صنف مشتق من ذلك الصنف الأساس،
- عندما يشير مؤشر إلى صنف أساس إلى غرض مشتق يحتوي على تابع ظاهري، فإن C++ تحدد أي نسخة من ذلك التابع سيتم استدعاؤها وذلك بحسب نوع الغرض المشار إليه من قبل المؤشر، وهذا التحديد يتم خلال وقت التنفيذ. وبالتالي عندما تتم الإشارة إلى أغراض مختلفة ، يتم تنفيذ نسخ مختلفة من التابع الظاهري. التأثير نفسه يحصل لدى استخدام المراجع إلى صنف أساس.
- لنحاول اختبار مثل هذا الأمر من خلال تعديل تعريف الصنف **Point** وجعل التابع **print** تابعاً ظاهرياً وذلك بإضافة الكلمة المفتاحية **virtual** إلى بداية التصريح عن التابع كما يلي :

```
virtual void print() const;
```

ومن ثم تنفيذ البرنامج التالي:

```
#include <iostream>
#include <iomanip>
#include "point.h"    // Point class definition
#include "circle.h"   // Circle class definition
using namespace std;
```

Virtual functions

2-4- التوابع الظاهرية 3

```
int main()
{   Point point( 30, 50 );
   Circle circle( 120, 89, 2.7 );
   cout<<"Calling print function without pointers : "<<endl;
   point.print();   cout<<endl;
   circle.print();     cout<<"\n\n";

   Point *pointPtr = &point;     // base-class pointer
   Circle *circlePtr = &circle;  // derived-class pointer
   cout<<"Calling print function using pointers : "<<endl;
   pointPtr->print();  // invokes Point's print
   cout<<endl;
   circlePtr->print();  // invokes Circle's print
   cout<<"\n\n";

   cout<<"using a base-class pointer to reference a derived-class object\n”;
```

Virtual functions

4 التوابع الظاهرية 2-4-

```
pointPtr = &circle;
   cout<<"Calling print function : "<<endl;
   pointPtr->print();  // invokes Circle's print
   cout<<"\n\n";

   cout<<"using a base-class reference to a derived-class object "<<endl;
   Point &pointRef = circle;
   cout<<"Calling print function : "<<endl;
   pointRef.print();  // invokes Circle's print
   cout<<endl;system("pause");    return 0;
}      // end main
```

الخرج:

```
Calling print function without pointers :
[30, 50]
center = [120, 89]; radius = 2.7
```

Virtual functions

2-4- التوابع الظاهرية 5

```
Calling print function using pointers :
[30, 50]
center = [120, 89]; radius = 2.7

using a base-class pointer to reference a derived-class object
Calling print function :
center = [120, 89]; radius = 2.7

using a base-class reference to a derived-class object Calling
print function :
center = [120, 89]; radius = 2.7
Press any key to continue . . .
```

قمنا في المثال السابق باستدعاء التابع الظاهري **print** من خلال مؤشر ومرجع إلى الصنف الأساس Point حيث نلاحظ من الخرج بأن نسخة التابع التي نفذت هي النسخة المرتبطة بالغرض المؤشر إليه.

2-4- التوابع الظاهرية 6 Virtual functions

- إن هذه القدرة لدى الأغراض المرتبطة بعدة أصناف مختلفة متصلة ببعضها البعض من خلال أساليب الوراثة على التجاوب بشكل متمايز مع نفس الاستدعاء لتابع من التوابع الأعضاء تدعى بتعددية الأشكال Polymorphism وتبنى هذه القدرة من خلال التوابع الظاهرية.
- تساعد تقنية تعدد الأشكال في تطوير برامج عامة بدلاً من كتابة برامج مخصصة لهدف معين ومحدد، إذ يمكن استخدام هذه التقنية لكتابة برامج قادرة على التعامل مع أصناف تنتمي إلى نفس البنية الهرمية المشتقة من صنف أساس محدد.
- تسمح عملية تعدد الأشكال بزيادة القدرة على توسيع البرامج لأنها تمكننا من كتابة هذه البرامج دون النظر إلى نوع الأغراض المتعامل معها وبالتالي يمكننا إضافة أنماط جديدة من الأغراض دون أي تعديل على مستوى النظام الأساس وإنما فقط تعديل البرامج الزبونة التي ستتعامل مع النوع الجديد.
- في حال استدعاء التابع الظاهري بشكل عادي فإن الاستجابة تكون شبيهة بالاستجابة في حال عدم استخدام التوابع الظاهرية وتدعى تعددية الأشكال في هذه الحالة باسم تعددية الأشكال وقت الترجمة compile-time polymorphism ، أما في حال استدعاء التابع الظاهري باستخدام المؤشرات أو المراجع فإن الاستجابة تكون مرتبطة بنوع الغرض المؤشر إليه وتدعى تعددية الأشكال في هذه الحالة باسم تعددية الأشكال وقت التنفيذ run-time polymorphism.

3-4- وراثة الصفة الظاهرية 1

The Inheritance of visual attribute

- عندما تتم وراثة تابع ظاهري، فإن طبيعته الظاهرية تتم وراثتها أيضاً. وهذا يعني أنه عندما يتم استخدام صنف مشتق قام بوراثة تابع ظاهري كصنف أساس لصنف مشتق آخر فإن التابع الظاهري يمكن أن يتم إعادة تعريفه مجدداً. وبكلام آخر،
  بغض النظر عن عدد مرات وراثة التابع الظاهري فهو يبقى ظاهرياً.
- يمكن اختبار هذا الأمر من خلال توسيع البنية الهرمية Point/Circle واشتقاق الصنف Cylinder من الصنف Circle باستخدام أسلوب الوراثة العامة. تم إعادة تعريف التابع print ضمن الصنف Cylinder وبما أن للاسطوانة مساحة لذا تم إعادة تعريف التابع getArea ضمن هذا الصنف وإعطاء التابع getArea الصفة الظاهرية ضمن الصنف Circle فأصبح

```
virtual double getArea() const;
```

الملف الرأسي للصنف Cylinder :

```
#ifndef CYLINDER_H
#define CYLINDER_H
#include "circle.h"

// Circle class definition
class Cylinder : public Circle {

public:
```

3-4- وراثة الصفة الظاهرية 2

The Inheritance of visual attribute

```
Cylinder(int=0,int =0,double =0.0,double =0.0);
   void setHeight( double );
   double getHeight() const;
   double getArea() const;          // return Cylinder's area
   double getVolume() const;        // return Cylinder's volume
   void print() const;              // output Cylinder
 private:
   double height;                  // Cylinder's height
}; // end class Cylinder
#endif
```

ملف تعريف الصنف CylinderF :

```
#include <iostream>
#include <iomanip>
#include "cylinder.h"    // Cylinder class definition
using namespace std;
```

3-4- وراثة الصفة الظاهرية 3

The Inheritance of visual attribute

```
// default constructor
Cylinder::Cylinder(int xValue,int yValue,double radiusValue, double heightValue)
:Circle(xValue, yValue,radiusValue) {    setHeight( heightValue );} // end
Cylinder constructor

// set Cylinder's height
void Cylinder::setHeight( double heightValue )
{   height = ( heightValue < 0.0 ? 0.0 : heightValue );
} // end function setHeight

// get Cylinder's height
double Cylinder::getHeight() const
{   return height;} // end function getHeight

// override virtual function getArea: return Cylinder area
double Cylinder::getArea() const
{   return 2*Circle::getArea()+getCircumference()* getHeight();
} // end function getArea
```

3-4- وراثة الصفة الظاهرية 4

The Inheritance of visual attribute

```
// override virtual function getVolume: return Cylinder volume
double Cylinder::getVolume() const
{   return Circle::getArea()*getHeight();} // end function getVolume
```

```
// override virtual function print: output Cylinder object
void Cylinder::print() const
{   Circle::print();      cout << "; height is " << getHeight();
} // end function print
```

أما برنامج الاختبار :

```
#include <iostream>
#include <iomanip>
#include "point.h"      // Point class definition
#include "circle.h"     // Circle class definition
#include "cylinder.h"   // Cylinder class definition
#include "cylinder.h"    // Cylinder class definition
using namespace std;
```

3-4- وراثة الصفة الظاهرية 5

The Inheritance of visual attribute

```
int main()
{  Point point( 7, 11 );Circle circle( 22, 8, 3.5 );
   Cylinder cylinder(10,10,3.3,10);
   cout<<"calling print without pointers :"<<endl;
   cout<<"================================"<<endl;

   point.print();cout << endl;
   circle.print();cout <<endl;
   cylinder.print();cout <<"\n\n";

   cout<<"calling print using a pointer to Point objects  :"<<endl;
   cout<<"========================================="<<endl;
   Point *pointPtr=&point;pointPtr->print();   cout<<endl;
   pointPtr=&circle;pointPtr->print();   cout<<endl;
   pointPtr=&cylinder;    pointPtr->print();   cout<<"\n\n";
```

أما ناتج التنفيذ فهو :

3-4- وراثة الصفة الظاهرية 6

The Inheritance of visual attribute

```
cout<<"calling print and getArea using a pointer to circle objects  :\n";
  cout<<"=========================================="<<endl;

   Circle *circlePtr=&circle;   circlePtr->print();   cout<<endl;
   cout<<circlePtr->getArea();  cout<<endl;
   circlePtr=&cylinder;circlePtr->print();   cout<<endl;
   cout<<circlePtr->getArea();  cout<<endl;
   system("pause");   return 0;
} // end main
```

أما ناتج التنفيذ فهو :

calling print without pointers :

==========

[7, 11]
center = [22, 8]; radius = 3.5
center = [10, 10]; radius = 3.3; height is 10

calling print using a pointer to Point objects :

==========

[7, 11]
[22, 8]; radius = 3.5
[22, 8]; radius = 3.510; height is 10

calling print and getArea using a pointer to circle objects :

==========

[22, 8]; radius = 3.5
38.4845
[22, 8]; radius = 3.510; height is 10
Area =38.4845
Press any key to continue . . .

نلاحظ من هذا المثال أن التابع print المعرف ضمن الصنف Circle قد اكتسب الصفة الظاهرية للتابع print الخاص بالصنف Point.

ماذا لو لم يتضمن الصنف Cylinder تعريفاً للتابع print فكيف سيكون سلوك البرنامج السابق ؟.

3-4- وراثة الصفة الظاهرية 8

The Inheritance of visual attribute

في هذه الحالة سيتم استخدام التابع المعرف في الصنف الأساس ويمكن تعميم هذه الخاصية بالقول :

بما أن الوراثة في لغة C++ ذات طبيعة شجرية، فإن ذلك يعني أن التوابع الظاهرية أيضاً ذات طبيعة شجرية، وهذا يعني أنه عندما يفشل صنف مشتق في إعادة تعريف تابع ظاهري فإن أول عملية إعادة تعريف تمت باتجاه معاكس للاشتقاق هي التي تستخدم.

ويكون خرج البرنامج السابق في هذه الحالة :

calling print without pointers :

==========

[7, 11]
[22, 8]; radius = 3.5
[22, 8]; radius = 3.5

calling print using a pointer to Point objects :

==========

[7, 11]
[22, 8]; radius = 3.5
[22, 8]; radius = 3.5

calling print and getArea using a pointer to circle objects :

==========

[22, 8]; radius = 3.5
38.4845
[22, 8]; radius = 3.5
Area =38.4845
Press any key to continue . . .

- نلاحظ أن التابع print للصنف Circle هو الذي تم تنفيذه لدى استدعاء التابع print مع أغراض من النمط Cylinder نظراً لعدم تعريف التابع print() في الصنف Cylinder. أي أنه عندما لا يتم إعادة تعريف التابع الظاهري في الصنف المشتق فإن النسخة المعرفة في الصنف الأساس هي التي سيتم استدعاؤها.
- لنحاول الآن عكس الصورة، بمعنى آخر ماذا لو لم يكن للتابع الظاهري تعريفاً ذو معنى في الصنف الأساس؟.
- في هذه الحالة فإن التابع الظاهري يجب أن يعاد تعريفه في الأصناف المشتقة ويتم التصريح عن التابع الظاهري في الصنف الأساس على شكل تابع ظاهري صرف (التابع الظاهري الصرف هو تابع ظاهري ليس لتعريفه معنى في الصنف الأساس).

يأخذ التصريح عن تابع ظاهري صرف الشكل العام التالي :

4-4- التوابع الظاهرية الصرفة والأصناف المجردة 2

Pure virtual function and abstract classes

```
virtual type func-name(parameter-list) = 0;
```

- عندما يتم جعل تابع ظاهري ما صرفاً، فإن أي صنف مشتق يجب أن يقوم بتعريفه، وإن فشل الصنف المشتق في تعريف التابع الظاهري الصرف، فإن ذلك يؤدي إلى ظهور خطأ وقت الترجمة compile-time error.
- نقول عن الصنف الذي يحوي على تابع ظاهري صرف واحد على الأقل على أنه صنف مجرد abstract class. وبما أن الصنف المجرد يحوي تابعاً واحداً أو أكثر ليس لتعريفها معنى (أي توابع ظاهرية صرفة) فإنه لا يمكن إنشاء أغراض من صنف مجرد، بدلاً من ذلك فإن الصنف المجرد ينشئ نمطاً غير مكتمل يمكن استخدامه في تشكيل أصناف مشتقة. وتدعى الاصناف التي يمكن أن نشتق منها أغراض بالأصناف المجسدة Concerete classes.
- عند محاولة اشتقاق أغراض من صنف مجرد إلى حدوث أخطاء أثناء الترجمة, وعند تحميل التوابع الظاهرية الصرفه بشكل زائد ومن ثم محاولة اشتقاق أغراض من هذا الصنف إلى حدوث أخطاء أثناء الترجمة ايضاً.
- لاختبار هذا النمط من التوابع، سنحاول إعادة تعريف البنية الهرمية Point/Circle/Cylinder حيث سنعرف ضمن الصنف Point التابعين getArea ، getVolume الذين يقومان بحساب المساحة والحجم على التوالي. إن هذين التابعين لا يملكان تعريفاً ذو معنى بالنسبة للصنف Point وبالتالي سنقوم بالتصريح عنهما على أنهما تابعين ظاهريين صرفين ومن ثم سنقوم بإعادة تعريف هذين التابعين في الصنفين المشتقين Circle و Cylinder كما يلي (نركز فقط على التعديلات الواجب إجراؤها لتحقيق المثال تجنباً لتكرار الشيفرة) :

4-4- التوابع الظاهرية الصرفة والأصناف المجردة 3

التعديلات على الملف الرأسي لتعريف الصنف Point :

التصريح عن التابعين الظاهريين الصرفين getArea و getVolume كما يلي :

```
virtual double getArea() const=0;     // Pure virtual function
virtual double getVolume() const=0;   // Pure virtual function
```

التعديلات على ملف تعريف الصنف PointF :

لا يوجد أي تعديلات.

التعديلات على الملف الرأسي لتعريف الصنف Circle:

إن الملف الرأسي يتضمن تصريحاً عن التابع getArea وبالتالي يكفي أن يتم إضافة التصريح عن التابع getVolume كما يلي :

```
double getVolume() const;
```

التعديلات على ملف تعريف الصنف CircleF :

إن الصنف Circle يتضمن تعريفاً للتابع getArea إلا أنه لا يتضمن تعريفاً للتابع getVolume وبالتالي لابد من إعطاءه تعريفاً ضمن ملف تعريف الصنف وليكن التعريف التالي :

```
double Circle::getVolume() const
{      return 0.0;   }
```

أما تعريف الصنف Cylinder فليس بحاجة لأي تعديل لأنه يقوم أساساً بإعادة تعريف التابعين المذكورين.
يمكن اختبار هذه البنية من خلال المثال التوضيحي التالي :

```
#include <iostream>
#include <iomanip>
#include "cylinder.h"    // Cylinder class definition
using namespace std;
int main()
{  Circle circle( 22, 8, 3.5 );  // create a Circle
   Cylinder cylinder(10,10,3.3,10);  // create a Cylinder
   cout<<circle.getArea()<<endl;
   cout<<circle.getVolume()<<endl;
   cout<<cylinder.getArea()<<endl;
   cout<<cylinder.getVolume()<<endl;
   system("pause");   return 0;
} // end main
```

الخرج:

```
38.4845
0
275.769
342.119
Press any key to continue . . .
```

- على الرغم من أنك لا تستطيع إنشاء أغراض من صنف مجرد، فإن بإمكانك إنشاء مؤشرات ومراجع إلى صنف مجرد. وهذا يتيح للصنف المجرد أن يدعم تعددية الأشكال وقت التنفيذ والتي تستند إلى المؤشرات والمراجع إلى الصنف الأساس لاختيار التابع الظاهري المناسب كما في المثال التالي :

```
#include <iostream>
#include <iomanip>
#include "cylinder.h"   // Cylinder class definition
using namespace std;
```

Pure virtual function and abstract classes

```
int main()
{
   Point *pointPtr;  // legal declaration
   Circle circle( 22, 8, 3.5 );    // create a Circle
   Cylinder cylinder(10,10,3.3,10);  // create a Cylinder

   pointPtr=&circle;
   cout<<pointPtr->getArea()<<endl;
   cout<<pointPtr->getVolume()<<endl;

   pointPtr=&cylinder;
   cout<<pointPtr->getArea()<<endl;
   cout<<pointPtr->getVolume()<<endl;
  system("pause");   return 0;
} // end main
```

سيكون نفس الخرج السابق.

انتهت تمارين الأسبوع الثامن