كلية الهندسة – قسم المعلوماتية

مقرر برمجة 2

أ. د. علي عمران سليمان

محاضرات الأسبوع السابع

inheritance 2

الفصل الاول 2025-2026

المحتوى

1. Introduction.
2. Defining inheritance.
3. Create an inheritance relationship between the base class and the derived class.
4. Identify the relationship between inheritance and protected class members.
5. Identify the different forms of inheritance (public, private, protected). Create multiple inheritance.
6. Create multiple inheritance.
7. Knowing the relationship between inheritance and the functions of construction and demolition.
8. Inheriting virtual classes.

1. مقدمة.
2. تعريف الوراثة
3. إنشاء علاقة وراثة بين صنف أساس وصنف مشتق.
4. التعرف على العلاقة بين الوراثة والأعضاء المحمية للصنف.
5. التعرف على الأشكال المختلفة للوراثة (عامة، خاصة، محمية).
6. **إنشاء الوراثة المتعددة.**
7. **معرفة العلاقة بين الوراثة وتوابع البناء والهدم.**
8. **وراثة الأصناف الظاهرية.**

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

- تعرفنا على مفهوم الوراثة الوحيدة single inheritance التي يجري وفقها اشتقاق صنف انطلاقاً من صنف أساس واحد، إلا أنه من الممكن أن نشتق صنفاً من عدة أصناف أساس ونسمي ذلك بالوراثة المتعددة multiple inheritance . يعتبر هذا النوع من الوراثة وسيلة هامة لإعادة استخدام البرمجيات.
- نبين مثل هذا الاستخدام من خلال المثال العام التالي الذي يتم فيه اشتقاق الصنف Derived من صنفين أساس Base1 و Base2:

✓ أولاً – تعريف الصنف الأساس Base1:

```
#ifndef BASE1_H
#define BASE1_H
// class Base1 definition
class Base1 {
public:
   Base1(int parameterValue) {value = parameterValue;}
   int getData() const { return value; }
```

```
protected:        // accessible to derived classes
   int value;     // inherited by derived class
};                // end class Base1
#endif            // BASE1_H
```

ثانياً – تعريف الصنف الأساس Base2:

```
#ifndef BASE2_H
#define BASE2_H
// class Base2 definition
class Base2 { public:
   Base2(char characterData) {letter = characterData;}
   char getData() const { return letter; }
protected:          // accessible to derived classes
   char letter; /* inherited by derived class*/};  // end Base2
#endif  // BASE2_H
```

ثالثاً– تعريف الصنف المشتق Derived

```
#ifndef DERIVED_H
#define DERIVED_H
#include <iostream>
using std::ostream;
#include "Base1.h"
#include "Base2.h"
// class Derived definition
class Derived : public Base1, public Base2 {
   friend ostream &operator<<( ostream &, const Derived & );
public:
   Derived( int, char, double );
   double getReal() const;
```

```
private:    double real;   // derived class's private data
};  // end class Derived
#endif  // DERIVED_H
```

ملف تعريف الصنف DerivedF :

```
//  DerivedF.cpp  Testing class DerivedF.

//#include <iostream>
//#include <iomanip>
#include "Derived.h"
//using namespace std;
/* constructor for Derived calls constructors for class Base1
and class Base2.use member initializers to call base-class
constructors */
```

```
Derived::Derived(int integer,char character,double double1 )
:Base1(integer),Base2(character),real( double1) { }
// return real
double Derived::getReal() const { return real; }
// display all data members of Derived
ostream &operator<<(ostream &output,const Derived &derived)
{   output << "    Integer: " << derived.value
           << "\n  Character: " << derived.letter
           << "\nReal number: " << derived.real;
   return output;   // enables cascaded calls
}  // end operator<<
```

ملف الاختبار main

```
#include <iostream>
```

Multiple inheritance

5-3- الوراثة المتعددة

```
#include <iomanip>
#include "Derived.h"
using namespace std;

int main()
{
 Base1 base1( 10 ), *base1Ptr = 0;     // create Base1 object
   Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
   Derived derived( 7, 'A', 3.5 );     // create Derived object
   // print data members of base-class objects
   cout <<"Object base1 contains integer " << base1.getData()
        <<"\nObject base2 contains character "<<base2.getData()
        << "\nObject derived contains:\n" << derived << "\n\n";
```

```
  // print data members of derived-class object
  // scope resolution operator resolves getData ambiguity
  cout << "Data members of Derived can be"
       << " accessed individually:"
       << "\n    Integer: " << derived.Base1::getData()
       << "\n  Character: " << derived.Base2::getData()
       << "\nReal number: " << derived.getReal() << "\n\n";
  cout << "Derived can be treated as an "
       << "object of either base class:\n";
  // treat Derived as a Base1 object
  base1Ptr = &derived;
cout<<"base1Ptr->getData() yields «
    <<base1Ptr->getData() << '\n';
```

Multiple inheritance

5-3- الوراثة المتعددة

```
// treat Derived as a Base2 object
base2Ptr = &derived;
   cout << "base2Ptr->getData() yields "
        << base2Ptr->getData() << endl;
  system("pause");   return 0;
} // end main
```

الخرج:

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5
```

```
Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
Press any key to continue . . .
```

- لاحظ إن التعبير عن الوراثة المتعددة بأن نذكر أسماء الأصناف الأساس مفصولة عن بعضها البعض بفاصلة.
- إن تابع البناء للصنف Derived يقوم صراحة باستدعاء توابع البناء للأصناف الأساس المكونة له ضمن قائمة الأعضاء التي تقوم بإعطاء قيم ابتدائية.
- يجب استدعاء توابع البناء للأصناف الأساس وفقاً لنفس ترتيب استخدامها أثناء عملية الوراثة.

هناك سؤالان هامان متعلقان بتوابع البناء والهدم لدى إجراء الوراثة.

السؤال الأول: متى يتم استدعاء توابع البناء والهدم للصنف الأساس والصنف المشتق ؟.

السؤال الثاني: كيف يمكن تمرير قيم لبارامترات التابع الباني للصنف الأساس ؟.

1- متى يتم استدعاء توابع البناء والهدم ؟

من الممكن أن يحتوي الصنف الأساس أوالصنف المشتق أو كلاهما تابع بناء وتابع هدم أو أحدهما. من المهم أن نفهم ما هو الترتيب الذي يتم وفقه تنفيذ هذه التوابع عند إنشاء غرض من صنف مشتق ولدى تدميره.

لننظر بداية إلى المثال القصير التالي:

```
// ConsturDeconsturInhe.cpp : main project file.
#include <iostream>
using namespace std;
class Base {
public:       Base() { cout << "Constructing Base\n"; }
              ~Base() { cout << "Destructing Base\n"; }
};
```

```
class Derived: public Base {
public:       Derived() { cout << "Constructing Derived\n"; }
              ~Derived() { cout << "Destructing Derived\n"; }
};
int main()
{      {Derived ob;}       system("pause");     return 0;
} // end main
```

عند تنفيذ هذا البرنامج فإن خرجه يكون على النحو التالي:

```
Constructing Base
Constructing Derived
Destructing Derived
Destructing Base
Press any key to continue . . .
```

كما هو ملاحظ، فإن التابع الباني للصنف base يتم تنفيذه أولاً، ثم يتم تنفيذ التابع الباني للصنف derived (عند البناء). أما عند الهدم، فيتم تابع الهدم للصنف derived ينفذ أولاً ومن ثم يتم تنفيذ تابع الهدم الصنف base. من المهم وضع {} كي يظهر التدمير وإلا سيظهر البناء فقط.

- يمكن تعميم نتيجة التجربة السابقة. عندما يتم إنشاء غرض من صنف مشتق، فإذا كان الصنف الأساس يحتوي على تابع بان فإنه سيتم استدعاؤه أولاً، ومن ثم يتم استدعاء التابع الباني للصنف المشتق. وعندما يتم تدمير غرض من صنف مشتق، فإن تابعه الهادم سيتم استدعاؤه أولاً ومن ثم استدعاء التابع الهادم للصنف الأساس (إن وجد)، بكلام آخر يتم استدعاء التوابع البانية بنفس ترتيب الاشتقاق فيما يتم استدعاء التوابع الهادمة بالترتيب المعاكس.
- إذا حاولنا تفسير هذه الظاهرة، إن التوابع البانية تنفذ بترتيب الاشتقاق، وذلك لأن الصنف الأساس ليس لديه أدنى فكرة عن الأصناف المشتقة، وبالتالي فأي عملية تهيئة يحتاجها منفصلة عن أي تهيئة تتم من قبل الصنف المشتق. وبالتالي يجب أن تنفذ أولاً.
- وبالمثل، فإن توابع الهدم تنفذ بترتيب معاكس للاشتقاق، وذلك لأن الصنف الأساس يقع تحت الصنف المشتق، وبالتالي فإن هدم الغرض الأساس تتطلب هدم الغرض المشتق، وبالتالي فإن تابع الهدم للصنف المشتق يتم استدعاؤه أولاً.
- في حال الوراثة متعددة المستويات (أي عندما يكون الصنف المشتق صنفاً أساس لصنف مشتق آخر)، فإن القاعدة العامة التي تطبق: يتم استدعاء توابع البناء بترتيب الاشتقاق، وتوابع الهدم بترتيب معاكس.

على سبيل المثال، ليكن البرنامج التالي:

```
#include <iostream>
using namespace std;
class Base {
public:
```

```
Base() { cout << "Constructing Base\n"; }
~Base() { cout << "Destructing Base\n"; } };

class Derived1: public Base {
public:
Derived1() { cout << "Constructing Derived1\n"; }
~Derived1() { cout << "Destructing Derived1\n"; } };

class Derived2: public Derived1 {
public:
Derived2() { cout << "Constructing Derived2\n"; }
~Derived2() { cout << "Destructing Derived2\n"; } };

int main()
{      { Derived2 ob; }system("pause");    return 0;
} // end main
```

الخرج:

```
Constructing Base
Constructing Derived1
Constructing Derived2
Destructing Derived2
Destructing Derived1
Destructing Base
Press any key to continue . . .
```

القاعدة العامة نفسها تنطبق في حال الوراثة المتعددة. على سبيل المثال، ليكن البرنامج التالي:

```
#include <iostream>
using namespace std;
class Base1 {
public:      Base1() { cout << "Constructing Base1\n"; }
             ~Base1() { cout << "Destructing Base1\n"; }   };
```

```
class Base2 {
public:
Base2() { cout << "Constructing Base2\n"; }
~Base2() { cout << "Destructing Base2\n"; }   };

class Derived: public Base1, public Base2 {
public:
Derived() { cout << "Constructing Derived\n"; }
~Derived() { cout << "Destructing Derived\n"; }      };

int main()
{
       { Derived ob;}
       system("pause");     return 0;
} // end main
```

فإن خرج البرنامج سيكون:

```
Constructing Base1
Constructing Base2
Constructing Derived
Destructing Derived
Destructing Base2
Destructing Base1
Press any key to continue . . .
```

كما تلاحظ، يتم استدعاء توابع البناء بترتيب الاشتقاق من اليسار إلى اليمين، كما هو محدد في لائحة وراثة الصنف derived. ويتم استدعاء توابع الهدم بترتيب معاكس من اليمين إلى اليسار. وهذا يعني أنه لو تم تحديد base2 قبل base1 في لائحة وراثة الصنف derived كما هو مبين فيما يلي:

```
class derived: public base2, public base1 {
```

فإن خرج البرنامج سيكون:

```
Constructing Base2
Constructing Base1
Constructing Derived
Destructing Derived
Destructing Base1
Destructing Base2
Press any key to continue . . .
```

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

class Base { protected:          int i;
public:      Base(int x) { i=x; cout << "Constructing Base\n"; }
             ~Base() { cout << "Destructing Base\n"; }     };

class Derived: public Base {     int j;
public:      // Derived uses x; y is passed along to base.
      Derived(int x, int y): Base(y)
             { j=x; cout << "Constructing Derived\n"; }
      ~Derived() { cout << "Destructing Derived\n"; }
      void show() { cout << i << " " << j << "\n"; }      };

int main()
{     {Derived ob(3, 4);          ob.show();} // displays 4 3
      system("pause");     return 0;     } // end main
```

يكون الخرج:

```
Constructing Base
Constructing Derived
4 3
Destructing Derived
Destructing Base
Press any key to continue . . .
```

- في هذا المثال، تم التصريح عن التابع الباني للصنف derived على أنه يأخذ بارامترين x وy. في حين أن التابع derived() يأخذ البارامتر x فقط، ويتم تمرير y من خلال التابع base().
- عموماً، يجب أن يتضمن التصريح عن التابع الباني للصنف المشتق كلا من البارامترات التي يحتاجها هو ويحتاجها الصنف الأساس.
- يوضح المثال السابق، أن البارامترات التي يحتاجها الصنف الأساس تم تمريرها في لائحة بارامترات الصنف الأساس المحدد بعد النقطتين.

فيما يلي مثال يستخدم عدة أصناف أساس:

2- تمرير البارامترات إلى توابع البناء للصنف الأساس.

✓ إن أياً من الأمثلة السابقة لم يتضمن توابع بناء تتطلب وسطاء. في حال كون التابع الباني للصنف المشتق يتطلب بارامتراً واحداً أو أكثر، فإنك تستخدم الصيغة القياسية للتابع الباني ذات البارامترات. لكن السؤال المطروح: هل يتم تمرير الوسطاء إلى التابع الباني في الصنف الأساس ؟. الجواب: هو استخدام شكل موسع للتصريح عن التابع الباني للصنف المشتق يقوم بتمرير وسطاء لتابع بان أو أكثر للصنف الأساس.

الشكل العام لهذا التصريح الموسع للتابع الباني للصنف المشتق يكون كمايلي:

```
derived-constructor(arg-list): base1(arg-list),
                               base2(arg-list),
                               // ...
                               baseN(arg-list)
{// body of derived constructor
}
```

حيث base1، base2، ....، baseN هي أسماء الأصناف الاساس التي تمت وراثتها من قبل الصنف المشتق. لاحظ أن النقطتين (:) تفصل التصريح عن التابع الباني للصنف المشتق عن محددات الصنف الاساس، وإن محددات الصنف الأساس مفصولة عن بعضها بفواصل في حال تعدد الأصناف الأساس.

لندرس المثال التالي:

```
#include <iostream>
using namespace std;
class Base1 {protected:          int i;
public:      Base1(int x) { i=x; cout << "Constructing Base1\n"; }
             ~Base1() { cout << "Destructing Base1\n"; }   };

class Base2 {protected:          int k;
public:      Base2(int x) { k=x; cout << "Constructing Base2\n"; }
             ~Base2() { cout << "Destructing Base2\n"; }   };

class Derived: public Base1, public Base2 {   int j;
public:Derived(int x, int y, int z): Base1(y), Base2(z)
       { j=x; cout << "Constructing Derived\n"; }
       ~Derived() { cout << "Destructing Derived\n"; }
       void show() { cout << i << " " << j << " " << k << "\n"; } };
```

```
int main()
{       { Derived ob(3, 4, 5);              ob.show(); // displays 4 3 5
        } system("pause");     return 0;
} //end main
```

الخرج:

```
Constructing Base1
Constructing Base2
Constructing Derived
4 3 5
Destructing Derived
Destructing Base2
Destructing Base1
Press any key to continue . . .
```

- من المهم أن نفهم أن وسطاء التابع الباني للصنف الأساس يتم تمريرها كوسطاء للتابع الباني للصنف المشتق، وبالتالي، فحتى لو لم يكن للتابع الباني للصنف المشتق أي بارامترات فإنه يظل محتاجاً لأن يحتوي على بارمترات إذا تطلب الصنف الأساس ذلك. في هذه الحالة، فإن البارامترات الممررة للصنف المشتق تمرر إلى الصنف الأساس.

3-6- توابع البناء والهدم والوراثة

في البرنامج التالي، ليس للتابع الباني للصنف المشتق أي وسطاء ولكن التابعين base1() و base2() لديهما وسطاء:

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

class Base1 { protected:  int i;
public:Base1(int x) { i=x; cout << "Constructing Base1\n"; }
       ~Base1() { cout << "Destructing Base1\n"; } };

class Base2 { protected:  int k;
public:       Base2(int x) { k=x; cout << "Constructing Base2\n"; }
              ~Base2() { cout << "Destructing Base2\n"; } };

class Derived: public Base1, public Base2 {
public:  /* Derived constructor uses no parameter,but still must be declared
as taking them topass them along to base classes.*/
   Derived(int x, int y): Base1(x),Base2(y){cout<<"Constructing Derived\n"; }
   ~Derived() { cout << "Destructing Derived\n"; }
       void show() { cout << i << " " << k << "\n"; } };
```

```
int main()
{       {Derived ob(3, 4);  ob.show(); // displays 3 4
        }system("pause");     return 0;
}//end main
```

يكون الخرج:

```
Constructing Base1
Constructing Base2
Constructing Derived
3 4
Destructing Derived
Destructing Base2
Destructing Base1
Press any key to continue . . .
```

إن لدى التابع الباني للصنف المشتق الحرية في استخدام أي من البارامترات المصرح عنها ضمنه أو كلها حتى لو كان بعضها يمرر إلى الصنف الأساس وبمعنى آخر فإن تمرير البارامترات إلى الصنف الأساس لا يحول دون استخدامه من قبل الصنف المشتق، على سبيل المثال، إن المقطع التالي لا يتضمن أي خطأ:

```
class derived: public base {
    int j;
public:
// derived uses both x and y and then passes them to base.
    derived( int x, int y): base(x, y)
    { j = x*y; cout << "Constructing derived\n"; }
```

بقي أخيراً أن نذكر بأنه عند نمرير القيم إلى التوابع البانية للصنف الاساس فإن الوسطاء يمكن أن تتضمن أي تعبير صالح بما في ذلك استدعاء التوابع أو المتحولات.

يمكن إيضاح مثل هذه العلاقة بين تابع البناء والهدم والوراثة من خلال مثال الوراثة الذي نعمل عليه منذ بداية الفصل :Point/Circle

```
#include <iostream>
using namespace std;
class Point { public:     Point( int = 0, int = 0 );  // default constructor
   ~Point();             // destructor
private:
```

```
   int x;           // x part of coordinate pair
   int y;           // y part of coordinate pair
}; // end class Point

// default constructor
Point::Point(int xValue,int yValue):x(xValue),y(yValue )
{cout << "Point constructor: "<<endl;}        // end Point constructor

// destructor
Point::~Point()
{   cout << "Point destructor: "<<endl;}      // end Point destructor

class Circle: public Point {
public:                                        // default constructor
   Circle( int = 0, int = 0, double = 0.0 );
   ~Circle();                                  // destructor
private:
```

```
   double radius;                          // Circle's radius
};                                         // end class Circle

// default constructor
Circle::Circle(int xValue,int yValue,double radiusValue):Point(xValue,yValue)
{   cout << "Circle constructor: "<<endl;  radius=radiusValue;
}                                          // end Circle constructor

// destructor
Circle::~Circle()
{ cout<<"Circle destructor: "<<endl;}   // end Point destructor

int main()
{  {   Point p ( 11, 22 );     cout << endl;
   Circle c1( 72, 29, 4.5 );   cout << endl;
   Circle c2( 5, 5, 10 );      cout << endl;}
system("pause");    return 0;  }        //end main
```

3-6-1- Passing Parameters to Base-Class Constructors

6-3- توابع البناء والهدم والوراثة

يعطي هذا البرنامج على خرجه:

```
Point constructor:

Point constructor:
Circle constructor:

Point constructor:
Circle constructor:

Circle destructor:
Point destructor:
Circle destructor:
Point destructor:
Point destructor:
Press any key to continue . . .
```

إن هناك بعض الغموض قد يحصل في البرامج المكتوبة بلغة C++ عند القيام بوراثة عدة أصناف أساس.
مثالاً: لننظر إلى البرنامج غير السليم التالي:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class Base {
public: int i;
};
// Derived1 inherits Base.
class Derived1: public Base {public:int j;
};
// Derived2 inherits Base.
class Derived2: public Base {public:int k;
};
```

```
/* Derived3 inherits both Derived1 and Derived2.This means that there are two
copies of base in derived3! */
class Derived3: public Derived1, public Derived2 {
public:              int sum;     };
int main()
{      Derived3 ob;
       ob.i = 10; // this is ambiguous, which i???
       ob.j = 20;           ob.k = 30;
       // i ambiguous here, too
       ob.sum = ob.i + ob.j + ob.k;
       // also ambiguous, which i?
       cout << ob.i << " ";
       cout << ob.j << " " << ob.k << " ";     cout << ob.sum;
       system("pause");     return 0;
}      //end main
```

سنحصل على الأخطأ التاليه:

```
1>constrDeconPxx.cpp(23): error C2385: ambiguous access of 'i'
1>          could be the 'i' in base 'Base'
1>          or could be the 'i' in base 'Base'
1>constrDeconPxx.cpp(26): error C2385: ambiguous access of 'i'
1>          could be the 'i' in base 'Base'
1>          or could be the 'i' in base 'Base'
1>constrDeconPxx.cpp(28): error C2385: ambiguous access of 'i'
1>          could be the 'i' in base 'Base'
1>          or could be the 'i' in base 'Base'
```

- كما تشير التعليقات ضمن البرنامج، فإن كلا من الصنفين derived1 وderived2 يرثان الصنف base. في حين أن derived3 يرث كلا من الصنفين derived1 وderived2. هذا يعني أن هناك نسختين من الصنف base موجودين في غرض من الصنف derived3. وبالتالي ففي تعبير من الشكل:

```
ob.i = 10;
```

- أي i نقصد، التي تنتمي إلى derived1 أم التي تنتمي إلى derived2 ؟ لأن هناك نسختين من الصنف base موجودتين في الغرض ob فإن هناك عضوين اسمهما i وبالتالي فإن التعبير غامض.

توجد طريقتين لتصحيح البرنامج السابق. الأولى أن نستخدم معامل تحديد المجال scope resolution مع i كما في المثال التالي:

```
int main() { Derived3 ob;
        ob.Derived1::i = 10; // scope resolved, use Derived1's i
        ob.j = 20;              ob.k = 30;
        // scope resolved
        ob.sum = ob.Derived1::i + ob.j + ob.k;
        // also resolved here
        cout << ob.Derived1::i << " ";
        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;
        system("pause");     return 0;
}//end main
```

الخرج:

```
10 20 30 60Press any key to continue . . .
```

كما ترى، نظراً لاستخدام المعامل:: فإن البرنامج اختار يدوياً النسخة base للصنف derived1. إلا أن هذا الاستخدام يفتح الباب أمام موضوع أعمق، ماذا لو كنا بحاجة فعلياً إلى نسخة واحدة فقط من الصنف base، فهل هناك طريقة لمنع وجود نسختين في الصنف derived3 ؟ الجواب هو بكل تأكيد نعم، ويتم تحقيق ذلك من خلال استخدام الأصناف الأساس الظاهرية.

عند اشتقاق غرضين أو أكثر من صنف أساس مشترك فبالإمكان منع وجود عدة نسخ من الصنف الأساس في صنف مشتق من هذه الأغراض من خلال التصريح عن الصنف الأساس على أنه ظاهري virtual عند وراثته. يمكن تحقيق ذلك من خلال وضع الكلمة المفتاحية virtual قبل اسم الصنف الأساس عند وراثته.

المثال التالي يبين نسخة أخرى من المثال السابق:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
class Base { public:      int i; };

// Derived1 inherits Base as virtual.
class Derived1: virtual public Base {  public:      int j;
};

// Derived2 inherits Base as virtual.
class Derived2: virtual public Base {  public:      int k;
};
```

```
/* derived3 inherits both Derived1 and Derived2.This time, there is only one
copy of base class. */
class Derived3: public Derived1, public Derived2 {
public:      int sum;     };

int main()
{
Derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;   ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";cout << ob.j << " " << ob.k << " ";cout << ob.sum;
system("pause");    return 0;
}//end main
```

3-7- الأصناف الأساس الظاهرية

يعطي هذا البرنامج على خرجه:

```
10 20 30 60Press any key to continue . . .
```

وكما ترى فإن الكلمة virtual قد سبقت تحديد الصنف الموروث. وبالتالي فإن كلا من الصنفين derived1 وderived2 قد قاما بوراثة الصنف base ظاهرياً، وبالتالي فإن أي وراثة متعددة منهما ستؤدي إلى احتواء الصنف المشتق على نسخة واحدة من الصنف base. وبالتالي، فإن derived3 هناك نسخة واحدة من الصنف base وبالتالي فإن التعبير ob.i صالح ولا يحوي أي غموض.

Case Study

9-2 –دراسة حالة

أولاً: صنف التاريخ (Date Class)

ليكن المطلوب

- **تطبيق الزيادة على صنف التاريخ من خلال زيادة يوم بشكل سابق ولاحق والذي قد ينعكس على الشهر وعلى العام.**
- **تعميم ذلك لزيادة عدد من الأيام.**

ثانياً: التحميل الزائد لبعض عمليات المصفوفات

نظراً لاستخدام المصفوفات كبديل للمؤشرات لذلك ينتج عن التعامل معها الكثير من المشاكل، ومثال ذلك

- **التجول في المصفوفة والخروج من مجالها نظرا لأن لغة C++.**
- **لا تتأكد من أن الدلائل لا تزال ضمن حدود المصفوفة.**
- **عدم إمكانية إدخال وإخراج مصفوفة دفعة واحدة بل المرور لكل عنصر بعنصر.**
- **المقارنة بين مصفوفتين، وكذلك عند تمرير مصفوفة لتابع يجب تمرير حجمها.**
- **لا يمكن إسناد مصفوفة إلى مصفوفة باستخدام عملية الإسناد.**

والمطلوب استخدام التحميل الزائد من أجل تأمين ذلك.

انتهت تمارين الأسبوع السابع