物件導向程式設計的基本特性

- 封裝(encapsulation)
  - 將物件實例的屬性與方法一起封裝到類別中，也就是說，「封裝」的作用是將物件的功能細節加以隱藏，而只顯示出所提供的功能介面。欲取得類別內的資料，必須透過此功能介面來取得，因此資料對外是隱藏的，是一種資訊隱藏（information hiding）的概念
- 繼承(inheritance)
  - 所謂「繼承」簡單來說就是用原有的類別物件－父類別(superclass)衍生一個新的類別物件－子類別(subclass)，子類別會繼承父類別的屬性與行爲 。利用繼承的方式來遺傳上層的功能及依需要增減其函式 。可以簡化重覆撰寫程式，以及減少錯誤。
- 多形(polymorphism)
  - 多形最直接的定義就是具有繼承關係的不同類別物件，可以對相同名稱的成員函數呼叫，並產生不同的反應結果。

16.1 虛擬函數

我們以一個錯誤的範例來說明什麼是虛擬函數，以及其重要性。下面是父類別 CWin 的定義

```
01  class CWin    // 定義 CWin 類別，在此為父類別
02  {
03    protected:
04      char id;
05      int width, height;
06    public:
07      CWin(char i='D',int w=10, int h=10)   // 父類別的建構元
08      {
09        id=i;
10        width=w;
11        height=h;
12      }
13      void show area()       // 父類別的 show area()函數
14      {
15        cout<<"Window "<<id<<", area = "<< area() <<endl;
16      }
17      int area()    // 傳回視窗物件的面積
18      {
19        return width*height;
20      }
21  };
```


假設有一個 CWin 類別的子類別 CMiniWin，裡面也定義了一個 area()函數，可用來顯示其可用面積。我們假設可用面積為其真實面積的 80%。於是可撰寫出下面 CMiniWin 類別的程式碼：

```
01  class CMiniWin : public CWin  // 定義子類別 CMiniWin
02  {
03    public:
04      CMiniWin(char i,int w,int h):CWin(i,w,h){}  // 子類別的建構元
05
06      int area()
07      {
08        return (int)(0.8*width*height);   // 傳回可用面積
09      }
10  };
```

如果以子類別的物件呼叫 show_area()函數時，因 show_area()會呼叫 area()函數，此時是父類別的 area()函數被呼叫，還是子類別的 area()函數呢？如下面的範例：

```
// prog16_1, 錯誤的範例，未使用虛擬函數
#include <iostream>
#include <cstdlib>
using namespace std;
class CWin    // 定義 CWin 類別，在此為父類別
{
  protected:
    char id;
    int width, height;
  public:
    CWin(char i='D',int w=10, int h=10)   // 父類別的建構元
    {
      id=i;
      width=w;
      height=h;
    }
     void show_area()       // 父類別的 show_area()函數
     {
       cout<<"Window "<<id<<", area = "<< area() <<endl;
     }
     int area()    // 父類別的 area()函數
     {
       return width*height;
     }
};

class CMiniWin : public CWin  // 定義子類別 CMiniWin
{
  public:
    CMiniWin(char i,int w,int h):CWin(i,w,h){}  // 子類別的建構元

    int area()    // 子類別的 area()函數
    {
      return (int)(0.8*width*height);
    }
};

int main(void)
{
  CWin win('A',70,80);            // 建立父類別物件 win
  CMiniWin m_win('B',50,60);      // 建立子類別物件 m_win

  win.show_area();        // 以父類別物件 win 呼叫 show_area()函數
  m_win.show_area();      // 以子類別物件 m_win 呼叫 show_area()函數

  system("pause");
  return 0;
}

/* prog16_1 OUTPUT---

Window A, area = 5600
Window B, area = 3000
-------------------*/
```


- 於 prog16_1 中，編譯器在編譯程式碼時，便把父類別 CWin 裡的 show_area() 和 area()函數連結在一起編譯了，因此呼叫 show_area()函數時，show_area() 所執行的 area()函數是父類別 CWin 裡的 area()，而非子類別裡的 area()。
- 這種 show_area()和 area()函數連結的方式稱為早期連結 (early binding)，或稱靜態連結(static linkage)。
- 要修正這個問題，可以把 area()改為虛擬函數(virtual function)。虛擬函數的好處在於，它可以與呼叫它的函數進行動態連結(dynamic linkage)，或稱為晚期連結(late binding)。

想把 area()宣告成虛擬函數，在其定義之前加上 virtual 這個關鍵字就可以了。如下面的程式碼：

```
01  // prog16_2, 使用虛擬函數來修正錯誤
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  class CWin    // 定義 CWin 類別，在此為父類別
06  {
07    protected:
08      char id;
09      int width, height;
10    public:
11      CWin(char i='D',int w=10, int h=10)   // 父類別的建構元
12      {
13        id=i;
14        width=w;
15        height=h;
16      }
17       void show_area()       // 父類別的 show_area()函數
18       {
19          cout<<"Window "<<id<<", area = "<< area() <<endl;
20       }
21       virtual int area()    // 父類別的 area()函數
```

```
22        {
23          return width*height;
24        }
25  };
26
27  class CMiniWin : public CWin  // 定義子類別 CMiniWin
28  {
29    public:
30      CMiniWin(char i,int w,int h):CWin(i,w,h){}  // 子類別的建構元
31
32      virtual int area()    // 子類別的 area()函數
33      {
34        return (int)(0.8*width*height);
35      }
36  };
37
38  // 將 prog16_1 的主函數 main() 放在這兒

/* prog16_2 OUTPUT-----
Window A, area = 5600
Window B, area = 2400
-------------------------*/
```


16.2 指向基底類別的指標

將 prog16_2 稍做修改，來說明如何將指向父類別物件的指標轉而指向子類別物件。

```
01  // prog16_3,簡單的應用-指向基底類別物件的指標
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  // 將 prog16_2 的 CWin 類別放在這兒
06  // 將 prog16_2 的 CMiniWin 類別放在這兒
07
08  int main(void)
09  {
10    CWin win('A',70,80);        // 建立子類別的物件
11    CMiniWin m_win('B',50,60);
12
13    CWin *ptr=0;              // 宣告指向基底類別(父類別)的指標
14
15    ptr=&win;                 // 將 ptr 指向父類別的物件 win
16    ptr->show_area();         // 以 ptr 呼叫 show_area()函數
17
18    ptr=&m_win;               // 將 ptr 指向子類別的物件 m_win
19    ptr->show_area();         // 以 ptr 呼叫 show_area()函數
```

- 但轉移後的指標無法取用子類別中與父類別不同的成員
- 子類別不可轉移給父類別

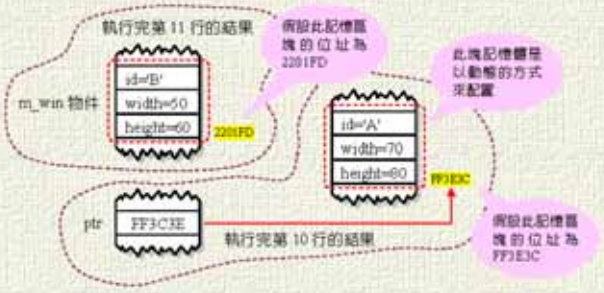
```
20
21      system("pause");
22      return 0;
23  }

/* prog16_3 OUTPUT-----
Window A, area = 5600
Window B, area = 2400
-------------------------*/
```


下面的範例修改自 prog16_3，其中 CWin 與 CMiniWin 類別均與 prog16_3 完全相同，故我們把它省略，只列出 main()主程式：

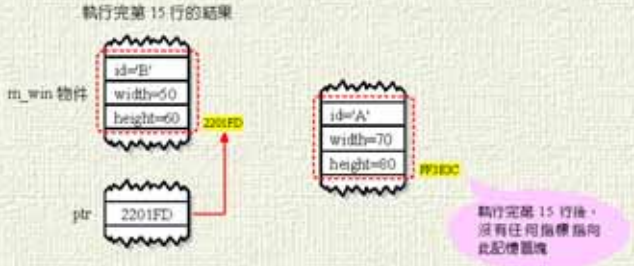
```
01  // prog16_4, 錯誤示範，指向由動態記憶體配置之物件的指標
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  // 將 prog16_3 的 CWin 類別放在這兒
06  // 將 prog16_3 的 CMiniWin 類別放在這兒
07
08  int main(void)
09  {
10      CWin *ptr=new CWin('A',70,80);  // 設定 ptr 指向 CWin 類別的物件
11      CMiniWin m_win('B',50,60);
12
13      ptr->show_area();       // 以 ptr 呼叫 show_area()函數
14
15      ptr=&m_win;             // 將 ptr 指向子類別的物件 m_win
16      ptr->show_area();       // 以 ptr 呼叫 show_area()函數
17
18      delete ptr;             // 清除 ptr 所指向的記憶空間
19
20      system("pause");
21      return 0;
22  }
```

```
/* prog16_4 OUTPUT---
Window A, area = 5600
Window B, area = 2400
-----------------------*/
```

執行完 10~11 行之後的結果



執行完第 15 行之後的結果



要修正這個錯誤，最簡單的方法就是在 ptr 指向的物件不再使用時，先用 delete 將它所佔的記憶體釋放，再將 ptr 指標指向另一個物件 m_win 就可以了，如下面的範例：

```
01  // prog16_5, 修正 prog16_4 的錯誤
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  // 將 prog16_3 的 CWin 類別放在這兒
06  // 將 prog16_3 的 CMiniWin 類別放在這兒
07
08  int main(void)
09  {
10     CWin *ptr=new CWin('A',70,80);  // 設定 ptr 指向 CWin 類別的物件
11     CMiniWin m_win('B',50,60);
12
13     ptr->show_area();       // 以 ptr 呼叫 show_area()函數
14     delete ptr;             // 先釋放 ptr 所指向的記憶空間
15
16     ptr=&m_win;             // 再將 ptr 指向子類別的物件 m_win
17     ptr->show_area();       // 以 ptr 呼叫 show_area()函數
18
19     system("pause");
20     return 0;
21  }
```

```
/* prog16_5 OUTPUT---
Window A, area = 5600
Window B, area = 2400
-------------------*/
```

```
// 假如virtual不見了
#include <iostream>
#include <cstdlib>
using namespace std;

class CWin    // 定義CWin類別，在此爲父類別
{
  protected:
    char id;
    int width, height;
  public:
    CWin(char i='D',int w=10, int h=10)
    {
      id=i;
      width=w;
      height=h;
    }
    void show_area()  // 父類別的show_area()函數
    {
      cout << "Window " << id << ", area = " << area()
      << endl;
    }
    virtual int area()    // 父類別的area()函數
    {
      return width*height;
    }
    void sayHello()
    {
      cout<<"Father!,Hello\n";
    }
};
```

```
class CMiniWin : public CWin
{
  public:
    CMiniWin(char i,int w,int h):CWin(i,w,h){}
    virtual int area()
    {
      return (int)(0.8*width*height);
    }
    void sayHello()
    {
      cout<<"Father!,Hello\n";
    }
};

int main(void)
{
  CWin *ptr=new CWin('A',70,80);
  CMiniWin m_win('B',50,60);
   ptr->show_area();
  delete ptr;

  ptr=&m_win;
  ptr->show_area();
  ptr->sayHello();
  system("pause");
  return 0;
}
```

將呼叫那一個area()？

```
#include <iostream.h>
class CShape
{
 public:
 virtual void display()=0;
};
//-----------------------------------------------
class CEllipse : public CShape
{
 public:
void display() { cout << "Ellipse \n"; }
};
//-----------------------------------------------
class CCircle : public CEllipse
{
 public:
 void display() { cout << "Circle \n"; }
};
//-----------------------------------------------
class CTriangle : public CShape
{
 public:
 void display() { cout << "Triangle \n"; }
};
//-----------------------------------------------
```

```
class CRect : public CShape
{ public:
 void display() { cout << "Rectangle \n"; }
};
//-----------------------------------------------
class CSquare : public CRect
{ public:
 void display() { cout << "Square \n"; }
};
void main()
{
 CShape aShape;     //Error不得建立內含純虛
                    //擬函式的類別物件
 CEllipse aEllipse;
 CCircle aCircle;
 CTriangle aTriangle;
 CRect aRect;
 CSquare aSquare;
 CShape* pShape[6] = {&aShape, &aEllipse,
   &aCircle, &aTriangle, &aRect, &aSquare };
 for (int i=0; i< 6; i++)
    pShape[i]->display();

 CRect *pRect=&aSquare;
 pRect->display();
}
```

```
得到的結果是：
Ellipse
Circle
Triangle
Rectangle
Square
Square
```

虛擬函式的使用時機

- 當你設計一套類別，你並不知道使用者會衍生什麼新的子類別出來。如果動物世界中出現了新品種名曰雅虎，類別使用者勢必在*CAnimal*之下衍生一個*CYahoo*。身爲基礎類別設計者的你，可以利用虛擬函式的特性，將所有動物必定會有的行爲（即規劃一些一般化動作,例如「讓每一種動物發出一聲哮叫」哮叫*roar*），規劃爲虛擬函式。雖然，你在設計基礎類別以及這個一般化動作時，無法掌握使用者自行衍生的子類別，但只要他(子類別)改寫了*roar*這個虛擬函式，你的一般化物件操作動作自然就可以呼叫到該函式。

虛擬函式的使用時機

- 如果你期望衍生類別重新定義一個成員函式，那麼你應該在基礎類別中把此函式設爲*virtual*。
- 以單一指令喚起不同函式，這種性質稱爲Polymorphism，也就是多型。
- 虛擬函式是C++ 語言的Polymorphism性質以及動態繫結的關鍵

```
#include <string.h>
class CEmployee // 職員
{
 private:
 char m_name[30];
 public:
 CEmployee();
 CEmployee(const char* nm)
 { strcpy(m_name, nm); }
 virtual float computePay();
};
//-----------------------------------------------
// 時薪職員是一種職員
class CWage : public CEmployee
{
 private :
 float m_wage;
 float m_hours;
 public :
 CWage(const char* nm) : CEmployee(nm)
 { m_wage = 250.0; m_hours = 40.0; }
 void setWage(float wg) { m_wage = wg; }
 void setHours(float hrs) { m_hours = hrs; }
 virtual float computePay();
};
//-----------------------------------------------
```

```
// 銷售員是一種時薪職員
class CSales : public CWage
{
 private :
      float m_comm;
      float m_sale;
 public :
      CSales(const char* nm) : CWage(nm)
      { m_comm = m_sale = 0.0; }
      void setCommission(float comm)
      { m_comm = comm; }
      void setSales(float sale) { m_sale = sale; }
       virtual float computePay();
};

// 經理也是一種職員
class CManager : public CEmployee
{
      private :
      float m_salary;
      public :
      CManager(const char* nm) : CEmployee(nm)
      { m_salary = 15000.0; }
      void setSalary(float salary) { m_salary = salary; }
       virtual float computePay();
};
```

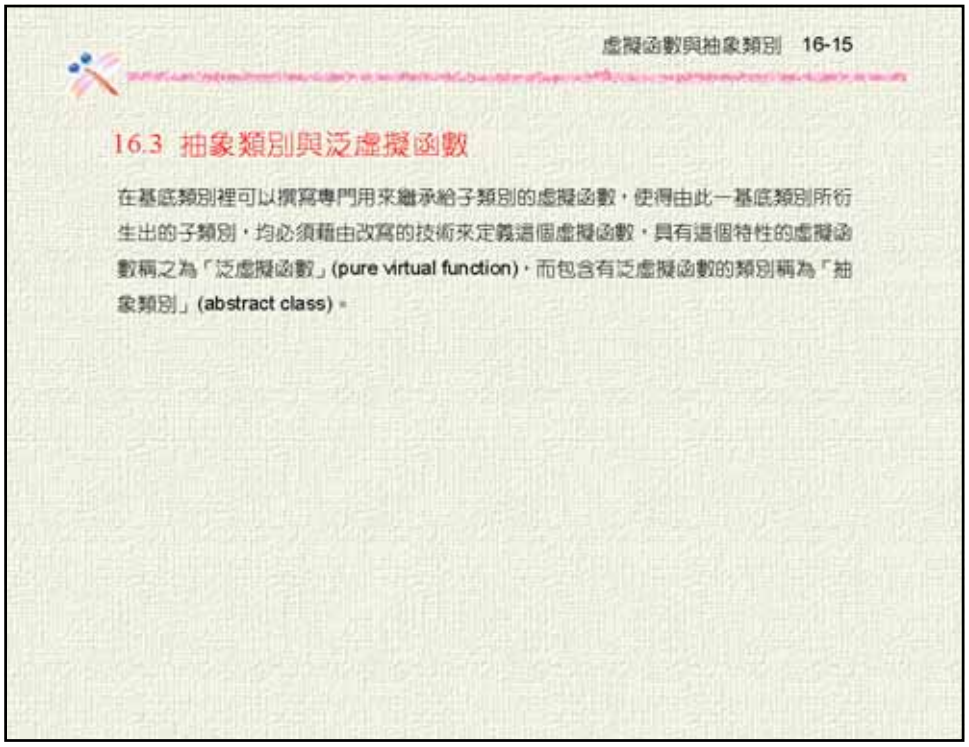
虛擬函式的使用時機

- *pEmp* 指向經理，我希望*pEmp->computePay* 是經理的薪水計算式。
- 當*pEmp* 指向銷售員，我希望*pEmp->computePay* 是銷售員的薪水計算式。
- 多型的功能使用在函數的參數呼叫特別有用：

例：polyphmFun(CEmpolyee * pEmp)

```
{     …
      pEmp->ComputePay();
      …
}
```

```
CSales aSales("周杰倫");
CManager aManager("劉德華");

polymphmFun(&Sales);
polymphmFun(&aManager);
```



純虛擬函式

```
class CShape
{
   public:
   virtual void display() = 0; // 注意"= 0"
};
```

- 純虛擬函式不需定義其實際動作，它的存在只是爲了在衍生類別中被重新定義，只是爲了提供一個多型介面。
- 只要是擁有純虛擬函式的類別，就是一種抽象類別，它是不能夠被具象化（instantiate）的，也就是說，你不能根據它產生一個物件（你怎能說一種形狀爲'Shape' 的物體呢）。如果硬要強渡關山➔CShape obj1;➔會換來這樣的編譯訊息：

  error : illegal attempt to instantiate abstract class.

抽象類別（abstract Class）

- 既然抽象類別中的虛擬函式不打算被呼叫，我們就不應該定義它，應該把它設爲純虛擬函式（在函式宣告之後加上"=0" 即可）。
- 我們可以說，擁有純虛擬函式者爲抽象類別（abstract Class），以別於所謂的具象類別（concrete class)。
- 抽象類別不能產生出物件實體，但是我們可以擁有指向抽象類別之指標，以便於操作抽象類別的各個衍生類別。
- 虛擬函式衍生下去仍爲虛擬函式。
  - *CCircle* 繼承了 *CShape* 之後，如果沒有改寫 *CShape* 中的純虛擬函式，那麼 *CCircle* 本身也就成爲一個擁有純虛擬函式的類別，於是它也是一個抽象類別。

純虛擬函數仍可有函數定義

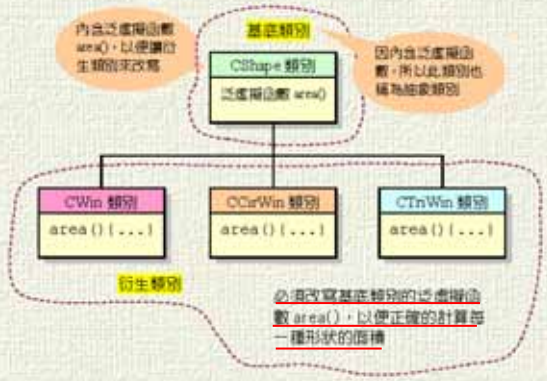
純虛擬函數的目的是要避免基礎類別被宣告成物件,若要提供衍生類別一些內容時,仍可宣告其內容,寫法如下: PureDef.cpp (張)

```
//---- 宣告類別 Shape --------
class Shape
{
private:
  int i;
public:
  Shape(): i(7){}
  ~Shape(){}
  virtual void Rotate() =0;
  virtual void Erase()=0;
  virtual void Center()=0;
};

//---定義純虛擬函數 Center() ----------
void Shape::Center()
  {cout << "Center point\n";}
```

```
//---- 宣告類別 Circle--------
class Circle : public Shape
{
private:
  int r;
public:
  Circle(): r(5)      {}
  Circle(int N): r(N) {}
  ~Circle()           {}
  void Rotate()  {cout<<"將圓形迴轉
     \n";}
  void Erase() {cout<<"把圓形清除\n";}
  void Center(){Shape::Center();}
};
```


16.3.1 定義泛虛擬函數

下圖是由抽象類別 CShape 衍生出子類別 CWin、CCirWin 與 CTriWin 類別的示意圖，圖中可觀察到定義在 CShape 裡的泛虛擬函數 area()，必須在子類別裡做改寫動作：



我們可以在父類別裡把 **area()**函數宣告成**泛虛擬函數**，而把 **area()**處理的方法留在子類別裡來定義。可撰寫出如下的 **CShape** 基底類別程式碼：

```
01  class CShape  // 定義抽象類別 CShape
02  {
03    public:
04      virtual int area()=0; // 定義 area()，並令設之為 0 代表它是泛虛擬函數
05
06      void show_area()     // 定義函數成員 show_area()
07      {
08        cout<<"area = "<< area() <<endl;
09      }
10  };
```


16.3.2 抽象類別的實作

從基底類別衍生的子類別，必須根據基底類別中的泛虛擬函數加以明確的定義，也就是做「**改寫**」的動作，如下面的程式碼：

```
01  class CWin : public CShape  // 定義由 CShape 類別所衍生出的子類別 CWin
02  {
03    protected:
04      int width, height;
05
06    public:
07      CWin(int w=10, int h=10)   // CWin()建構元
08      {
09        width=w;
10        height=h;
11      }
12      virtual int area()
13      {
14        return width*height;
15      }
16  };
```

在此處明確定義 area()的處理方式

prog16_6是抽象類別實作的完整實例，其中CShape是抽象類別，而CWin與CCirWin則是延伸自 CShape 抽象類別的子類別。

```
01  // prog16_6, 抽象類別的實作
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  class CShape  // 定義抽象類別 CShape
06  {
07    public:
08      virtual int area()=0; // 定義 area()，並令之為 0 來代表它是虛擬函數
09
10      void show_area()     // 定義函數成員 show_area()
11      {
12        cout<<"area = "<< area() <<endl;
13      }
14  };
15
16  class CWin : public CShape  // 定義由 CShape 所衍生出的子類別 CWin
17  {
18    protected:
19      int width, height;
```



```
20
21    public:
22      CWin(int w=10, int h=10)   // CWin()建構元
23      {
24        width=w;
25        height=h;
26      }
27      virtual int area()
28      {
29        return width*height;
30      }
31  };
32
33  class CCirWin : public CShape  // 定義由CShape 所衍生出的子類別 CCirWin
34  {
35    protected:
36      int radius;
37
38    public:
39      CCirWin(int r=10)   // CCirWin()建構元
40      {
41        radius=r;
42      }
43      virtual int area()
44      {
45        return (int) (3.14*radius*radius);
46      }
```

在此處明確定義 area()的處理方式

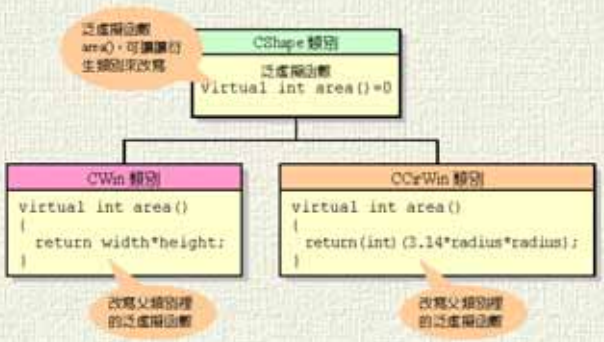
在此處明確定義 area()的處理方式

```
47      void show_area()
48      {
49        cout<<"CCirWin 物件的面積 = "<< area() <<endl;
50      }
51  };
52
53  int main(void)
54  {
55    CWin win1(50,60);          // 建立 CWin 類別的物件 win1
56    CCirWin win2(100);         // 建立 CCinWin 類別的物件 win2
57
58    win1.show_area();          // 用 win1 呼叫 show_area();
59    win2.show_area();          // 用 win2 呼叫 show_area();
60
61    system("pause");
62    return 0;
63  }

/* prog16_6 OUTPUT------
area = 3000
CCirWin 物件的面積 = 31400
-----------------------*/
```

改寫父類別的 show_area()函數

```
CShape *pWin;

pWin=&win2;
pWin->show_area();   ➡ ?

pWin=&win2;
pWin->show_area();   ➡ ?
```


CShape、CWin 與 CCirWin 的 area() 函數之間的關係可由下圖來表示：



16.3.3 使用抽象類別的注意事項

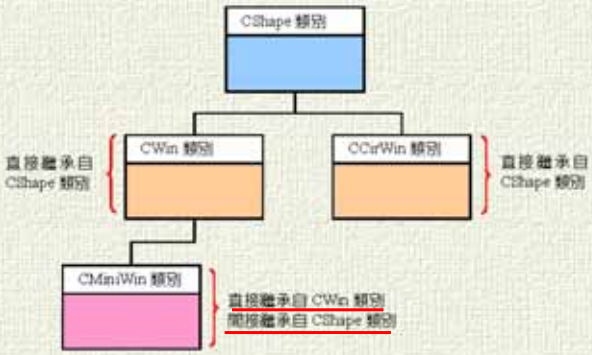
"抽象類別不能用來直接產生物件"，其原因在於它的抽象方法只有宣告，而沒有明確的定義，因此如果用它來建立物件，則物件根本不知要如何使用這個抽象方法。

也就是說，您不能撰寫如下的程式碼：

```
int main(void)
{
    CShape shape;   // 錯誤，不能用抽象類別來產生物件 shape
    ...
}
```


16.4 抽象類別於多層繼承的應用

類別是可以一再被繼承的。基底類別的角色並非一層不變，只要它繼承了某個類別，則此一基底類別就變成了別人的衍生類別了。如下圖所示：



前一節所提的抽象類別也可以應用於多層繼承的架構，我們來看看下面的範例：

```
// prog16_7, 抽象類別於多層繼承的應用
#include <iostream>
#include <cstdlib>
using namespace std;
class CShape  // 定義抽象類別 CShape
{
  public:
    virtual int area()=0;  // 定義 area()為泛虛擬函數

    void show_area()     // 定義函數成員 show_area()
    {
       cout<<"area = "<< area() <<endl;
    }
};

class CWin : public CShape  // 定義由 CShape 所衍生出的子類別 CWin
{
  protected:
    int width, height;

  public:
```


```
    CWin(int w=10, int h=10)   // CWin()建構元
    {
      width=w;
      height=h;
    }
    virtual int area()
    {
      return width*height;
    }
    void show_area()
    {
      cout<<"CWin 物件的面積 = "<< area() <<endl;
    }
};

class CCirWin : public CShape // 定義由 CShape 所衍生出的子類別 CCirWin
{
  protected:
    int radius;

  public:
    CCirWin(int r=10)   // CCirWin()建構元
    {
      radius=r;
    }
```

```
47      virtual int area()
48      {
49        return (int) (3.14*radius*radius);
50      }
51      void show_area()
52      {
53        cout<<"CCirWin 物件的面積 = "<< area() <<endl;
54      }
55  };
56
57  class CMiniWin : public CWin  // 定義由 CWin 所衍生出的子類別 CMiniWin
58  {
59    public:
60      CMiniWin(int w,int h):CWin(w,h){}  // 子類別的建構元
61
62      virtual int area()
63      {
64        return (int) (0.5*width*height);
65      }
66      void show area()
67      {
68        cout<<"CMiniWin 物件的面積 = "<< area() <<endl;
69      }
70  };
71
```



```
72  int main(void)
73  {
74    CWin win1(50,60);       // 建立子類別的物件
75    CCirWin win2(100);
76    CMiniWin win3(50,60);
77
78    win1.show_area();
79    win2.show_area();
80    win3.show_area();
81
82    system("pause");
83    return 0;
84  }

/* prog16_7 OUTPUT------
CWin 物件的面積 = 3000
CCirWin 物件的面積 = 31400
CMiniWin 物件的面積 = 1500
-----------------------*/
```

```
CShape *pWin;

pWin=&win2;
pWin->show_area();   ➡ ?

pWin=&win2;
pWin->show_area();   ➡ ?

pWin=&win2;
pWin->show_area();   ➡ ?
```

16.5 虛擬解構元

下面的範例修改自 **prog16_7**，所不同的是，拿掉了 **CCirWin** 類別以化簡程式碼，並在每一個類別內加入了解構元，用以追蹤解構元呼叫的情形：

```
01  // prog16_8, 錯誤的範例,虛擬函數與解構元
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  class CShape  // 定義抽象類別 CShape
06  {
07    public:
08      virtual int area()=0;  // 定義 area()為純虛擬函數
09      void show_area()
10      {
11        cout<<"area = "<< area() <<endl;
12      }
13      ~CShape()   // ~CShape() 解構元
14      {
15        cout<<"~CShape()解構元被呼叫了..."<<endl;
16        system("pause");
17      }
18  };
19
20  class CWin : public CShape  // 定義由 CShape 所衍生出的子類別 CWin
```


```
21  {
22    protected:
23      int width, height;
24
25    public:
26      CWin(int w=10, int h=10):width(w),height(h){} // CWin()建構元
27
28      virtual int area() {return width*height; }
29
30      void show_area() {
31        cout<<"CWin 物件的面積 = "<< area() <<endl;
32      }
33      ~CWin()  // ~CWin() 解構元
34      {
35        cout<<"~CWin()解構元被呼叫了..."<<endl;
36        system("pause");
37      }
38  };
39
40  class CMiniWin : public CWin  // 定義由 CWin 所衍生出的子類別 CMiniWin
41  {
42    public:
43      CMiniWin(int w,int h):CWin(w,h){}  // CMiniWin()建構元
44
45      virtual int area() {
46        return (int) (0.5*width*height);
47      }
48      void show_area(){
49        cout<<"CMiniWin 物件的面積 = "<< area() <<endl;
50      }
```

```
51      ~CMiniWin()   // ~CMiniWin() 解構元
52      {
53         cout<<"~CMiniWin()解構元被呼叫了..."<<endl;
54         system("pause");
55      }
56  };
57
58  int main(void)
59  {
60     CShape *ptr=new CWin(50,60);
61     ptr->show_area();
62     cout << "銷毀 CWin 物件..." << endl;
63     delete ptr;
64     cout << endl;
65
66     ptr=new CMiniWin(50,50);
67     ptr->show_area();
68     cout << "銷毀 CMiniWin 物件..." << endl;
69     delete ptr;
70     cout << endl;
71
72     CMiniWin m_win(100,100);
73     m_win.show_area();
74
75     system("pause");
76     return 0;
77  }
```



```
/* prog16_8 OUTPUT------
area = 3000                          ——— 抽象類別 CShape 的 show_area()函數被呼叫了
銷毀 CWin 物件...
~CShape()解構元被呼叫了...           ——— 63 行的執行結果
請按任意鍵繼續 . . .

area = 1250                          ——— 抽象類別 CShape 的 show_area()函數被呼叫了
銷毀 CMiniWin 物件...
~CShape()解構元被呼叫了...           ——— 69 行的執行結果
請按任意鍵繼續 . . .

CMiniWin 物件的面積 = 5000
請按任意鍵繼續 . . .
~CMiniWin()解構元被呼叫了...
請按任意鍵繼續 . . .                 } 自動處理物件的銷毀，此時會先執行
~CWin()解構元被呼叫了...               自己的解構元再執行父類別的解構
請按任意鍵繼續 . . .                   元，最後再執行基底類別的解構元
~CShape()解構元被呼叫了...
請按任意鍵繼續 . . .
*/
```

如果基底類別的函數或解構元不是虛擬時，則以指向基底類別型態的指標呼叫函數或銷毀指標所指向的物件時，則只會執行基底類別的函數或解構元，而其衍生類別裡「改寫」父類別的函數或解構元則永遠不會被呼叫到。

要解決前述的問題，把基底類別的解構元改成虛擬解構元就可以了，如下面的程式：

```
01  // prog16_9, 使用虛擬解構元
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  class CShape  // 定義抽象類別 CShape
06  {
07     public:
08        virtual int area()=0;          // 定義 area()為泛虛擬函數
09        virtual void show_area()       // 定義 show_area()為虛擬函數
10        {
11           cout<<"area = "<< area() <<endl;
12        }
13        virtual ~CShape()   // 定義 ~CShape() 為虛擬解構元
14        {
15           cout<<"~CShape()解構元被呼叫了..."<<endl;
16           system("pause");
17        }
18  };
19
20  // 將 prog16_8的 CWin 類別放在這兒
21  // 將 prog16_8的 CMiniWin 類別放在這兒
22  // 將 prog16_8的 main()主程式放在這兒
```

虛擬解構函數（virtual destructor）

- 在動態配置 (dynamic memory allocation) 物件，必需將解構函數改成虛擬解構函數 (virtual destructor) 才能順利完整的回收記憶體資源。
- 將解構函數虛擬化後，才能夠依序執行所有基礎類別和衍生類別的解構函數，完全釋放所有的記憶空間。
- 一般而言，如果在基礎類別內宣告虛擬成員函數，就必需同時宣告虛擬解構函數。
- 建構函數不需要虛擬化。