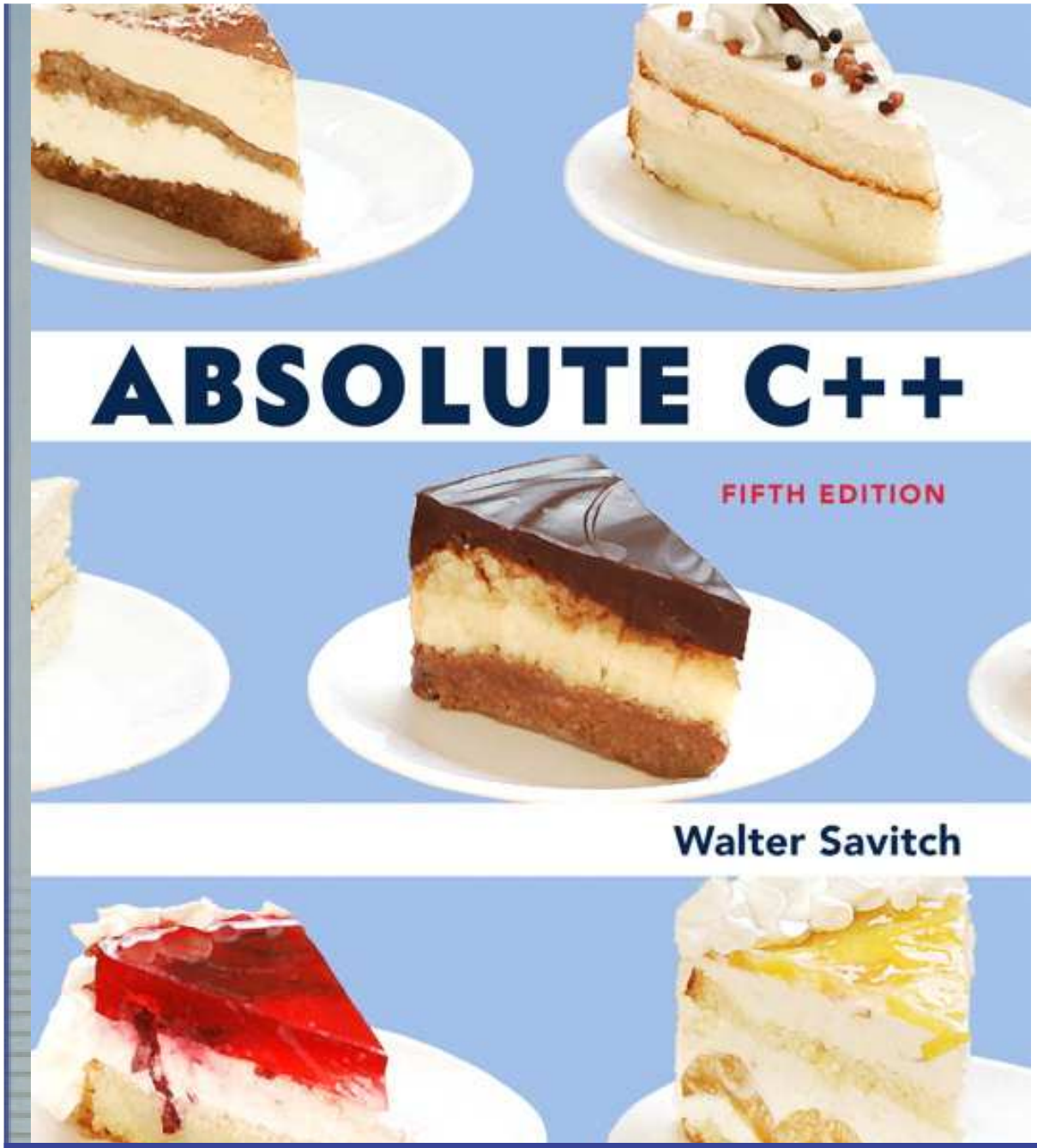


Chapter 15

Polymorphism and Virtual Functions

Learning Objectives

- Virtual Function Basics
  - Late binding
  - Implementing virtual functions
  - When to use a virtual function
  - Abstract classes and pure virtual functions
- Pointers and Virtual Functions
  - Extended type compatibility
  - Downcasting and upcasting
  - C++ "under the hood" with virtual functions

Early binding (Wrong example) (1/2)

```
01  // prog17_1, 錯誤的範例，未使用虛擬函數
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       }
```

```
/* prog17_1 OUTPUT------

Window A, area = 5600
Window B, area = 3000

----------------------*/
```

Early binding (Wrong example) (2/2)

```
17        void show area()              // 父類別的 show area() 函數
18        {
19           cout << "Window " << id << ", area = " << area() << endl;
20        }
21        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        int area()                    // 子類別的 area() 函數
33        {
34           return (int)(0.8*width*height);
35        }
36  };
37
38  int main(void)
39  {
40     CWin win('A',70,80);             // 建立父類別物件 win
41     CMiniWin m win('B',50,60);       // 建立子類別物件 m win
42
43     win.show area();          // 以父類別物件 win 呼叫 show area() 函數
44     m win.show area();        // 以子類別物件 m win 呼叫 show area() 函數
45
46     system("pause");
47     return 0;
48  }
```

```
/* prog17_1 OUTPUT------

Window A, area = 5600
Window B, area = 3000

-----------------------*/
```

Virtual Function Basics

- Polymorphism
  - Associating many meanings to one function
  - Virtual functions provide this capability
  - Fundamental principle of object-oriented programming!
- Virtual
  - Existing in "essence" though not in fact
- Virtual Function
  - Can be "used" before it's "defined"

Virtual Function

- Tells compiler:
  - "Don’t know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"
- Called late binding or dynamic binding
  - Virtual functions implement late binding
  - Decide which function will be called on execution time, not compiling time

Modify using virtual function (1/2)

```
01  // prog17_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;                                      /* prog17_2 OUTPUT-----
14        width=w;                                   Window A, area = 5600
15        height=h;                                  Window B, area = 2400
16      }                                            ----------------------*/
17       void show area()                  // 父類別的 show area()函數
18       {
19          cout << "Window " << id << ", area = " << area() << endl;
20       }
```

Modify using virtual function (2/2)

```
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 prog17_1 的主函數 main()放在這兒
```

```
/* prog17_2 OUTPUT-----
Window A, area = 5600
Window B, area = 2400
----------------------*/
```

Figures Example

- Best explained by example:
- Classes for several kinds of figures
  - Rectangles, circles, ovals, etc.
  - Each figure an object of different class
    - Rectangle data: height, width, center point
    - Circle data: center point, radius
- All derive from one parent-class: Figure
- Require function: draw()
  - Different instructions for each figure

Figures Example 2

- Each class needs different *draw* function
- Can be called "draw" in each class, so:

```
Rectangle r;
Circle c;
r.draw();  //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
```

- Nothing new here yet…

Figures Example: center()

- Parent class Figure contains functions that apply to "all" figures; consider: center(): moves a figure to center of screen
  - Erases 1st, then re-draws
  - So Figure::center() would use function draw() to re-draw
  - Complications!
    - Which draw() function?
    - From which class?

Figures Example: New Figure

- Consider new kind of figure comes along:
  Triangle class
  derived from Figure class
- Function center() inherited from Figure
  - Will it work for triangles?
  - It uses draw(), which is different for each figure!
  - It will use Figure::draw() → won't work for triangles
- Want inherited function center() to use function Triangle::draw() NOT function Figure::draw()
  - But class Triangle wasn't even WRITTEN when Figure::center() was! Doesn't know "triangles"!

Figures Example: Virtual!

- Virtual functions are the answer
- Tells compiler:
  - "Don’t know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"
- Called late binding or dynamic binding
  - Virtual functions implement late binding

Virtual Functions: Another Example

- Bigger example best to demonstrate
- Record-keeping program for automotive parts store
  - Track sales
  - Don't know all sales yet
  - 1st only regular retail sales
  - Later: Discount sales, mail-order, etc.
    - Depend on other factors besides just price, tax

Virtual Functions: Auto Parts

- Program must:
  - Compute daily gross sales
  - Calculate largest/smallest sales of day
  - Perhaps average sale for day
- All come from individual bills
  - But many functions for computing bills will be added "later"!
    - When different types of sales added!
- So function for "computing a bill" will be virtual!

Class Sale Definition

- class Sale
  {
  public:
  Sale();
  Sale(double thePrice);
  double getPrice() const;
  virtual double bill() const;
  double savings(const Sale& other) const;
  private:
  double price;
  };

Member Functions savings and operator <

- double Sale::savings(const Sale& other) const
  {
  return (bill() – other.bill());
  }
- bool operator < (const Sale& first,
  const Sale& second)
  {
  return (first.bill() < second.bill());
  }
- Notice BOTH use member function bill()!

Class Sale

- Represents sales of single item with no added discounts or charges.
- Notice reserved word "virtual" in declaration of member function *bill*
  - Impact: Later, derived classes of Sale can define THEIR versions of function bill
  - Other member functions of Sale will use version based on object of derived class!
  - They won't automatically use Sale's version!

Derived Class DiscountSale Defined

- class DiscountSale : public Sale
  {
  public:
  DiscountSale();
  DiscountSale(double thePrice,
  double the Discount);
  double getDiscount() const;
  void setDiscount(double newDiscount);
  double bill() const;
  private:
  double discount;
  };

DiscountSale's Implementation of bill()

- double DiscountSale::bill() const
  {
  double fraction = discount/100;
  return (1 – fraction)*getPrice();
  }
- Qualifier "virtual" does not go in actual function definition
  - "Automatically" virtual in derived class
  - Declaration (in interface) not required to have "virtual" keyword either (but usually does)

DiscountSale's Implementation of bill()

- Virtual function in base class:
  - "Automatically" virtual in derived class
- Derived class declaration (in interface)
  - Not required to have "virtual" keyword
  - But typically included anyway, for readability

Derived Class DiscountSale

- DiscountSale's member function bill() implemented differently than Sale's
  - Particular to "discounts"
- Member functions *savings* and "<"
  - Will use this definition of bill() for all objects of DiscountSale class!
  - Instead of "defaulting" to version defined in Sales class!

Virtual: Wow!

- Recall class Sale written long before derived class DiscountSale
  - Members savings and "<" compiled before even had ideas of a DiscountSale class
- Yet in a call like:
  DiscountSale d1, d2;
  d1.savings(d2);
  - Call in savings() to function bill() knows to use definition of bill() from DiscountSale class
- Powerful!

Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to "wait" until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle!

Overriding

- Virtual function definition changed in a derived class
  - We say it's been "overidden"
- Similar to redefined
  - Recall: for standard functions
- So:
  - Virtual functions changed: ***overridden***
  - Non-virtual functions changed: ***redefined***

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

DYNAMIC OBJECT POINTER

Pointer points to base class (1/2)

- A pointer points to base class can be pointed to the objects which created by its derived class

```
01  // prog17_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 prog17_2 的 CWin 類別放在這兒
06  // 將 prog17_2 的 CMiniWin 類別放在這兒
07
08  int main(void)
09  {
10     CWin win('A',70,80);
11     CMiniWin m_win('B',50,60); // 建立子類別的物件
12
```

```
/* prog17_3 OUTPUT-----
Window A, area = 5600
Window B, area = 2400
----------------------*/
```

Pointer points to base class (2/2)

```
13      CWin *ptr=NULL;                 // 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
20
21      system("pause");
22      return 0;
23  }
```

```
/* prog17_3 OUTPUT-----
Window A, area = 5600
Window B, area = 2400
----------------------*/
```

Incorrect example for dynamic pointer (1/3)

```
01  // prog17_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 prog17_3 的 CWin 類別放在這兒
06  // 將 prog17_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
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  }
```

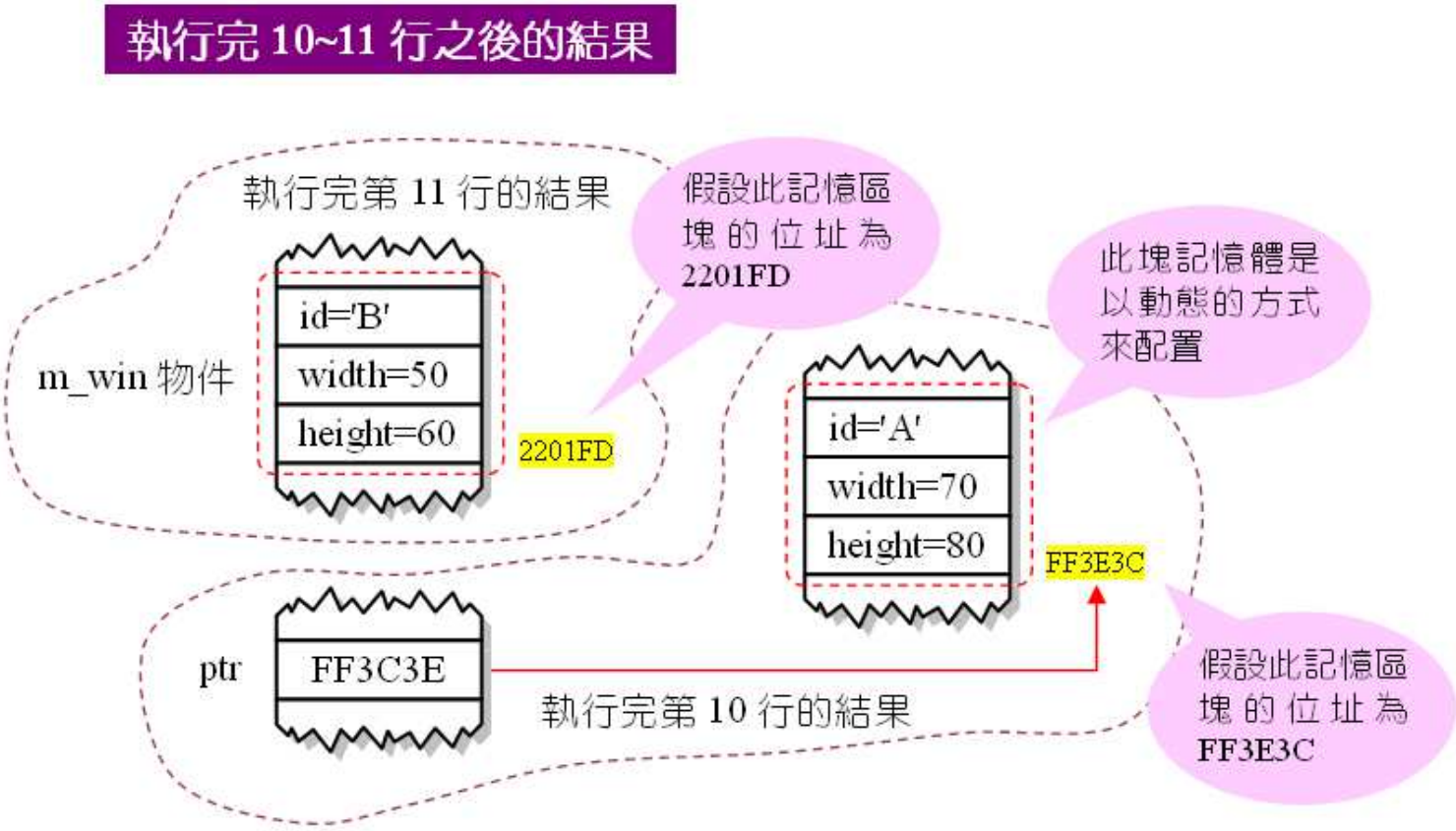
```
/* prog17_4 OUTPUT----

Window A, area = 5600
Window B, area = 2400

----------------------*/
```

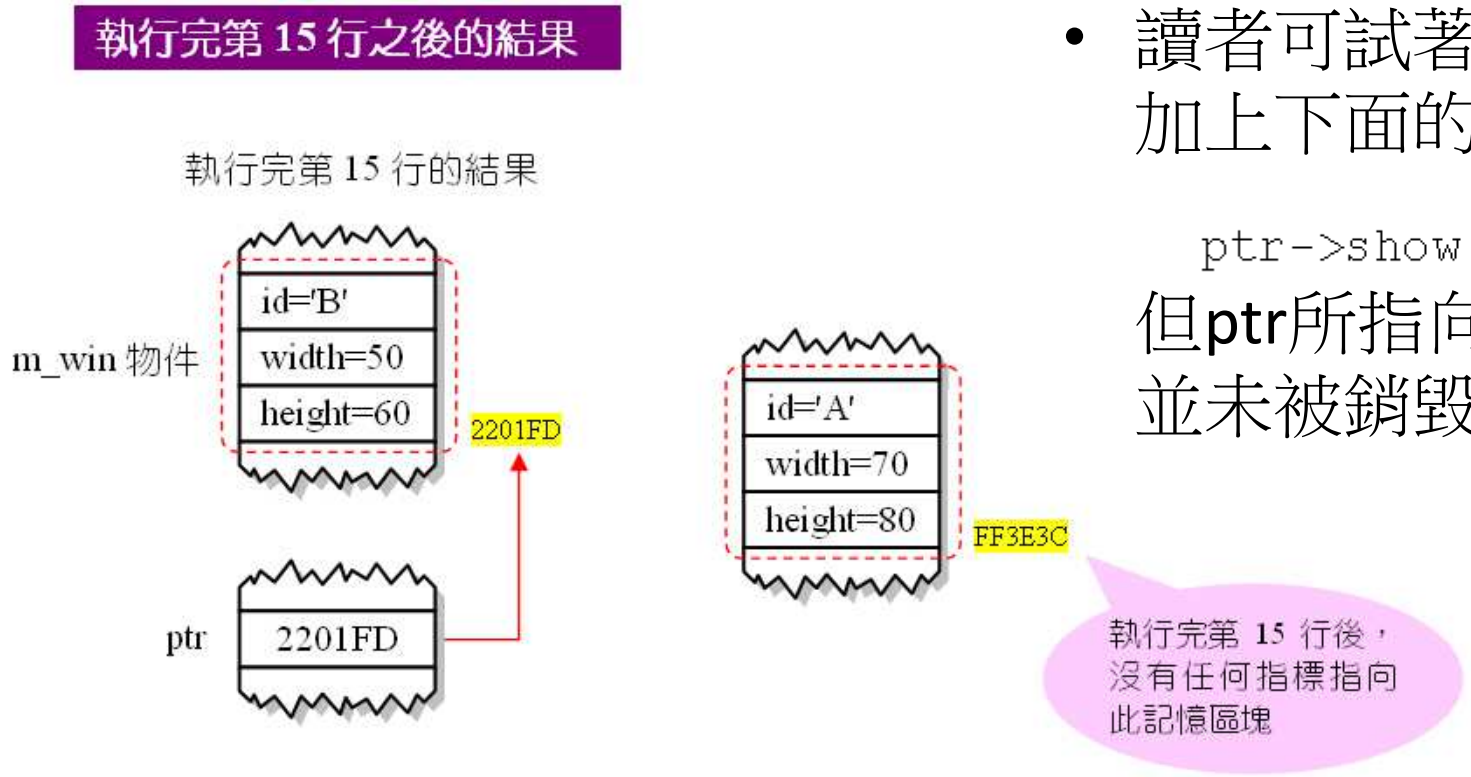
Incorrect example for dynamic pointer (2/3)

```
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);
```



Incorrect example for dynamic pointer (3/3)

```
15      ptr=&m_win;    // 將 ptr 指向子類別的物件 m_win
```



- 讀者可試著在第19行加上下面的敘述

```
ptr->show_area();
```

但ptr所指向的物件並未被銷毀

Modify Incorrect example

```
01  // prog17_5, 修正 prog17_4 的錯誤
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05  // 將 prog17_3 的 CWin 類別放在這兒
06  // 將 prog17_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  }
```

```
/* prog17_5 OUTPUT----

Window A, area = 5600
Window B, area = 2400

----------------------*/
```

VIRTUAL FUNCTION & ABSTRACT BASE CLASS

Pure Virtual Functions

- Base class might not have "meaningful" definition for some of it's members!
  - It's purpose solely for others to derive from
- Recall class Figure
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw!
- Make it a pure virtual function:
  virtual void draw() = 0;

Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: **abstract base class**
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn't have complete "definitions" of all it's members!
- If derived class fails to define all pure's:
  - It's an abstract base class too

Abstract class (1/4)

```
01  // prog17_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 來代表它是泛虛擬函數
09
10       void show_area()          // 定義成員函數 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
```

```
/* prog17_6 OUTPUT-------
area = 3000
CCirWin 物件的面積 = 31400
------------------------*/
```

Abstract class (2/4)

```
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         }
```

在此處明確定義 area()的處理方式

```
/* prog17_6 OUTPUT-------
area = 3000
CCirWin 物件的面積 = 31400
------------------------*/
```

Abstract class (3/4)

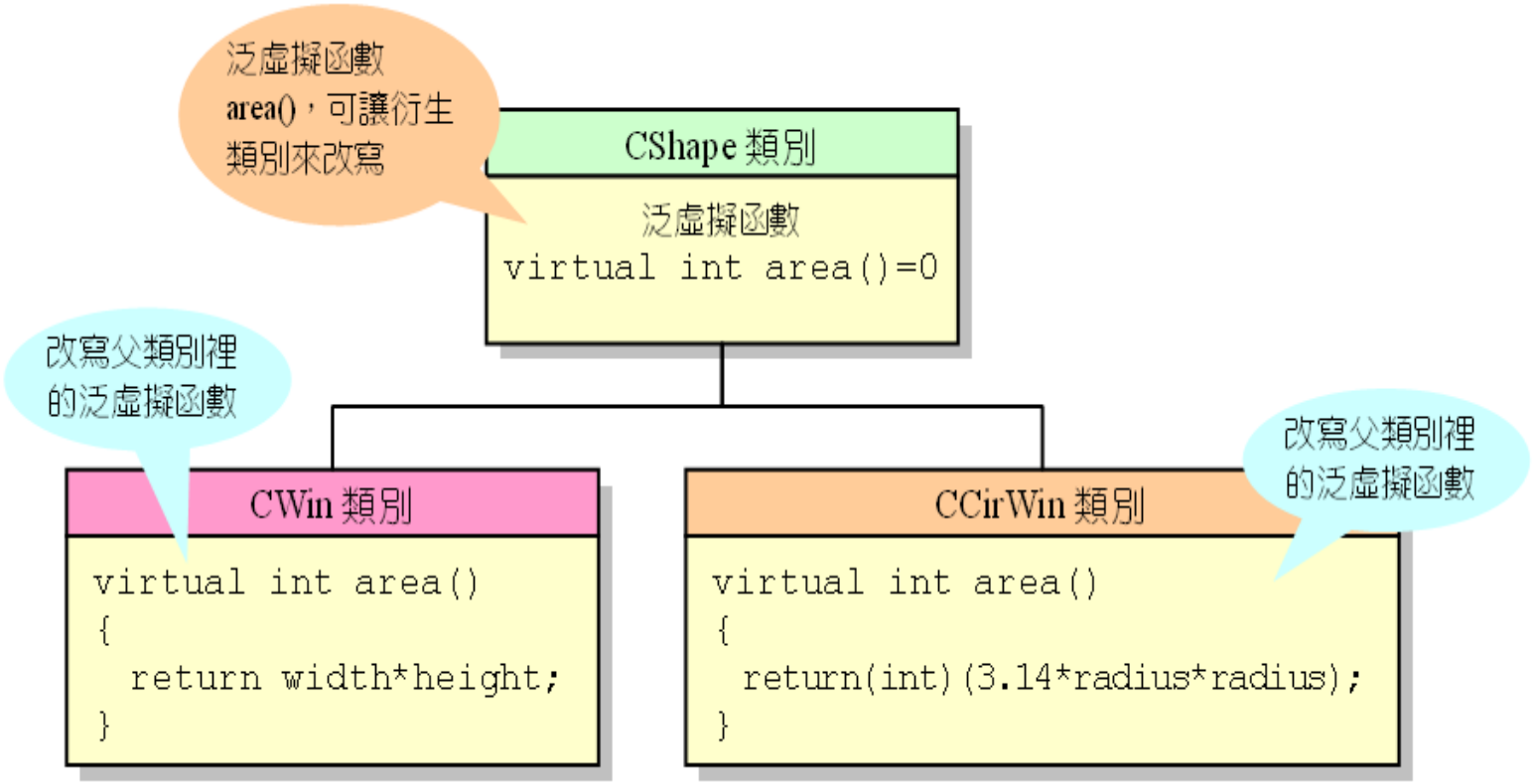
```
43        virtual int area()
44        {
45           return (int) (3.14*radius*radius);
46        }
47        void show area()
48        {
49           cout << "CCirWin 物件的面積 = " << area() <<endl;
50        }
51    };
52
53    int main(void)
54    {
55       CWin win1(50,60);          // 建立 CWin 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    }
```

在此處明確定義 area()的處理方式

改寫父類別的 show_area()函數

```
/* prog17_6 OUTPUT-------
area = 3000
CCirWin 物件的面積 = 31400
------------------------*/
```

Abstract class (4/4)

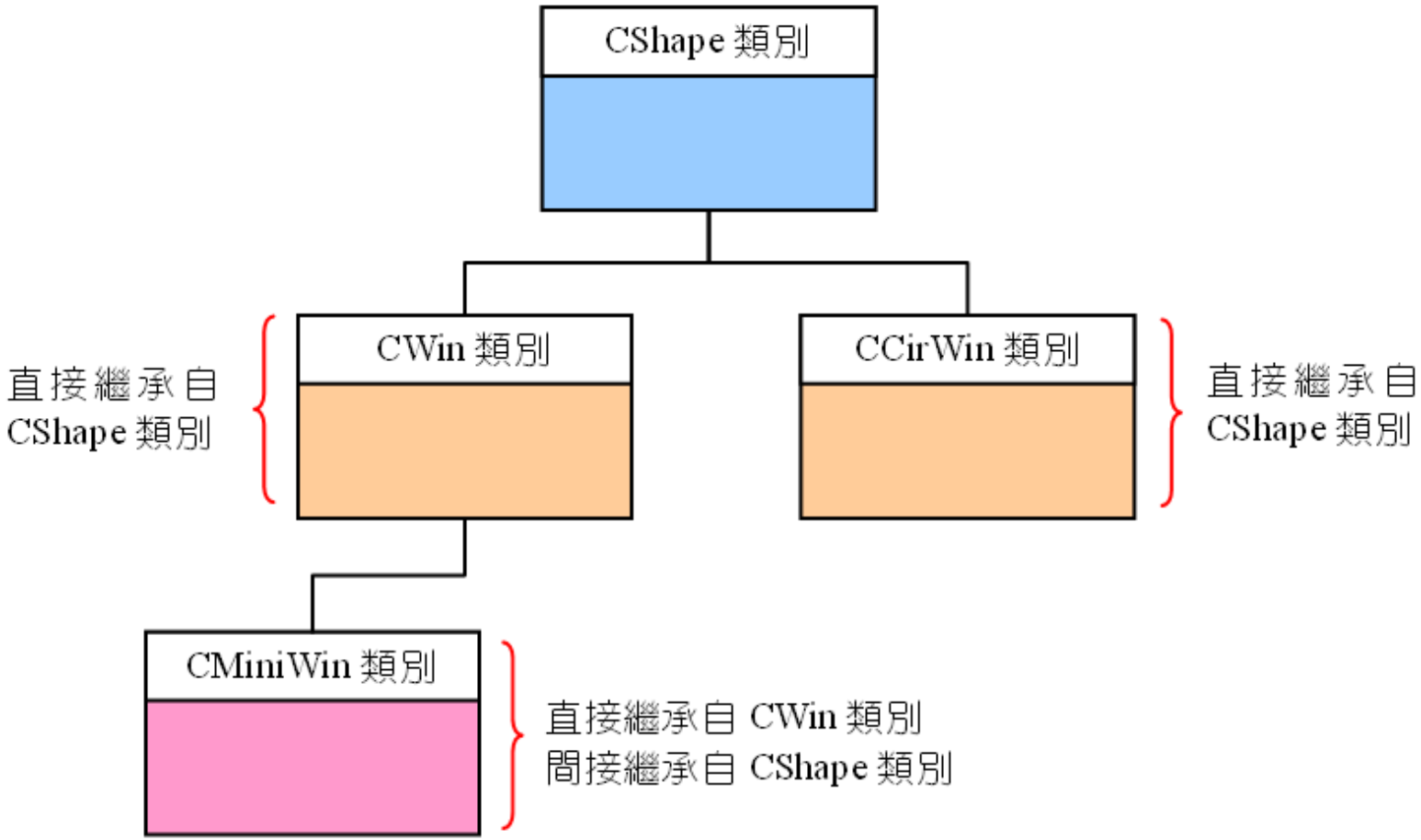


Notice

- No objects can ever be created from it
  - Since it doesn’t have complete "definitions" of all it’s members!

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

Relational figure of inheritance



Abstract class & multi-inheritance (1/4)

```
01  // prog17_7, 抽象類別於多層繼承的應用
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
10        void show area()          // 定義成員函數 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
```

```
/* prog17_7 OUTPUT------
CWin 物件的面積 = 3000
CCirWin 物件的面積 = 31400
CMiniWin 物件的面積 = 1500
-----------------------*/
```

Abstract class & multi-inheritance (2/4)

```
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        void show area()
32        {
33           cout << "CWin物件的面積 = " << area() << endl;
34        }
35  };
36
37  class CCirWin : public CShape  // 定義由CShape所衍生出的子類別CCirWin
38  {
39     protected:
40        int radius;
41
42     public:
43        CCirWin(int r=10)          // CCirWin()建構元
44        {
45           radius=r;
46        }
```

```
/* prog17_7 OUTPUT------
CWin物件的面積 = 3000
CCirWin物件的面積 = 31400
CMiniWin物件的面積 = 1500
-----------------------*/
```

Abstract class & multi-inheritance (3/4)

```
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
```

```
/* prog17_7 OUTPUT------
CWin 物件的面積 = 3000
CCirWin 物件的面積 = 31400
CMiniWin 物件的面積 = 1500
-----------------------*/
```

Abstract class & multi-inheritance (4/4)

```
72  int main(void)
73  {
74     CWin win1(50,60);
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  }
```

```
/* prog17_7 OUTPUT------
CWin 物件的面積 = 3000
CCirWin 物件的面積 = 31400
CMiniWin 物件的面積 = 1500
----------------------*/
```

Extended Type Compatibility

- Given:
  Derived is derived class of Base
  - Derived objects can be assigned to objects of type Base
  - But NOT the other way!
- Consider previous example:
  - A DiscountSale "is a" Sale, but reverse not true

Extended Type Compatibility Example

```
class Pet
{
public:
        string name;
        virtual void print() const;
};
class Dog : public Pet
{
public:
        string breed;
        virtual void print() const;
};
```

Classes Pet and Dog

- Now given declarations:
  Dog vdog;
  Pet vpet;
- Notice member variables name and breed are public!
  - For example purposes only! Not typical!

Using Classes Pet and Dog

- Anything that "is a" dog "is a" pet:
  - vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;
  - These are allowable
- Can assign values to parent-types, but not reverse
  - A pet "is not a" dog (not necessarily)

Slicing Problem

- Notice value assigned to vpet "loses" it's breed field!
  - cout << vpet.breed;
    - Produces ERROR msg!
  - Called slicing problem
- Might seem appropriate
  - Dog was moved to Pet variable, so it should be treated like a Pet
    - And therefore not have "dog" properties
  - Makes for interesting philosphical debate

Slicing Problem Fix

- In C++, slicing problem is nuisance
  - It still "is a" Great Dane named Tiny
  - We'd like to refer to it's breed even if it's been treated as a Pet
- Can do so with pointers to dynamic variables

Slicing Problem Example

- Pet *ppet;
  Dog *pdog;
  pdog = new Dog;
  pdog->name = "Tiny";
  pdog->breed = "Great Dane";
  ppet = pdog;
- Cannot access breed field of object pointed to by ppet:
  cout << ppet->breed; //ILLEGAL!

Slicing Problem Example

- Must use virtual member function:
  ppet->print();
  - Calls print member function in Dog class!
    - Because it's virtual
  - C++ "waits" to see what object pointer ppet is actually pointing to before "binding" call

VIRTUAL DESTRUCTOR

Virtual Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:

  Base *pBase = new Derived;

  …

  delete pBase;
  - Would call base class destructor even though pointing to Derived class object!
  - Making destructor ***virtual*** fixes this!
- Good policy for all destructors to be virtual

Incorrect destructor usage (1/4)

```
01  // prog17_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()為泛虛擬函數
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;
```

Incorrect destructor usage (2/4)

```
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         }
```

Incorrect destructor usage (3/4)

```
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
```

Incorrect destructor usage (4/4)

```
72      CMiniWin m win(100,100);
73      m win.show area();
74
75      system("pause");
76      return 0;
77  }
```

```
/* prog17_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 . . .
-------------------------*/
```

Modified version using virtual destructor (1/2)

- prog17_9是使用虛擬解構元的範例

```
01  // prog17_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prog17_8的CWin類別放在這兒
21  // 將prog17_8的CMiniWin類別放在這兒
22  // 將prog17_8的main()主程式放在這兒
```

Modified version using virtual destructor (2/2)

```
/* prog17_9 OUTPUT---------
CWin 物件的面積 = 3000
銷毀 CWin 物件...
~CWin()解構元被呼叫了...
請按任意鍵繼續 . . .
~CShape()解構元被呼叫了...
請按任意鍵繼續 . . .

CMiniWin 物件的面積 = 1250
銷毀 CMiniWin 物件...
~CMiniWin()解構元被呼叫了...
請按任意鍵繼續 . . .
~CWin()解構元被呼叫了...
請按任意鍵繼續 . . .
~CShape()解構元被呼叫了...
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 . . .
~CWin()解構元被呼叫了...
請按任意鍵繼續 . . .
~CShape()解構元被呼叫了...
請按任意鍵繼續 . . .
-------------------------*/
```

銷毀 CWin 物件的，此時會先執行~CWin()解構元，再執行基底類別的解構元~CShape()

銷毀 CMiniWin 物件，此時會先執行自己的解構元再執行父類別的解構元，最後再執行基底類別的解構元

自動處理物件的銷毀，此時會先執行自己的解構元再執行父類別的解構元，最後再執行基底類別的解構元

- 如果程式碼裡有使用抽象類別，可把基底類別的解構元設為virtual，如此可以確保解構元會正確地被呼叫以及釋放記憶空間

Casting

- Consider:
  Pet vpet;
  Dog vdog;
  …
  vdog = static_cast<Dog>(vpet); //ILLEGAL!
- Can't cast a pet to be a dog, but:
  vpet = vdog; // Legal!
  vpet = static_cast<Pet>(vdog); //Also legal!
- Upcasting is OK
  - From descendant type to ancestor type

Downcasting

- Downcasting dangerous!
  - Casting from ancestor type to descended type
  - Assumes information is "added"
  - Can be done with dynamic_cast:
    ```
    Pet *ppet;
    ppet = new Dog;
    Dog *pdog = dynamic_cast<Dog*>(ppet);
    ```
    - Legal, but dangerous!
- Downcasting rarely done due to pitfalls
  - Must track all information to be added
  - All member functions must be virtual

Inner Workings of Virtual Functions

- Don’t need to know how to use it!
  - Principle of information hiding
- Virtual function table
  - Compiler creates it
  - Has pointers for each virtual member function
  - Points to location of correct code for that function
- Objects of such classes also have pointer
  - Points to virtual function table

Summary 1

- Late binding delays decision of which member function is called until runtime
  - In C++, virtual functions use late binding
- Pure virtual functions have no definition
  - Classes with at least one are abstract
  - No objects can be created from abstract class
  - Used strictly as base for others to derive

Summary 2

- Derived class objects can be assigned to base class objects
  - Base class members are lost; slicing problem
- Pointer assignments and dynamic objects
  - Allow "fix" to slicing problem
- Make all destructors virtual
  - Good programming practice
  - Ensures memory correctly de-allocated