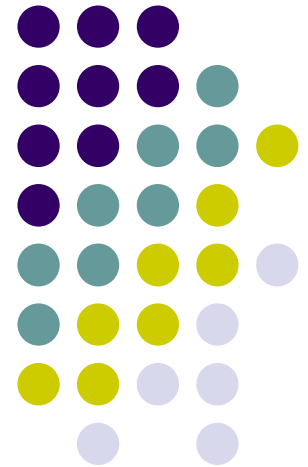


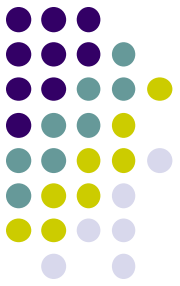
# 解構元與類別的繼承

認識解構元

學習動態記憶體配置與解構元的關係

使用拷貝建構元





# 解構元( Destructor )

- 建構元( constructor )是在物件初次被建立時呼叫
- 解構元( destructor )是在物件被銷毀 ( destroy ) 時呼叫
- 銷毀指的是釋放物件原先所佔有的記憶空間
- 解構元的名稱和類別的名稱相同，之前必須加上一個 ~ ( tilde ) 符號
- 解構元的定義格式

```
~類別名稱()  
{  
    程式敘述 ;  
    ...  
}
```

解構元的名稱必須和類別名稱相同

解構元不能傳入任何引數

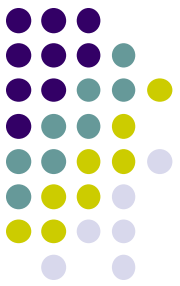
解構元沒有傳回值



# 解構元的使用 (1/2)

- 下面的範例裡加入一個解構元，以便觀察它的運作

```
01 // prog14_1, 解構元的使用
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義視窗類別 CWin
06 {
07     private:
08         char id;
09         int width, height;
10
11     public:
12         CWin(char i,int w,int h):id(i),width(w),height(h)
13         {
14             cout << "建構元被呼叫了..." << endl;
15         }
16         ~CWin() // 解構元
17         {
18             cout << "解構元被呼叫了, win " << this->id << "被銷毀了.." << endl;
19             system("pause");
20         }
```



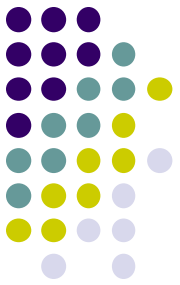
# 解構元的使用 (2/2)

```

21     void show member(void)
22     {
23         cout << "Window " << id << ": ";
24         cout << "width=" << width << ", height=" << height << endl;
25     }
26 };
27
28 int main(void)
29 {
30     CWin win1('A', 50, 40);
31     CWin win2('B', 40, 50);
32     CWin win3('C', 60, 70);
33     CWin win4('D', 90, 40);
34
35     win1.show member();
36     win2.show member();
37
38     system("pause");
39     return 0;
40 }

```

**/\* prog14\_1 OUTPUT-----**  
 建構元被呼叫了...  
 建構元被呼叫了...  
 建構元被呼叫了...  
 建構元被呼叫了...  
 Window A: width=50, height=40  
 Window B: width=40, height=50  
 請按任意鍵繼續 . . . ----- 執行第 38 行的結果  
 解構元被呼叫了, Win D 被銷毀了.. ----- win4 被銷毀, 這是執行第 18 行的結果  
 請按任意鍵繼續 . . . ----- 執行第 19 行的結果  
 解構元被呼叫了, Win C 被銷毀了.. ----- win3 被銷毀, 這是執行第 18 行的結果  
 請按任意鍵繼續 . . . ----- 執行第 19 行的結果  
 解構元被呼叫了, Win B 被銷毀了.. ----- win2 被銷毀, 這是執行第 18 行的結果  
 請按任意鍵繼續 . . . ----- 執行第 19 行的結果  
 解構元被呼叫了, Win A 被銷毀了.. ----- win1 被銷毀, 這是執行第 18 行的結果  
 請按任意鍵繼續 . . . ----- 執行第 19 行的結果  
**-----\*/**



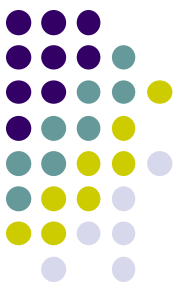
# 解構元的位置

- 在類別內部宣告解構元的原型

```
~CWin();    // 解構元的原型
```

- 在類別外面定義解構元時，要指明其所屬的建構元

```
CWin::~~CWin()  
{  
    // 解構元的程式碼  
}
```



# 固定空間的記憶體配置 (1/2)

- 先看一個簡單的例子，此例無關動態記憶體配置

```
01 // prog14_2, 固定空間的記憶體配置
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義視窗類別 CWin
06 {
07     private:
08     char id,title[20]; // 在編譯時就已經分配固定的記憶體空間
09
10     public:
11     CWin(char i='D', char *text="Default window"):id(i)
12     {
13         strcpy(title,text); // 將 text 指向的字串拷貝到 title 陣列裡
14     }
15     ~CWin() // 解構元
16     {
17         cout << "解構元被呼叫了，Win " << this->id << "被銷毀了.." << endl;
18         system("pause");
19     }
```



## 固定空間的記憶體配置 (2/2)

```

20     void show(void)           // 顯示id與title 成員
21     {
22         cout << "Window " << id << ": " << title << endl;
23     }

```

```

24 };

```

```

25

```

```

26 int main(void)

```

```

27 {

```

```

28     CWin win1('A',"Main window");

```

```

29     CWin win2('B');

```

```

30

```

```

31     win1.show();

```

```

32     win2.show();

```

```

33

```

```

34     cout << "sizeof(win1)= " << sizeof(win1) << endl;

```

```

35     cout << "sizeof(win2)= " << sizeof(win2) << endl;

```

```

36

```

```

37     system("pause");

```

```

38     return 0;

```

```

39 }

```

**/\* prog14\_2 OUTPUT-----**

Window A: Main window

Window B: Default window

sizeof(win1)= 21

sizeof(win2)= 21

請按任意鍵繼續 . . .

解構元被呼叫了, Win B 被銷毀了..

請按任意鍵繼續 . . .

解構元被呼叫了, Win A 被銷毀了..

請按任意鍵繼續 . . .

**-----\*/**

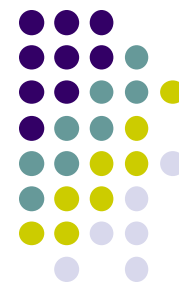


# 使用動態記憶體配置 (1/3)

- 下面的範例將prog14\_2改以動態的方式來配置記憶體

```
01 // prog14_3, 使用動態記憶體配置
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義視窗類別 CWin
06 {
07     private:
08         char id, *title; // 宣告 title 為指向字元陣列的指標
09
10     public:
11         CWin(char i='D', char *text="Default window"):id(i)
12         {
13             title=new char[strlen(text)+1]; // 配置記憶體空間
14             strcpy(title,text);
15         }
16         ~CWin() // 解構元的原型
17         {
18             cout << "解構元被呼叫了，Win " << this->id << "被銷毀了.." << endl;
19             delete [] title; // 釋放 title 所指向的記憶體空間
20             system("pause");
21         }
```





# 使用動態記憶體配置 (2/3)

## ● 原始程式編譯及連結的過程

```

22     void show(void)
23     {
24         cout << "Window " << id << ": " << title << endl;
25     }
26 };
27
28 int main(void)
29 {
30     CWin win1('A', "Main window");
31     CWin win2('B');
32
33     win1.show();
34     win2.show();
35     cout << "sizeof(win1)= " << sizeof(win1) << endl;
36     cout << "sizeof(win2)= " << sizeof(win2) << endl;
37
38     system("pause");
39     return 0;
40 }

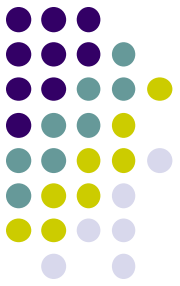
```

**/\* prog14\_3 OUTPUT-----\***

```

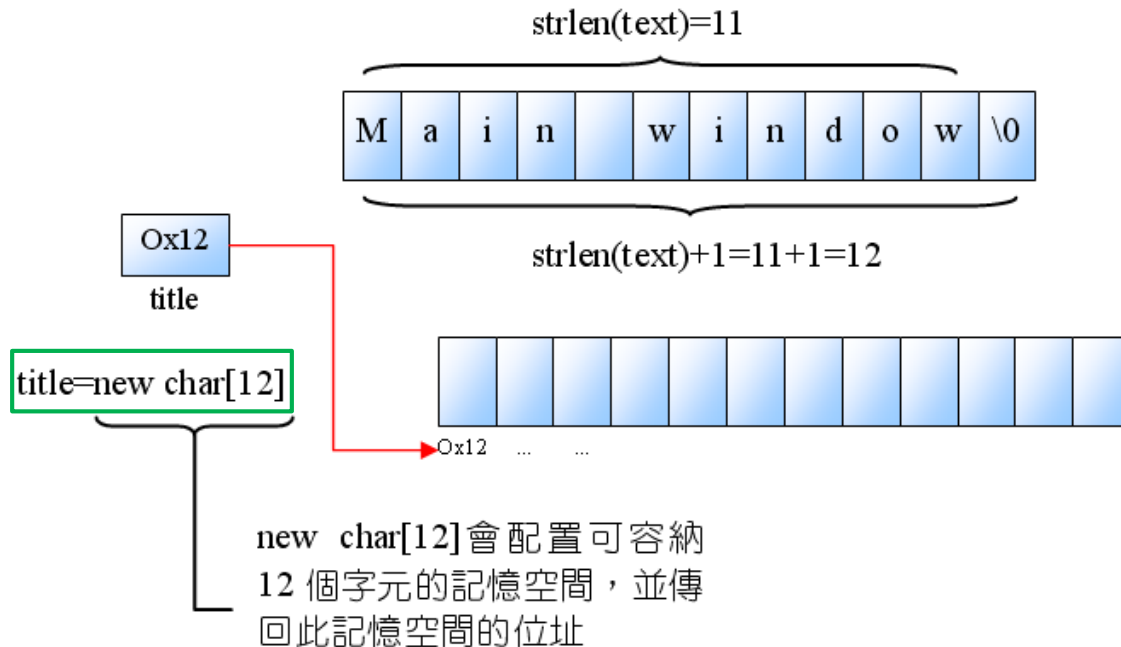
Window A: Main window
Window B: Default window
sizeof(win1)= 8
sizeof(win2)= 8
請按任意鍵繼續 . . .
解構元被呼叫了, Win B 被銷毀了..
請按任意鍵繼續 . . .
解構元被呼叫了, Win A 被銷毀了..
請按任意鍵繼續 . . .
-----*/

```



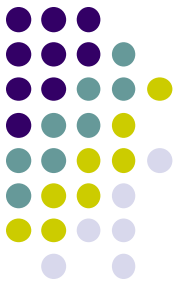
# 使用動態記憶體配置 (3/3)

- 下圖為prog14\_3中，記憶空間配置過程的說明

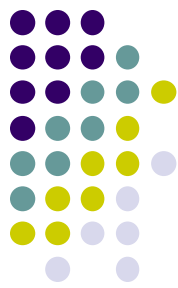


# 補充:

## 動態記憶體與靜態記憶體差別



	靜態配置記憶體	動態配置記憶體
變數宣告	<code>int num</code>	<code>int* numPtr = new int</code>
釋放變數	隨程式結束時自動消失	<code>delete num</code>
優點	程式編譯時就知道(固定) 變數的記憶體大小	可在執行程式時，根據 需求隨意配置變數的記 憶體大小
缺點	無法隨意改變變數佔記 憶體大小	分配與釋放記憶體都需 要占用CPU資源
存放位置	Stack	Heap



# 錯誤的使用動態記憶體配置 (1/2)

- 下面的範例修改自prog14\_3，這是個錯誤的示範

```

01 // prog14_4, 使用動態記憶體配置, 錯誤的示範
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05
06 // 將 prog14_3 CWin 類別的定義放在這裡
07
08 int main(void)
09 {
10     CWin win1('A', "Main window");
11     CWin *ptr;
12     ptr=new CWin('B');
13
14     win1.show();
15     ptr->show();
16
17     system("pause");
18     return 0;
19 }

```

```

/* prog14_4 OUTPUT-----
Window A: Main window
Window B: Defaule window
請按任意鍵繼續 . . .
解構元被呼叫了, Win A 被銷毀了..
請按任意鍵繼續 . . .
-----*/

```

```

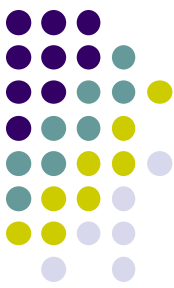
// 宣告 ptr 為指向 CWin 物件的指標
// 建立新的物件, 並讓 ptr 指向它

```

```

// 以 win1 物件呼叫 show() 函數
// 以 ptr 指標呼叫 show() 函數

```



# 錯誤的使用動態記憶體配置 (2/2)

- 下面的範例是更正過後的程式碼

```

01 // prog14_5, 使用動態記憶體配置
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05
06 // 將 prog14_3 CWin 類別的定義放在這裡
07
08 int main(void)
09 {
10     CWin win1('A', "Main window");
11     CWin *ptr;
12     ptr=new CWin('B');
13
14     win1.show();
15     ptr->show();
16
17     system("pause");
18
19     delete ptr;
20
21     return 0;
22 }

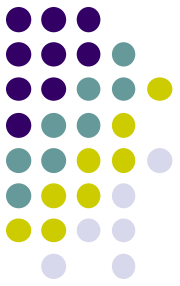
```

```

/* prog14_5 OUTPUT-----
Window A: Main window
Window B: Defaule window
請按任意鍵繼續 . . .
解構元被呼叫了，Win B 被銷毀了..
請按任意鍵繼續 . . .
解構元被呼叫了，Win A 被銷毀了..
請按任意鍵繼續 . . .
-----*/

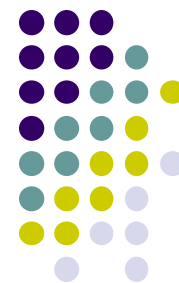
```

// 釋放 ptr 所指向物件之記憶體



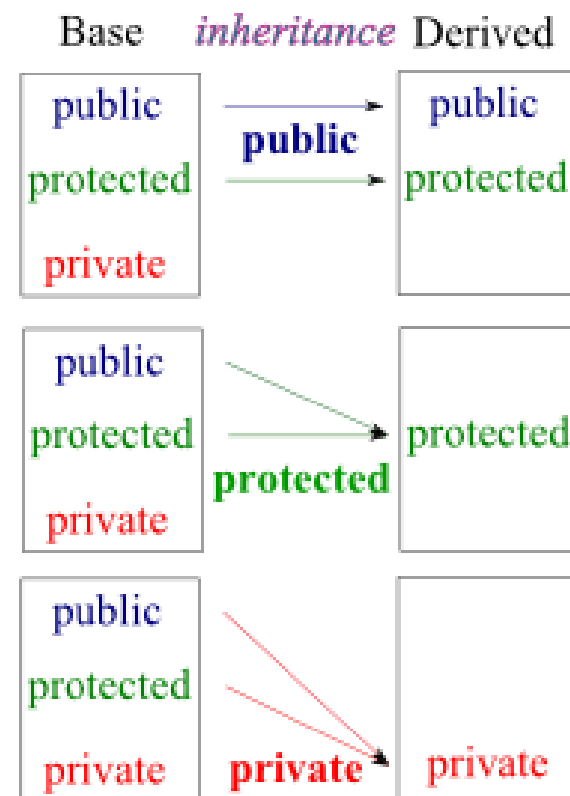
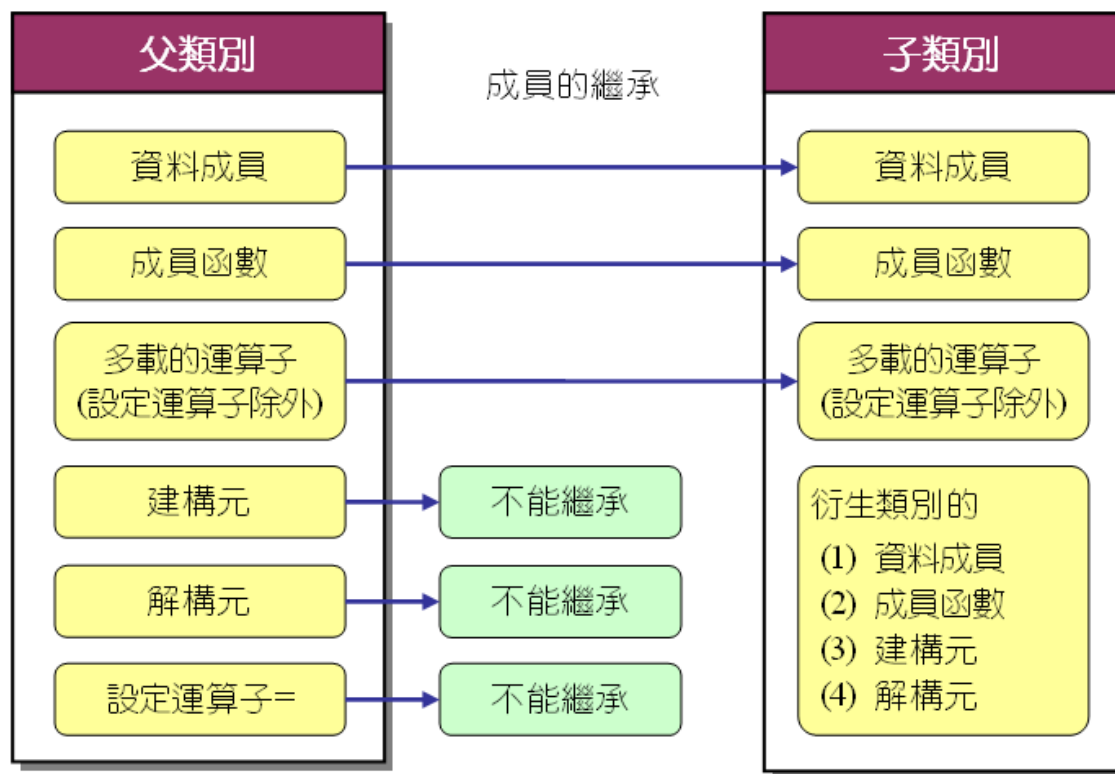
# 基底類別與衍生類別 (1/2)

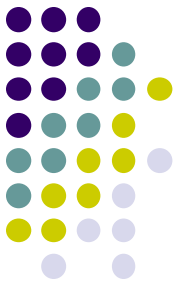
- 以既有類別為基礎，進而衍生出另一類別，稱為「類別的繼承」( inheritance of classes )
- 原有的類別稱為「父類別」( super class ) 或「基底類別」( basis class )
- 因繼承而產生的新類別則稱為「子類別」( sub class ) 或「衍生類別」( derived class )



# 基底類別與衍生類別 (2/2)

## ● 類別成員繼承的關係





# 簡單的繼承範例 (1/6)

- 類別繼承的格式

```
class 父類別名稱    // 定義父類別  
{  
    父類別裡的各種成員  
}
```

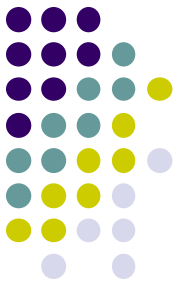
} 父類別

```
class 子類別名稱 : 修飾子 父類別名稱  
{  
    子類別裡的各種成員  
}
```

可為 public, private 或 protected

} 子類別，即由父類別衍生而出的類別

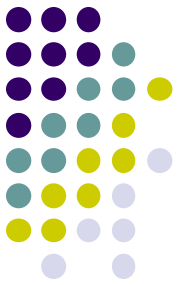




# 簡單的繼承範例 (2/6)

- 下面的範例簡單說明繼承的使用方法

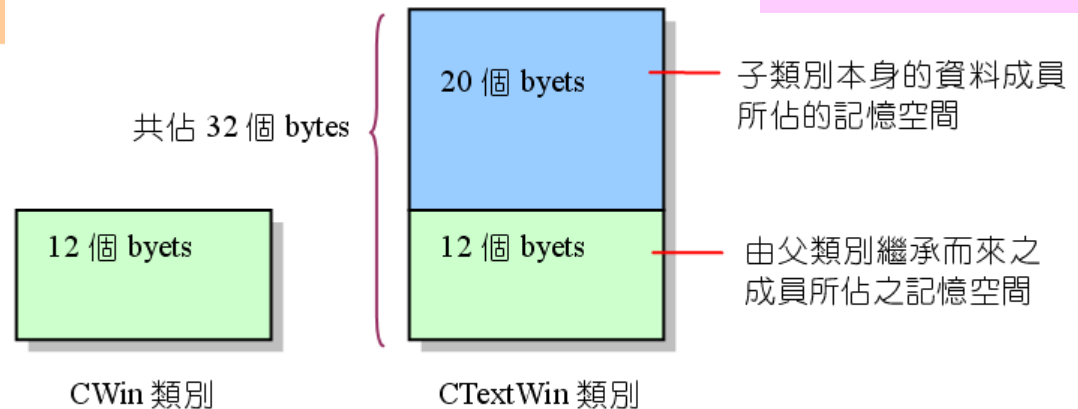
```
01 // prog16_1, 繼承的簡單範例
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義 CWin 類別，在此為父類別
06 {
07     private:
08         char id;
09         int width,height;
10
11     public:
12         CWin(char i='D',int w=10,int h=10):id(i),width(w),height(h)
13         {
14             cout << "CWin()建構元被呼叫了..." << endl;
15         }
16         void show member(void) // 成員函數，用來顯示資料成員的值
17         {
18             cout << "Window " << id << ": ";
19             cout << "width=" << width << ", height=" << height << endl;
20         }
21 };
```

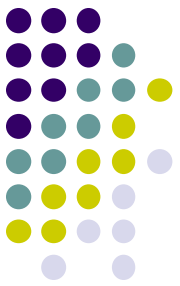


# 簡單的繼承範例 (3/6)

```
22
23 class CTextWin : public CWin // 定義 CTextWin 類別，繼承自 CWin 類別
24 {
25     private:                                // 子類別裡的私有成員
26         char text[20];
27
28     public:                                // 子類別裡的公有成員
29         CTextWin(char *tx)                // 子類別的建構元
30         {
31             cout << "CTextWin() 建構元被呼叫了..." << endl;
32             strcpy(text, tx);
33         }
34         void show text()                  // 子類別的成員函數
35         {
36             cout << "text = " << text << endl;
37         }
38     };
39
```

下圖是本例中，父類別與子類別所佔記憶體之比較





# 簡單的繼承範例 (4/6)

```

40  int main(void)
41  {
42      CWin win('A',50,60);           // 建立父類別的物件
43      CTextWin txt("Hello C++");   // 建立子類別的物件
44
45      win.show member();             // 以父類別物件呼叫父類別的函數
46      txt.show member();             // 以子類別物件呼叫父類別的函數
47      txt.show text();              // 以子類別物件呼叫子類別的函數
48
49      cout << "win 物件佔了 " << sizeof(win) << " bytes" << endl;
50      cout << "txt 物件佔了 " << sizeof(txt) << " bytes" << endl;
51
52      system("pause");
53      return 0;
54  }

```

**/\* prog16\_1 OUTPUT-----**

CWin() 建構元被呼叫了... ——— 42 行建立父類別的物件所得的結果

CWin() 建構元被呼叫了...

CTextWin() 建構元被呼叫了...

} 43 行會先呼叫父類別的建構元，再呼叫子類別的建構元

Window A: width=50, height=60

——— 45 行以 win 物件呼叫 show\_member()

Window D: width=10, height=10

——— 46 行以 txt 物件呼叫 show\_member()

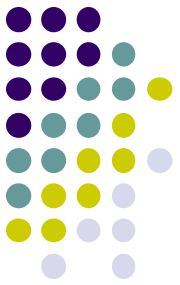
text = Hello C++

——— 47 行以 txt 物件呼叫 show\_text()

win 物件佔了 12 bytes

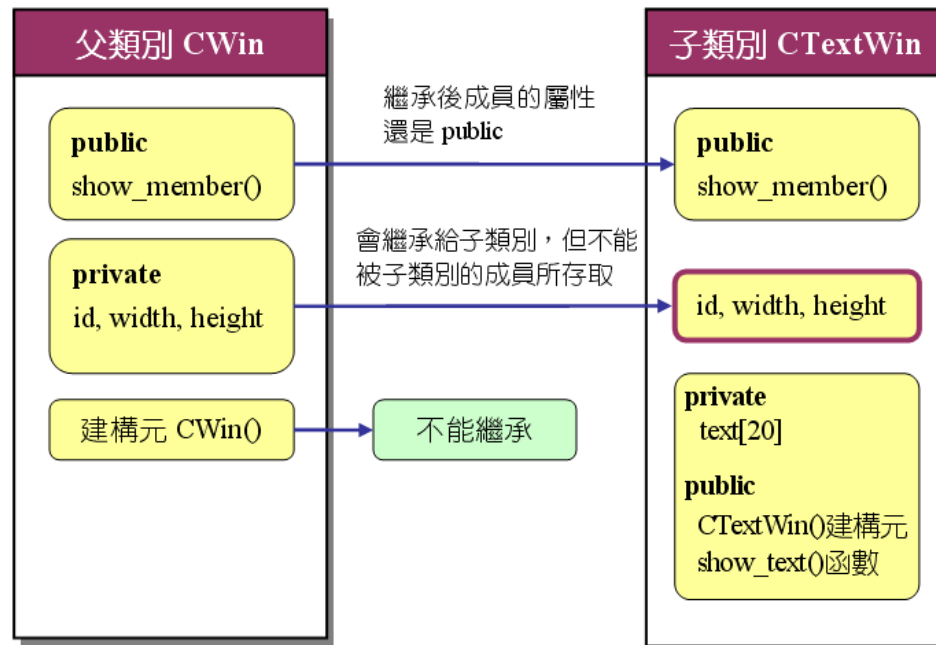
txt 物件佔了 32 bytes

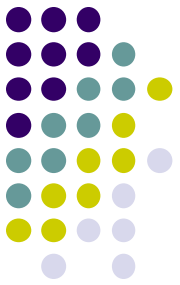
-----\*/



# 簡單的繼承範例 (5/6)

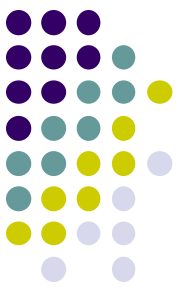
- 本例的繼承關係圖繪製如下





# 簡單的繼承範例 (6/6)

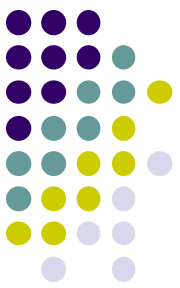
- 由前一個範例可學到下列幾點重要的觀念
  - 透過類別的繼承，可將父類別的成員繼承給子類別
  - 在執行子類別的建構元之前，會先自動呼叫父類別中沒有引數的建構元
  - 子類別物件所佔的位元組，等於自己資料成員所佔的位元組，加上繼承過來之成員所佔用的位元組



# 呼叫父類別中特定的建構元 (1/4)

- 下面是呼叫父類別CWin裡特定建構元的範例

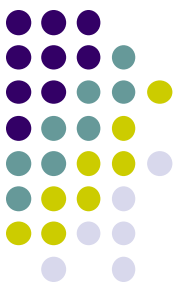
```
01 // prog16_2, 設定運算子多載的進階應用
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義視窗類別 CWin
06 {
07     private:
08         char id;
09         int width,height;
10
11     public:
12         CWin(char i='D',int w=10,int h=10):id(i),width(w),height(h)
13         {
14             cout << "CWin()建構元被呼叫了..." << endl;
15         }
16         CWin(int w,int h):width(w),height(h)
17         {
18             cout << "CWin(int w,int h)建構元被呼叫了..." << endl;
19             id='K';
20         }
```



# 呼叫父類別中特定的建構元 (2/4)

```
21     void show member(void)
22     {
23         cout << "Window " << id << ": ";
24         cout << "width=" << width << ", height=" << height << endl;
25     }
26 };
27
28 class CTextWin : public CWin
29 {
30     private:
31         char text[20];
32
33     public:
34         CTextWin(int w,int h) : CWin(w,h)    // CTextWin(int,int)建構元
35         {
36             cout << "CTextWin(int w,int h)建構元被呼叫了..." << endl;
37             strcpy(text,"Have a good night");
38         }
39         CTextWin(char *tx)    // CTextWin(char *)
40         {
41             cout << "CTextWin(char *tx)建構元被呼叫了..." << endl;
42             strcpy(text,tx);
43         }
```

呼叫父類別裡 16~20 行的  
CWin(w,h) 建構元



# 呼叫父類別中特定的建構元 (3/4)

```

44     void show text()
45     {
46         cout << "text = " << text << endl;
47     }
48 };
49
50 int main(void)
51 {
52     CTextWin tx1("Hello C++"); // 呼叫 39~43 行的 CTextWin() 建構元
53     CTextWin tx2(60,70);       // 呼叫 34~38 行的 CTextWin() 建構元
54
55     tx1.show member();
56     tx1.show text();
57
58     tx2.show member();
59     tx2.show text();
60
61     system("pause");
62     return 0;
63 }

```

**/\* prog16\_2 OUTPUT-----**

CWin() 建構元被呼叫了...

CTextWin(char \*tx) 建構元被呼叫了...

CWin(int w,int h) 建構元被呼叫了...

CTextWin(int w,int h) 建構元被呼叫了...

Window D: width=10, height=10

text = Hello C++

Window K: width=60, height=70

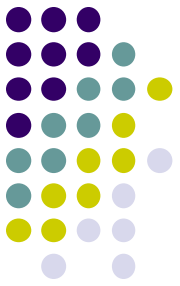
text = Have a good night

**-----\*/**

} 52 行建立 tx1 物件後的執行結果

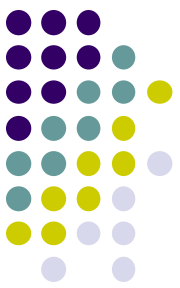
} 53 行建立 tx2 物件後的執行結果





# 呼叫父類別中特定的建構元 (4/4)

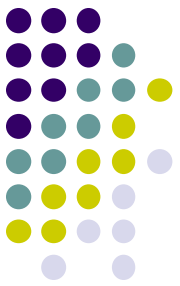
- 這裡有很重要的兩點要提醒讀者：
  - 如果省略34行的敘述，則父類別中沒有引數的建構元還是會被呼叫。
  - 呼叫父類別中特定的建構元，其敘述必須寫在子類別建構元第一行的後面，並以「：」連接，不能置於它處



# 使用建構元常見的錯誤 (1/3)

- 下面是呼叫父類別建構元時常犯的錯誤範例

```
01 // prog16_3, 呼叫父類別建構元時常犯的錯誤
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義視窗類別 CWin
06 {
07     private:
08         char id;
09         int width,height;
10
11     public:
12         CWin(int w,int h):width(w),height(h) // 有兩個引數的建構元
13         {
14             cout << "CWin(int w,int h)建構元被呼叫了..." << endl;
15             id='K';
16         }
17 };
18
```

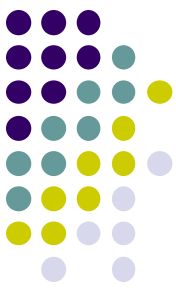


# 使用建構元常見的錯誤 (2/3)

```
19 class CTextWin : public CWin // 定義子類別 CTextWin
20 {
21     private:
22         char text[20];
23
24     public:
25         CTextWin(char *tx) ——— 執行此建構元之前，會先呼叫父類別
26         {                               裡沒有引數的建構元
27             cout << "CTextWin() 建構元被呼叫了..." << endl;
28             strcpy(text, tx);
29         }
30 };
31
32 int main(void)
33 {
34     CTextWin tx1("Hello C++");
35
36     system("pause");
37     return 0;
38 }
```

以 Dev C++ 為例，編譯時會得到如下的錯誤訊息：

no matching function for call to 'CWin::CWin()'



# 使用建構元常見的錯誤 (3/3)

- 下面的程式是修正 prog16\_3 的錯誤

```
01 // prog16_4, prog16_3 錯誤的修正
```

```
02 #include <iostream>
```

```
03 #include <cstdlib>
```

```
04 using namespace std;
```

```
05 class CWin // 定義視窗類別 CWin
```

```
06 {
```

```
07     private:
```

```
08         char id;
```

```
09         int width,height;
```

```
10
```

```
11     public:
```

```
12         CWin(int w,int h):width(w),height(h)
```

```
13         {
```

```
14             cout << "CWin(int w,int h)建構元被呼叫了..." << endl;
```

```
15             id='K';
```

```
16         }
```

```
17         CWin()
```

```
18         {
```

```
19             cout<< "沒有引數的 CWin() 建構元被呼叫了..." << endl;
```

```
20         }
```

```
21     };
```

```
22
```

```
23 // 將 prog16_3, CTextWin 類別的定義放在這兒
```

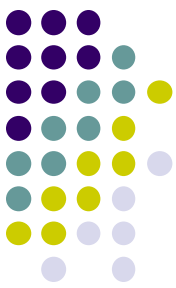
```
24 // 將 prog16_3, main() 主函數放在這兒
```

```
/* prog16_4 OUTPUT-----
```

沒有引數的 CWin() 建構元被呼叫了...

CTextWin() 建構元被呼叫了...

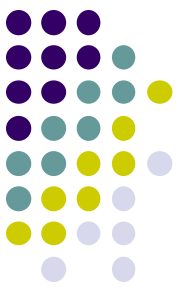
```
-----*/
```



# 父類別裡私有成員的存取 (1/4)

- 錯誤的例子--存取到父類別裡的私有成員

```
01 // prog16_5, 錯誤的例子--存取到父類別裡的私有成員
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義父類別 CWin
06 {
07     private:
08         char id;
09
10     public:
11         CWin(char i):id(i) {}
12 };
13
14 class CTextWin : public CWin // 定義子類別 CTextWin
15 {
16     private:
17         char text[20];
18
```



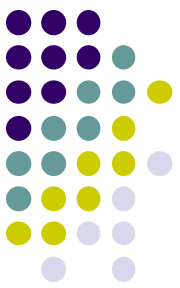
## 父類別裡私有成員的存取 (2/4)

```
19     public:
20         CTextWin(char i, char *tx):CWin(i)
21         {
22             strcpy(text,tx);
23         }
24         void show()
25         {
26             cout << "Window " << id << ": "; // 讀取父類別裡的私有成員
27             cout << "text = " << text << endl;
28         }
29     };
30
31 int main(void)
32 {
33     CTextWin txt('A',"Hello C++");
34
35     txt.show();
36
37     system("pause");
38     return 0;
39 }
```

編譯時將出現下列的錯誤訊息：

```
'id' : cannot access private
member declared in class 'CWin'
```

只要在父類別裡建立公有函數來存取它們  
即可改正錯誤。

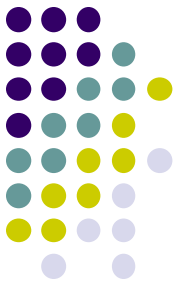


# 父類別裡私有成員的存取 (3/4)

- 修正prog16\_5的錯誤

```
01 // prog16_6, prog16_5 的修正版
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin // 定義父類別 CWin
06 {
07     private:
08         char id;
09
10     public:
11         CWin(char i):id(i) {}
12
13         char get id() // get id() 函數，用來取得 id 成員
14         {
15             return id;
16         }
17 };
18
```

**/\* prog16\_6 OUTPUT-----**  
Window A: text = Hello C++  
**-----\*/**

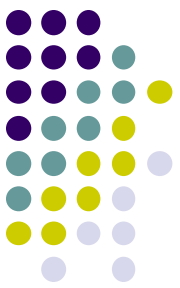


# 父類別裡私有成員的存取 (4/4)

```
19 class CTextWin : public CWin          // 定義子類別 CTextWin
20 {
21     private:
22         char text[20];
23
24     public:
25         CTextWin(char i, char *tx):CWin(i)
26         {
27             strcpy(text, tx);
28         }
29         void show()
30         {
31             cout << "Window " << get id() << ": "; // 呼叫父類別裡的 get id()
32             cout << "text = " << text << endl;
33         }
34 };
35
36 int main(void)
37 {
38     CTextWin txt('A', "Hello C++");
39
40     txt.show();
41
42     system("pause");
43     return 0;
44 }
```

```
/* prog16_6 OUTPUT-----
Window A: text = Hello C++
-----*/
```



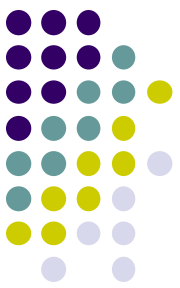


# 使用protected成員 (1/2)

- 把成員宣告成protected最大的好處是兼顧到成員的安全性與便利性
- 下面的範例是prog16\_5的小改版

```
01 // prog16_7, protected 成員的使用
02 #include <iostream>
03 #include <cstdlib>
04 using namespace std;
05 class CWin    // 定義視窗類別 CWin
06 {
07     protected:
08         char id;    // 把 id 宣告成 protected 成員，使得它也可以在子類別裡使用
09
10     public:
11         CWin(char i):id(i) {}
12 };
13
```

**/\* prog16\_7 OUTPUT-----**  
Window A: text = Hello C++  
**-----\*/**



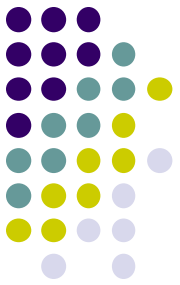
# 使用protected成員 (2/2)

```
14 class CTextWin : public CWin
15 {
16     private:
17         char text[20];
18
19     public:
20         CTextWin(char i, char *tx):CWin(i)
21         {
22             strcpy(text, tx);
23         }
24         void show()
25         {
26             cout << "Window " << id << ": "; // 讀取父類別裡的保護成員
27             cout << "text = " << text << endl;
28         }
29 };
30
31 // 將prog16 5的main()放置在這兒
```

**/\* prog16\_7 OUTPUT-----**

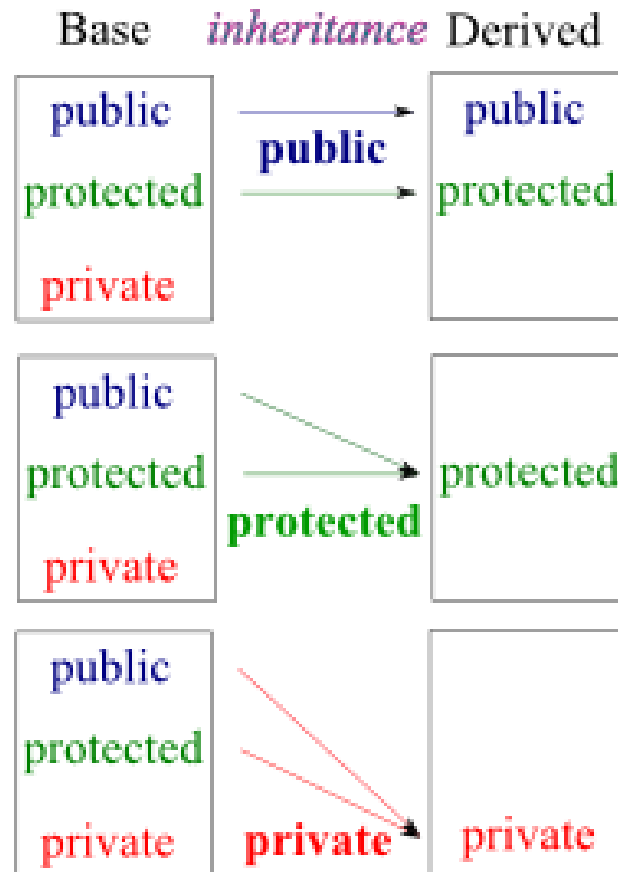
Window A: text = Hello C++

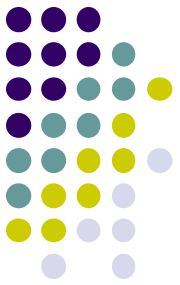
**-----\*/**



# 類別繼承的存取模式

- 下圖說明類別繼承的存取模式





# 多載與改寫

- 「多載」是函數名稱相同，在不同的場合可做不同的事
- 「改寫」是在子類別裡定義與父類別名稱相同的函數，用來覆蓋父類別裡函數功能的一種技術



# 改寫的範例 (1/2)

## ● 簡單的改寫範例

```

01  // prog16_8, 繼承的簡單範例
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
10      public:
11          CWin(char i):id(i){}
12
13          void show member(void)                // 父類別的 show member() 函數
14          {
15              cout << "父類別的 show member() 函數被呼叫了..." << endl;
16              cout << "Window " << id << endl;
17          }
18  };
19

```

```

/* prog16_8 OUTPUT-----
子類別的 show member() 函數被呼叫了...
Window i: text = Hello C++
-----*/

```



## 改寫的範例 (2/2)

```

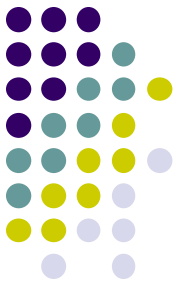
20 class CTextWin : public CWin // 定義 CTextWin 類別，繼承自 CWin 類別
21 {
22     private:                                // 子類別裡的私有成員
23         char text[20];
24
25     public:
26         CTextWin(char i, char *tx):CWin('i') // 子類別的建構元
27         {
28             strcpy(text, tx);
29         }
30         void show member() // 子類別的 show member() 函數
31         {
32             cout << "子類別的 show member() 函數被呼叫了..." << endl;
33             cout << "Window " << id << ": ";
34             cout << "text = " << text << endl;
35         }
36 };
37
38 int main(void)
39 {
40     CTextWin txt('A', "Hello C++"); // 建立子類別的物件
41
42     txt.show member(); // 以子類別物件呼叫 show member() 函數
43
44     system("pause");
45     return 0;
46 }

```

```

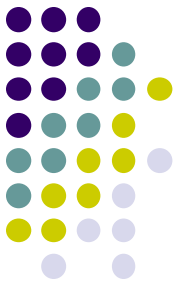
/* prog16_8 OUTPUT-----
子類別的 show member() 函數被呼叫了...
Window i: text = Hello C++
-----*/

```



# 「改寫」與「多載」的比較

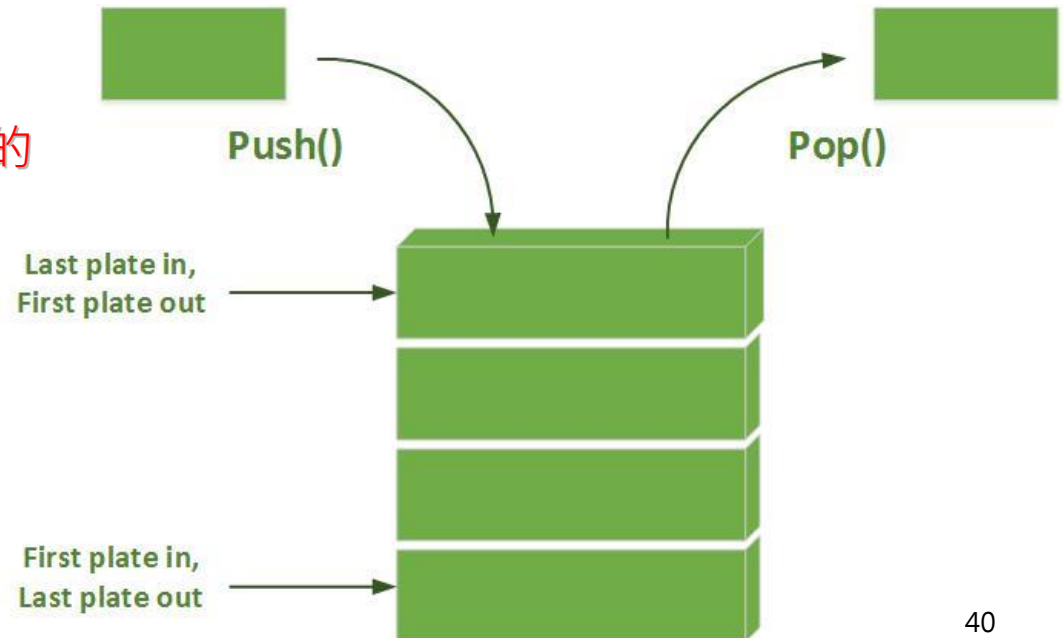
- 「多載」：英文名稱為overloading
  - 它是在相同類別內，定義名稱相同，但引數個數或型態不同的函數，C++可依據引數的個數或型態，呼叫相對應的函數
- 「改寫」：英文名稱為overriding
  - 它是在子類別當中，定義名稱、引數個數與型態均與父類別相同的函數，用以改寫父類別裡函數的功用



## 建構元與解構元的呼叫時機

- 建立物件時，父類別的建構元會先被執行，然後再執行子類別的建構元
- 銷毀物件時，子類別的解構元會先被執行，然後再執行父類別的解構元

因為在建立物件時程式所使用的  
記憶體區間為 **stack**





# 虛擬函數 (1/2)

## 17.1 虛擬函數



- 當函數的執行內容現在無法定義，或現在已定義，將來卻可能會被改變，就可以將函數以虛擬函數的方式撰寫
- 錯誤的範例 - 由早期連結所引起

```
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，-----*/
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

# 虛擬函數 (2/2)

## 17.1 虛擬函數



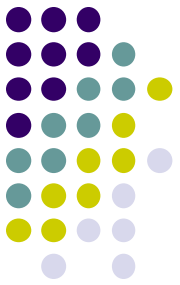
```
17     void show area()                // 父類別的 show area() 函數
18     {
19         cout << "Window " << id << ", area = " << area() << endl;
20     }
21     int area()                      // 父類別的 area() 函數
22     {
23         return width*height;
24     }
25 };
26
27 class CMiniWin : public CWin        // 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

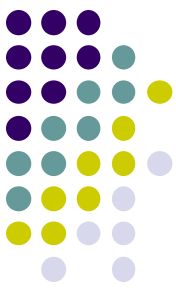
Window B, area = 3000

-----\*/



# 認識虛擬函數

- 編譯時便把父類別裡的show\_area() 和area() 函數連結在一起編譯，這種函數連結的方式稱為**早期連結 (early binding)**
- **虛擬函數可以與呼叫它的函數進行晚期連結 (late binding)**，也就是於程式執行時才由當時的情況來決定是哪一個函數被呼叫，而非在編譯時就把函數配對

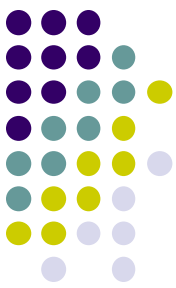


# 修正使用虛擬函數的錯誤 (1/2)

- 下面的程式碼是使用虛擬函數來修正錯誤

```
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)
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         }
```

**/\* prog17\_2 OUTPUT-----**  
Window A, area = 5600  
Window B, area = 2400  
**-----\*/**



## 修正使用虛擬函數的錯誤 (2/2)

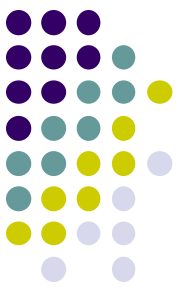
```
21      virtual int area()                // 父類別的 area() 函數
22      {
23          return width*height;
24      }
25  };
26
27  class CMiniWin : public CWin           // 定義子類別 CMiniWin
28  {
29      public:
30          CMiniWin(char i,int w,int h):CWin(i,w,h){} // 子類別的建構元
31
32      virtual int area()                // 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

**-----\*/**

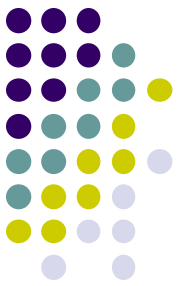


# 指向基底類別物件的指標 (1/2)

- 指向基底類別的指標，也可指向衍生類別所建立的物件(但不能呼叫衍生類別獨有的function)。
- 下面的範例是指向基底類別物件的指標之應用

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



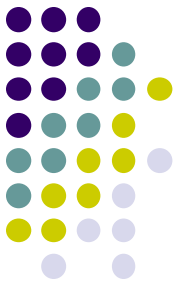
# 指向基底類別物件的指標 (2/2)

```
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

**-----\*/**



# 泛虛擬函數與抽象類別

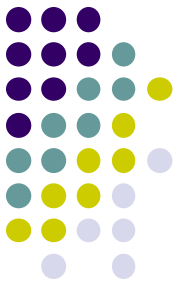
- 「泛虛擬函數」 ( pure virtual function )

在基底類別裡撰寫虛擬函數，使得子類別必須藉由改寫的技術重新定義虛擬函數，具有這個特性的虛擬函數稱為泛虛擬函數

- 「抽象類別」 ( abstract class )

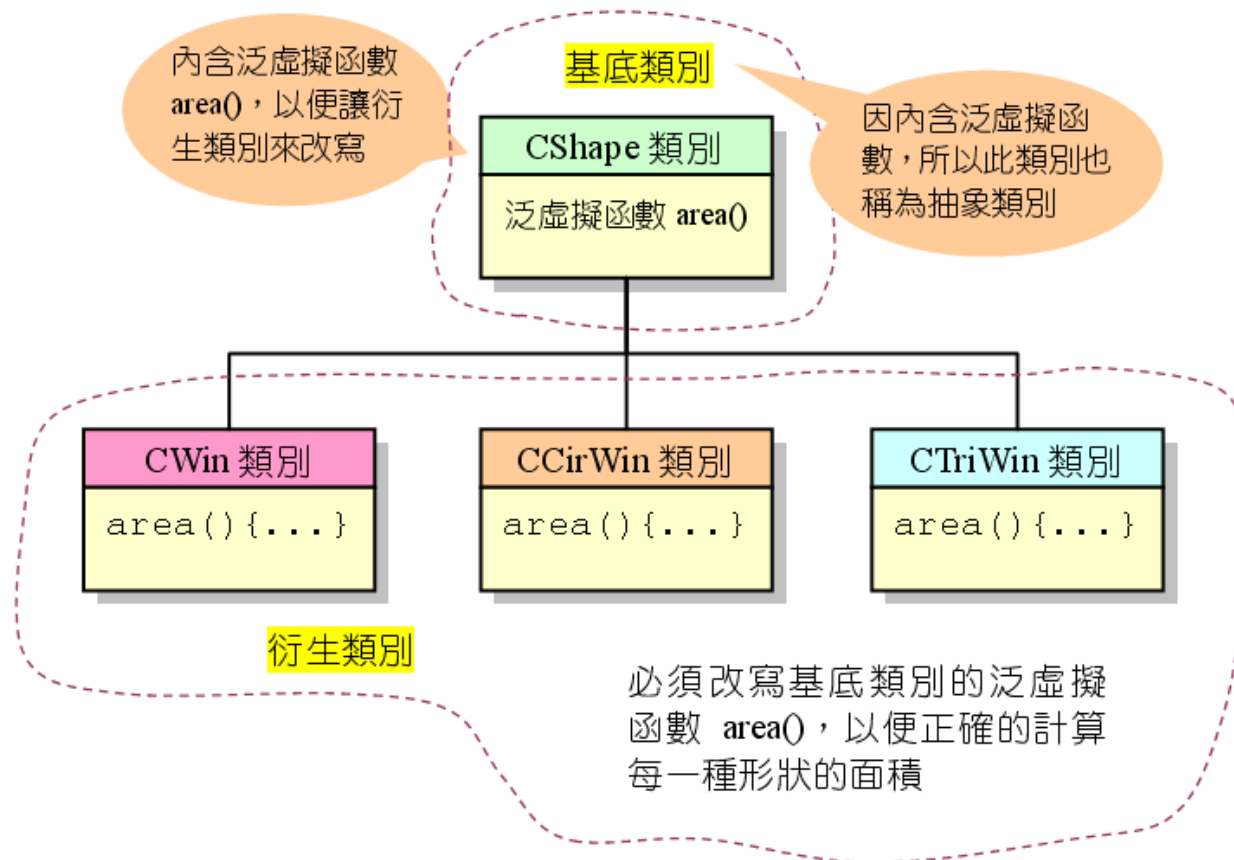
包含有泛虛擬函數的類別稱為抽象類別





# 定義泛虛擬函數 (1/2)

- 下圖是由抽象類別CShape衍生出子類別的示意圖



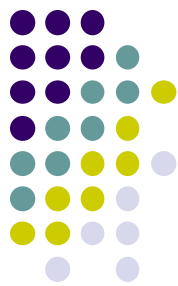


## 定義泛虛擬函數 (2/2)

- 下面為CShape基底抽象類別程式碼

```
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()        // 定義成員函數 show area()
07          {
08              cout << "area = " << area() << endl;
09          }
10  };
```

- 抽象類別有點類似「範本」的作用，其目的是要讓您依據它的格式來修改並建立新的類別

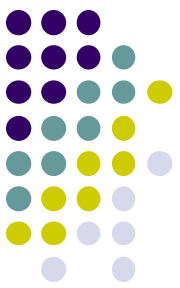


# 抽象類別的實作

- 下面的程式碼是以子類別CWin為例來撰寫的

```
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



# 抽象類別的完整實例 (1/4)

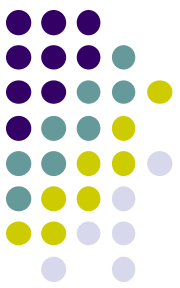
- prog17\_6是抽象類別實作的完整實例

```

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  
**-----\*/**



## 抽象類別的完整實例 (2/4)

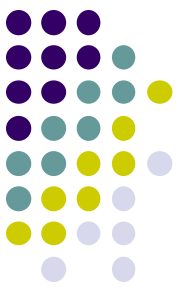
```

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         }

```

/\* prog17\_6 OUTPUT-----  
area = 3000  
CCirWin 物件的面積 = 31400  
-----\*/

在此處明確定義 area() 的處理方式



## 抽象類別的完整實例 (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 };

```

在此處明確定義 area() 的處理方式

改寫父類別的 show\_area() 函數

```

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 CCirWin 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 }

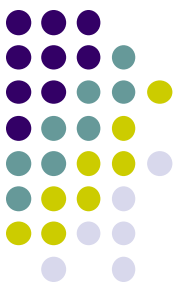
```

**/\* prog17\_6 OUTPUT-----**

```

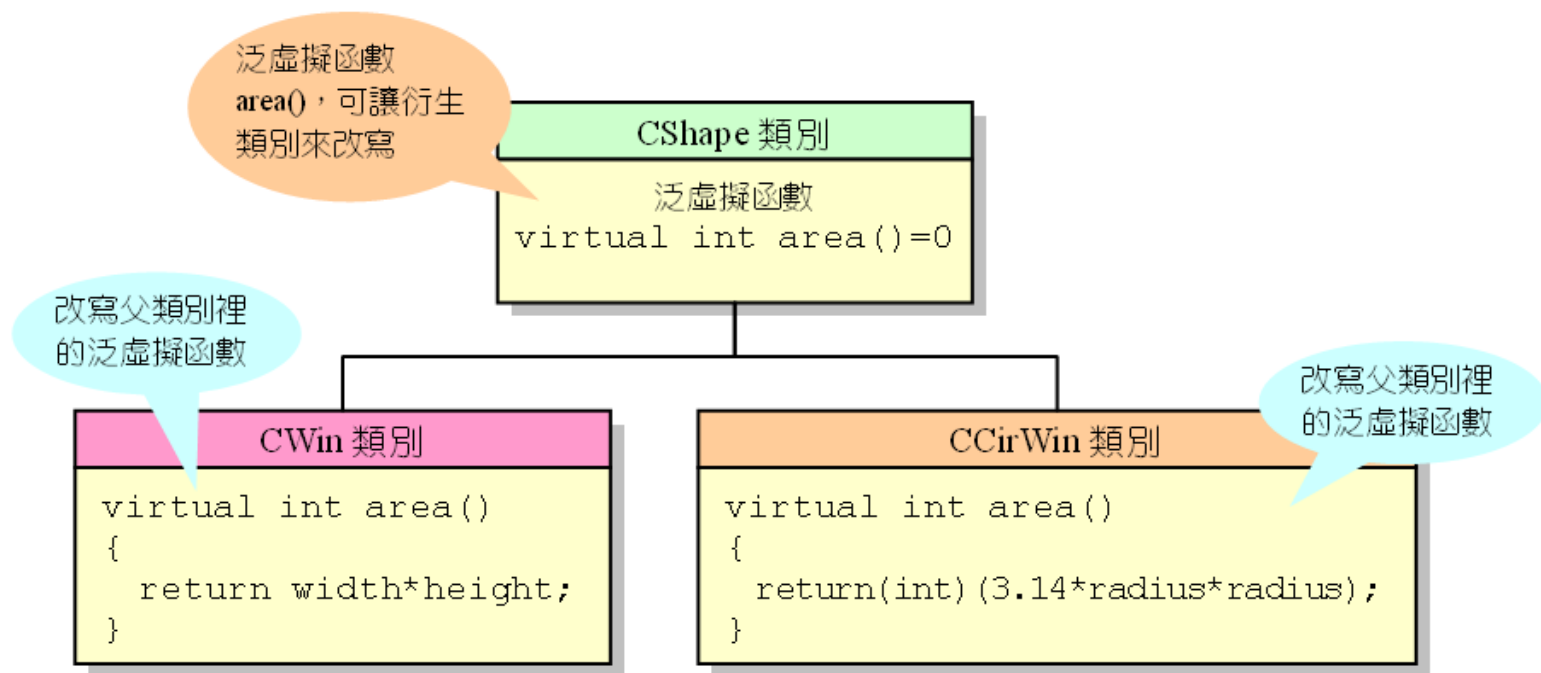
area = 3000
CCirWin 物件的面積 = 31400
-----*/

```



## 抽象類別的完整實例 (4/4)

- 下圖是抽象類別CShape內，泛虛擬函數的實作示意圖





# 使用抽象類別的注意事項

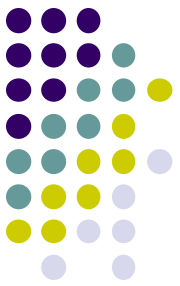
- 抽象類別不能用來直接產生物件

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
    ...
}
```

因為所創建的物件在泛虛擬函數部分並沒有定義！

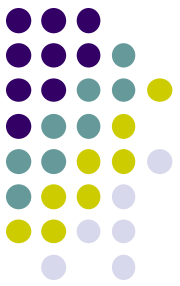




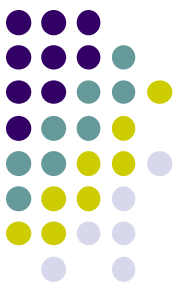
# 練習題

- 根據以下 prototype 完成 implement 的內容。

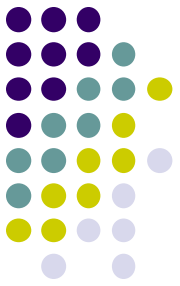
```
1  #include <iostream>
2  #include <math.h>
3
4  using namespace std;
5
6  class shape
7  {
8  public:
9      shape(string Obj, int Num_edges): obj(Obj), num_edges(Num_edges)
10     {
11         cout << "create a " << obj << endl;
12         // implement your code here
13         // dynamic memory allocation for edges of shape
14     }
15
16     void set_edges(double* Edges)
17     {
18         edges = Edges;
19     }
20
21     double* get_edges() const
22     {
23         return edges;
24     }
25 }
```



```
26     int get_num_edges() const
27     {
28         return num_edges;
29     }
30
31     virtual double area() const = 0;
32
33     ~shape()
34     {
35         // implement your code here
36         // delete memory allocation of edges of shape
37     }
38
39     private:
40         string obj;
41         int num_edges;
42         double* edges;
43     };
44
45     class triangle: public shape
46     {
47     public:
48         triangle(double* Edges): shape("triangle", 3)
49         {
50             // implement your code here
51             // set edges of shape
52         }
53
54         double area() const
55         {
56             // implement your code here
57             // get num_edges from shape
58             // get edges from shape
59             // calculate with Heron's formula
60         }
61     };
62
```



```
63 class circle: public shape
64 {
65 public:
66     circle(double Edge): shape("circle", 1)
67     {
68         // implement your code here
69         // set edge of shape
70         // set radius of circle
71     }
72
73     void set_radius(double edge)
74     {
75         // implement your code here
76     }
77
78     double area() const
79     {
80         // implement your code here
81         // get radius of circle
82         // R^2*pi
83     }
84
85 private:
86     double radius;
87 };
88
89 int main()
90 {
91     double tt_edges[] = {30.0, 30.0, 50.0};
92     triangle tt(tt_edges);
93     cout << tt.area() << endl; // the output should be 414.5781
94
95     double cc_edge = 2*M_PI;
96     circle cc(cc_edge);
97     cout << cc.area() << endl; // the output should be 3.141592
98
99     return 0;
100 }
```



The End-