

Example

28

類別與物件： 建構元、解構元

王小明在園遊會擺攤販賣 3 種現沖咖啡。定義一個販售咖啡的類別，用以表示攤位名稱、販售的咖啡名稱、單價與販售數量。設計建構元初始化類別，並以動態記憶體方式配置咖啡名稱、單價與販售數量。輸入攤位名稱、咖啡名稱、單價與數量來設定類別相關的資料成員之後，再完整顯示這些資料。

一、學習目標

建構元（Constructor）是類別內特定的成員函式，可以在物件建立時自動被呼叫；因此，通常用來設定類別的初始狀態或是資料成員設定初始值。解構元（Destructor）也是類別內特定的成員函式，可以在物件被釋放（或稱為銷毀 Destory）時自動被呼叫；因此，通常是用來作為物件在被釋放之前，釋放曾經動態配置過的記憶體。

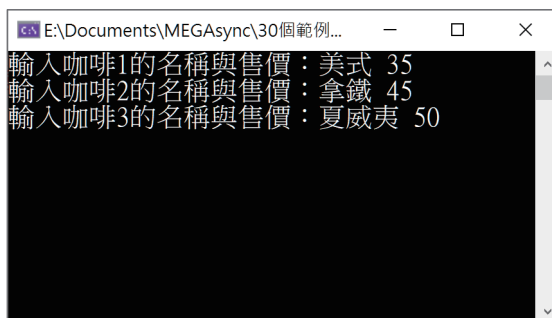
建構元與解構元可以視需求使用或不使用。建構元也可以多載，以符合各種不同情形的物件宣告方式。

二、執行結果

如下圖左為剛開始執行的畫面；下圖右為輸入 3 種咖啡的名稱以及單價。

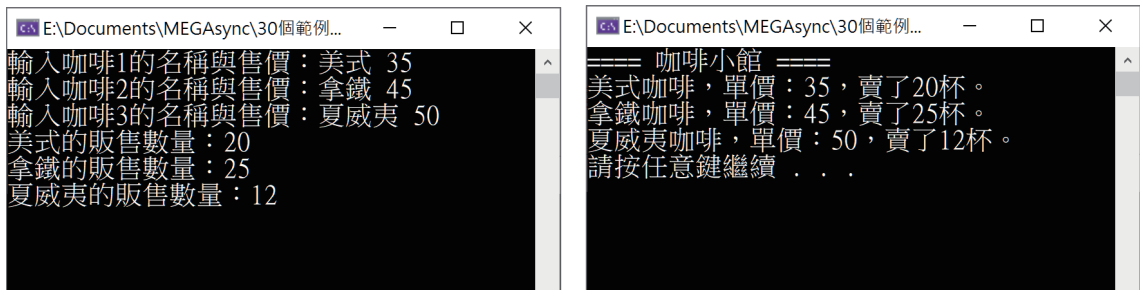


```
E:\Documents\MEGAsync\30個範例...
輸入咖啡1的名稱與售價：
```



```
E:\Documents\MEGAsync\30個範例...
輸入咖啡1的名稱與售價：美式 35
輸入咖啡2的名稱與售價：拿鐵 45
輸入咖啡3的名稱與售價：夏威夷 50
```

下圖左為輸入 3 種咖啡的銷售數量；下圖右為顯示完整的咖啡資料。



28-1 建構元

建構元（或建構子）是類別內特定的成員函式，其函式名稱與類別名稱相同。在物件建立時自動被執行；因此，建構元通常用來作為設定物件剛被建立時的初始狀態，或是資料成員設定初始值。此外，建構元也可以多載，以配合物件不同的建立方式。

建構元的形式

假設記錄 BMI 資料的類別為 `BMIInfo`，則一個最簡單의建構元，如下程式碼第 11-14 行所示。

```

1  class BMIInfo
2  {
3      private:
4          double BMI;    //BMI 值
5
6      public:
7          string name;    // 姓名
8          double height;  // 身高：公尺
9          double weight;  // 體重：公斤
10
11      BMIInfo()
12      {
13          cout << "呼叫建構元 BMIInfo()" << endl;
14      }
15  };

```

← 建構元

建構元沒有函式回傳值型別，並且其名稱與類別名稱相同；此建構元只有 1 列輸出訊息："呼叫建構元 `BMIInfo()`"。此行程式敘述只是為了驗證物件在建立之時會自動呼叫建構元，並沒有其他特別的用意。因此，在物件被建立時，如下程式碼第 19 行便會輸出訊息："呼叫建構元 `BMIInfo()`"。

```

17 int main()
18 {
19     BMIInfo myBMI;
20
21     system("pause");
22 }

```

其實，沒有內容的建構元也稱為預設建構元，是 Visual Studio C++ 自動提供的建構元；當類別裡沒有撰寫建構元時，Visual Studio C++ 便會以自動使用預設建構元。以下為 BMIInfo() 建立 1 個用於初始化資料成員的建構元。如下所示，程式碼第 11-17 行為建構元，在建構元內第 13-16 行設定所有資料成員的初始值。

```

1  class BMIInfo
2  {
3      private:
4          double BMI;
5
6      public:
7          string name;
8          double height;
9          double weight;
10
11      BMIInfo()
12      {
13          BMI = 0;
14          name = "";
15          height = 0;
16          weight = 0;
17      }
18 };

```

} 此建構元設定所有資料
成員的初始值。

建構元如同一般的成員函式，除了在類別內宣告與撰寫程式碼之外，也能在類別中只宣告建構元的函式原型，然後在類別之外撰寫建構元的程式碼；如下所示。程式碼第 11 行只在類別內宣告建構元的函式原型，程式碼第 14-20 行為建構元的程式本體；因為程式本體寫在類別之外，所以必須在建構元的名稱之前使用範圍運算子 "::" 與類別名稱 BMIInfo，表示這是屬於類別 BMIInfo 的成員。

```

1  class BMIInfo
2  {
3      private:
4          double BMI;
5
6      public:
7          string name;
8          double height;
9          double weight;
10
11         BMIInfo(); ← 建構元的原型宣告
12 };
13
14 BMIInfo::BMIInfo()
15 {
16     BMI = 0;
17     name = "";
18     height = 0;
19     weight = 0;
20 }
```

建構元的程式本體，
寫於類別之外。

建構元多載

宣告類別的物件時，可能會有不同的初始設定之狀況；例如：宣告 **BMIInfo** 的物件時，可能尚不知道姓名、身高與體重；或是只知道姓名，但身高與體重尚未測量；或者姓名、身高與體重都已經知道。爲了因應這些不同的初始條件，便可以使用建構元多載的方式來處理。只要建構元所接收的參數個數或是資料型別不同，便可視爲不同的建構元。

例如：以下範例替計算 BMI 的類別 **BMIInfo** 設計 3 個建構元，如下程式碼所示；分別爲：第 11-17 行、第 19-25 行與第 27-33 行此 3 個建構元。第 11-17 行的第 1 個建構元沒有帶任何的參數，第 13-16 行設定類別裡的資料成員的初始值。第 19-25 行的第 2 個建構元接收 1 個 **string** 型別的參數 **name**，用於設定資料成員 **name**，其餘的資料成員則設定爲預設的初始值。第 27-33 行的第 3 個建構元，接收 3 個參數，分別是姓名 **name**、身高 **height** 與體重 **weight**，用於設定類別內的資料成員的初始值。

```

1  class BMIInfo
2  {
3      private:
4          double BMI;
5
```

```

6      public:
7          string name;
8          double height;
9          double weight;
10
11      BMIInfo()
12      {
13          BMI = 0;
14          name = "";
15          height = 0;
16          weight = 0;
17      }
18
19      BMIInfo(string name)
20      {
21          BMI = 0;
22          this->name = name;
23          height = 0;
24          weight = 0;
25      }
26
27      BMIInfo(string name, double height, double weight)
28      {
29          BMI = 0;
30          this->name = name;
31          this->height = height;
32          this->weight = weight;
33      }
34  };

```

第 1 個建構元

第 2 個建構元

第 3 個建構元

因為類別 `BMIInfo` 有了 3 個建構元之後，宣告物件變得更有彈性與方便；如下所示：第 38-40 行分別宣告 `BMIInfo` 類別的物件 `bmi1-bmi3`。

```

36 int main()
37 {
38     BMIInfo bmi1; ← 執行第 1 個建構元
39     BMIInfo bmi2(" 王小明 "); ← 執行第 2 個建構元
40     BMIInfo bmi3(" 真美麗 ", 1.62, 50); ← 執行第 3 個建構元
41 }

```

第 38 行所宣告的物件 `bmi1()` 並沒有傳遞任何引數，因此當物件建立時會自動呼叫第 1 個建構元。第 39 行所宣告的物件 `bmi2()` 只傳遞 1 個字串引數，因此當物件建立時會自動呼叫

第 2 個建構元。第 40 行所宣告的物件 `bmi3()` 傳遞 3 個引數，其引數的型別為：字串、浮點數、浮點數，因此當物件建立時會自動呼叫第 3 個建構元。

特別注意，當類別提供了帶有參數的建構元之後，並且沒有提供不帶參數的建構元，則使用類別宣告不帶參數的物件時，就必須加上 1 組小括弧 `()`。例如：上述的 `BMIInfo` 類別若刪除了第 1 個建構元，則剩下 2 個都帶有參數的建構元。那麼程式第 38 行宣告 `BMIInfo` 物件 `bmi1` 會發生錯誤，應該修正為：

```
38 BMIInfo bmi1();
```

如此修正表示自動使用 Visual Studio C++ 預設的沒有帶任何參數的預設建構元。

容易混淆的建構元多載

建構元的參數也可以有預設值，就如同一般自訂函式相同的方式。例如，再替 `BMIInfo` 類別增加第 4 個建構元，如下所示。然而，執行時會發生錯誤：因為第 3 個建構元與第 4 個建構元是相同的形式，視同宣告了 2 個相同的成員函式，所以發生了錯誤。

```
1 class BMIInfo
2 {
3     :
35     BMIInfo(string name, double height=1.5, double weight=45)
36     {
37         BMI = 0;
38         this->name = name;
39         this->height = height;
40         this->weight = weight;
41     }
42 };
```

第 3 個建構元一定要接收 3 個參數，而第 4 個建構元一定要接收第 1 個參數，其餘 2 個參數可以接收也可以不接收。因此，第 3 個建構元可以視為是第 4 個建構元的其中一種情形；所以第 3 個建構元變成了多餘的建構元，也就和第 4 個建構元重複了（其實，第 2 個建構元只需要接收一個字串參數，也是第 4 種建構元的其中一種情形。）。

函式多載可以讓程式變得更有彈性，但也容易在設計上造成如上述的錯誤。因此，將類別 `BMIInfo` 的多個建構元重新調整為：刪除第 2、3 個建構元，保留第 1、4 個建構元；如此一來類別 `BMIInfo` 可以更靈活地利用不同的建構元宣告物件。

經過調整之後的類別 BMIInfo 如下所示。

```

1  class BMIInfo
2  {
3      private:
4          double BMI;
5
6      public:
7          string name;
8          double height;
9          double weight;
10
11         BMIInfo()
12         {
13             BMI = 0;
14             name = "";
15             height = 0;
16             weight = 0;
17         }
18
19         BMIInfo(string name, double height = 1.5, double weight = 45)
20         {
21             BMI = 0;
22             this->name = name;
23             this->height = height;
24             this->weight = weight;
25         }
26 };

```

資料成員的簡易設定

建構元中對於資料成員的初始值設定，還有另一種簡單的表示方法；以上述程式碼第 19-25 行的建構元 BMIInfo() 為例，改寫後如下所示：

```

BMIInfo(string n,double h = 1.5,double w = 45):name(n),height(h),weight(w)
{
    其他程式敘述；
}

```

↑
資料成員初始值設定

其中 name(n) 表示使用參數 n 來做為資料成員 name 的初始值，height(h) 表示使用參數 h 作為資料成員 height 的初始值，weight(w) 表示使用參數 w 作為資料成員 weight 的初始值；如此的方式可以簡化資料成員的初始值設定。

練習 1：誰的購買金額比較多

王小明與真美麗 2 人在買飲料，飲料每瓶 25 元。真美麗買了 7 瓶飲料，王小明購買飲料的數量則需要另外輸入。購買飲料的資料使用類別表示，資料成員有：飲料單價、姓名、購買金額、購買數量。類別提供建構元，並提供以下功能之成員函式：計算購買金額、設定購買數量、比較誰的購買金額比較多、取得姓名與購買金額。

■ 解說

假設類別名稱爲 `Consumption`，因為真美麗已知道購買了 7 瓶飲料，而王小明則需要輸入飲料的購買數量，由此可知需要提供 2 種建構元：`Consumption(姓名, 購買數量)` 與 `Consumption(姓名)`。真美麗使用第 1 種建構元：`Consumption(" 真美麗 ", 7)`，王小明則使用第 2 種建構元：`Consumption(" 王小明 ")`，然後再使用設定購買數量的成員函式設定王小明購買飲料的數量。因此，類別 `Consumption`、資料成員與建構元的形式如下所示。

```

1  class Consumption
2  {
3      private:
4          const int price = 25; // 飲料單價
5          string name; // 姓名
6          int total; // 購買總金額
7          int num; // 購買數量
8
9      public:
10         Consumption(string name, int num) // 第 1 個建構元
11         {
12             this->name = name;
13             this->num = num;
14             total = 0;
15             compute(); // 呼叫 compute() 成員函式計算總金額
16         }
17
18         Consumption(string name) // 第 2 個建構元
19         {
20             this->name = name;
21             num=0;
22             total = 0;
23         }
24
25         void compute() // 計算總金額
26         {

```



```

27         total = num * price;
28     }
29     :
30     其他的成員函式
31 };

```

Consumption 類別將所有的資料成員都放置於 **private** 區段內，如程式碼第 4-7 行所示；**public** 區段則放置建構元與其他的成員函式。

第 10-16 行、第 18-23 行分別為 2 個建構元；第 1 個建構元接受 2 個參數：姓名與購買數量，第 2 個建構元只接受 1 個參數：姓名。第 1 個建構元裡除了設定資料成員的初始值之外，第 15 行還呼叫了成員函式 **compute()** 計算購買的總金額。

至於「誰的購買金額比較多」這個成員函式，可以有不同的設計方法；請參考分析與討論。在此設計為：接受一個傳址呼叫的物件參數，並比較自己（**this**）的購買金額和物件參數的購買金額誰比較高，並回傳購買金額比較高的那個物件；如下所示：

```

1  class Consumption
2  {
3      private:
4          :
5          資料成員
6
7      public:
8          Consumption* comsMore(Consumption *com) // 比較誰的購買金額多
9          {
10             if (this->total >= com->total)
11                 return this;
12             else
13                 return com;
14         }
15
16         :
17         其他的成員函式
18 };

```

程式碼第 8 行成員函式 **comsMore()** 用於判斷誰的購買金額比較多，接收 1 個 **Consumption** 型別的指標物件參數 **com**；由此可知，物件參數 **com** 使用傳址呼叫的方式。第 10 行判斷若自己的購買金額 **this->total** 若大於等於物件參數的購買金額 **com->total**，則回傳自己 **this**（不用加上 **&** 運算子，因為 **this** 本身已是指標），否則回傳物件參數 **com**。

執行結果

輸入王小明要買幾瓶飲料：5
真美麗購買比較多，共 175 元

程式碼列表

```
1 #include <iostream>
2 using namespace std;
3
4 class Consumption
5 {
6     private:
7         const int price = 25; // 飲料單價
8         string name; // 姓名
9         int total; // 購買總金額
10        int num; // 購買數量
11
12    public:
13        Consumption(string name, int num) // 建構元 1
14        {
15            this->name = name;
16            this->num = num;
17            total = 0;
18            compute(); // 呼叫 compute() 成員函式計算總金額
19        }
20
21        Consumption(string name) // 建構元 2
22        {
23            this->name = name;
24            num=0;
25            total = 0;
26        }
27
28        void compute() // 計算總金額
29        {
30            total = num * price;
31        }
32
33        void setNumber(int num) // 設定數量
34        {
35            this->num = num;
```

```

36     }
37
38     Consumption* comsMore(Consumption *com) // 比較誰的金額多
39     {
40         if (this->total >= com->total)
41             return this;
42         else
43             return com;
44     }
45
46     string getName(int& total) //getName() 多載：回傳姓名，並取得總金額
47     {
48         total = this->total;
49         return name;
50     }
51
52     string getName() //getName() 多載：回傳姓名
53     {
54         return name;
55     }
56 };
57
58 int main()
59 {
60     Consumption com1(" 王小明 "), com2(" 真美麗 ", 7), *ptrCom;
61     int number;
62     int total=0;
63
64     cout << " 輸入 " << com1.getName() << " 要買幾瓶飲料：";
65     cin >> number;
66     com1.setNumber(number);
67     com1.compute();
68
69     ptrCom = com1.comsMore(&com2);
70     cout << ptrCom->getName(total) << " 購買比較多，共 " << total << " 元 " << endl;
71
72     system("pause");
73 }

```

程式講解

1. 程式碼第 1-2 行引入 `iostream` 標頭檔與宣告使用 `std` 命名空間。
2. 程式碼第 4-56 行定義類別 `Consumption`，第 6-10 行的 `private` 區段內有 4 個資料成員，分別為：使用 `const` 修飾字修飾的整數變數 `price`，表示物品的價錢、

`string` 型別的變數 `name` 表示姓名、整數型別的 `total` 與 `num`，各自表示購買總金額與購買數量。

在 `public` 區段內則放置建構元與成員函式。第 13-19 行、第 21-26 行為建構元；第 1 個建構元接收姓名 `name` 與數量 `num` 此 2 個參數，設定資料成員的初始值，並呼叫成員函式 `compute()` 計算購買總金額。第 2 個建構元只接收 1 個姓名 `name` 參數，並設定資料成員的初始值。

3. 第 28-31 行為成員函式 `compute()`，用於計算購買金額。第 30 行將飲料的單價 `price` 乘上數量 `num`，並將計算出來的金額儲存於資料成員 `total`。第 33-36 行為成員函式 `setNumber()`，用於設定飲料的購買數量，接收一個整數型別的參數 `num`，並將資料成員 `this->num` 設定為參數 `num`。第 38-44 行為成員函式 `comsMore()`，用於比較誰的購買金額比較多，並回傳購買金額比較多的那個物件。
4. 第 46-50 行、第 52-55 行為多載的成員函式 `getName()`，用於回傳類別的資料成員 `name`。第 1 個 `getName()` 函式接收 1 個參考型別的參數 `total`，由此可知此參數用於回傳資料。第 48 行將資料成員 `this->total` 設定給參數 `total`，所以原來的引數的值便等於 `this->Total`。第 49 行回傳資料成員 `name`。第 52-55 行為第 2 個 `getName()` 函式，第 54 行回傳資料成員 `name`。
5. 開始於 `main()` 主函式中撰寫程式。第 60 行宣告 `Consumption` 類別的物件 `com1`、`com2` 與指標 `ptrCom`；物件 `com1` 與 `com2` 分別代表王小明與真美麗購買飲料的資料。物件 `com1` 傳遞 1 個字串參數 (" 王小明 ")，所以會自動執行第 21-26 行的類別建構元。物件 `com2` 傳遞 2 個參數 (" 真美麗 ", 7)，所以會自動執行第 13-19 行的類別建構元。

指標物件 `ptrCom` 則用於接收成員函式 `comsMore()` 所回傳物件。第 61-62 行的整數變數 `number` 與 `total`，則表示購買飲料的數量與購買的金額。

6. 因為物件 `com1` 在建立之時只有設定姓名 " 王小明 "，但沒有設定購買飲料的數量；因此，第 64-65 行需要設定購買飲料的數量。第 64 行呼叫 `com1` 的成員函式 `getName()` 取得姓名 " 王小明 "（因為只有傳遞 1 個字串型別的參數，所以會執行第 52-55 行的 `getName()` 成員函式。），然後顯示購買飲料數量的提示。第 65 行讀取輸入的飲料數量，並儲存於變數 `number`。

第 66 行先使用物件 `com1` 的成員函式 `setNumber()` 設定購買的飲料數量 `number`，第 67 行再呼叫物件 `com1` 的成員函式 `compute()` 計算購買總金額。

7. 程式碼第 69 行由物件 `com1` 執行成員函式 `comsMore()` 比較 `com1` 與 `com2` 誰的購買金額比較多，所以將物件 `com2` 作為引數傳遞給 `comsMore()` 函式。因為 `comsMore` 函式接收的是傳址呼叫的參數，所以在引數 `com2` 之前加上求址運算子 "&"；並將

`comsMore()` 函式所回傳的物件儲存於指標物件 `ptrCom`；因此，`ptrCom` 便指向購買金額比較多的那個物件（`com1` 或是 `com2`）。

8. 程式碼第 70 行先呼叫指標物件 `ptrCom` 的成員函式 `getName()` 取得姓名，並傳入 1 個參考呼叫的引數 `total`，接著顯示姓名與購買金額。

28-2 解構元

解構元（或解構子）也是類別裡特定的成員函式，在物件被釋放時會自動被呼叫。例如：在一個自訂函式內宣告了類別的物件，當結束自訂函式要返回呼叫者時，此物件也會被釋放，此時物件的解構元就會自動執行。需要特別注意，如果是使用 `new` 所配置的物件，則要執行 `delete` 命令時，解構元才會被執行。

因此，解構元通常用於當物件要被釋放之前，將某些變數的狀態重新設定，或是釋放經由 `new` 所配置的記憶體空間。解構元的語法如下所示：

```
~類別名稱 ()
{
    程式敘述;
}
```

解構元由字元 `'~'` 開始，並接著類別名稱與一組小括弧，並在左右大括弧內撰寫程式敘述。解構元沒有多載，也不可以有函式回傳值型別、參數。解構元的程式本體也可以寫在類別之外，就如同成員函式相同的方式。例如，撰寫類別 `myClass` 的解構元：如程式碼第 8-11 行，這是一個什麼事情都沒做的解構元；正因為什麼事情都沒做，所以這個結構元是多餘的。

```

1  class myClass
2  {
3      private:
4          :
5      public:
6          :
7
8      ~myClass()
9      {
10
11      }
12 };

```

~myClass()
{

}

← 解構元

再看以下的例子：程式碼第 4-18 行定義類別 `myClass`，只有一個字串型別的資料成員 `ID`，用於記錄物件的名稱。第 9-12 行為建構元，用於設定資料成員 `ID`。第 14-17 行為解構元，其中只有一行程敘述：顯示物件名稱與字串 " 執行解構元 "，用於作為解構元被執行的證明。

```

1  #include <iostream>
2  using namespace std;
3
4  class myClass
5  {
6      public:
7          string ID;
8
9          myClass(string ID)
10         {
11             this->ID = ID;
12         }
13
14         ~myClass()
15         {
16             cout << ID << " 執行解構元 " << endl;
17         }
18 };
19
20 void myfunc()
21 {
22     myClass clsB("clsB ");
23 }
24
25 int main()
26 {
27     myClass clsA("clsA ");
28     myClass* ptrCls = new myClass("ptrCls ");
29
30     delete ptrCls;
31
32     myfunc();
33
34     system("pause");
35 }

```

程式碼第 20-23 行定義自訂函式 `myFunc()`，裡面只有一行程式敘述：宣告 `myClass` 類別的物件 `clsB`，並傳遞字串 "clsB"，表示這個物件的名稱為 `clsB`。

第 27 行宣告 `myClass` 類別的物件 `clsA`，並傳遞字串 `"clsA"`，表示這個物件的名稱為 `clsA`。第 28 行宣告 `myClass` 類別的指標物件 `ptrCls`，並使用 `new` 配置記憶體，也傳遞字串 `"ptrCls"` 表示這個物件的名稱為 `ptrCls`。

第 30 行使用 `delete` 指令釋放指標物件 `ptrCls`，因此會自動執行解構元，所以會顯示：`"ptrCls 執行解構元"`。第 32 行執行自訂函式 `myFunc()`，當執行完畢返回時會釋放在自訂函式 `myFunc()` 中宣告的物件 `clsB`，此時也會執行結構元，所以會顯示 `"clsB 執行解構元"`。接著執行第 34 行的暫停命令，所以顯示 `"請按任意鍵繼續..."`。最後在主函式 `main()` 結束時會釋放物件 `clsA`，因此自動執行結構元，所以會顯示 `"clsA 執行解構元"`。所有顯示的訊息如下所示：

```
ptrCls 執行解構元
clsB 執行解構元
請按任意鍵繼續...
clsA 執行解構元
```

練習 2：解構元釋放動態配置的記憶體

一年級有 A、B 兩個班級，每位學生都有 1 個成績。輸入 A、B 兩班的人數之後，使用亂數模擬每位學生的成績：40-100 分之間的分數。

■ 解說

假設記錄學生成績的類別為 `stutClass`，資料成員應有：班級名稱、學生人數與學生成績。可以確認的資料成員只有班級名稱，因為 2 班的學生人數要等到輸入人數之後才能確定；所以也無法事先宣告固定長度的陣列來儲存學生的成績。因此，必須將儲存學生成績的陣列宣告為指標變數，等待學生人數確定之後，再使用動態記憶配置的方式配置記憶體空間。因此，類別 `stutClass` 的資料成員應如下所示：

```
1 class stutClass
2 {
3     private:
4         int stutNum; // 學生人數
5         int* score;  // 學生成績
6
7     public:
8         string clsID; // 班級名稱
9         成員函式;
10        :
11 };
```

至於成員函式的部分，可提供以下功能之成員函式：建構元 `stutClass()`、設定學生人數 `setNumber()`、設定所有學生成績 `setScore()`、取得學生人數 `getNumber()`、取得學生成績 `getScore()`，以及解構元 `~stutClass()`。

建構元 `stutClass()`：可以接收班級名稱作為參數，並設定其餘資料成員的初始值；因此，其函式架構為：

```

1  stutClass(string id)
2  {
3      資料成員初始化；
4  }
```

設定學生人數的成員函式 `setNumber()`：可以接收學生人數作為參數；除了設定學生人數 `stutNum` 之外，也能配置記憶體空間給儲存學生成績的指標變數 `score`；因此，其函式架構如下所示。若記憶體配置成功則回傳 `true`，否則回傳 `false`。

```

1  bool setNumber(int num)
2  {
3      設定學生人數；
4      配置記憶體給資料成員 score；
5      return true 或 false；
6  }
```

設定所有學生成績的成員函式 `setScore()`：使用亂數的方式隨機設定學生的成績，因此，其函式架構如下所示。

```

1  void setScore()
2  {
3      for (int i = 0; i < stutNum; i++)
4          score[i] = 介於 40-100 之間的亂數；
5  }
```

取得學生人數的成員函式 `getNumber()` 則只需要回傳資料成員 `stutNum`。取得學生成績的成員函式 `getScore()` 需要回傳學生成績的指標變數 `score`，因此其函式回傳值型別應為整數指標型別 `int *`。

解構元 `~stutClass()` 則用於釋放在成員函式 `setNumber()` 中所配置的記憶體空間；因此，其函式架構如下所示。

```

1  ~stutClass(string id)
2  {
3      delete[]score；
4  }
```

執行結果

輸入一年 A 班的人數：33
輸入一年 B 班的人數：28

一年 A 班成績：

91	54	52	92	66	91	52	72	70	51
47	96	40	67	53	76	66	86	84	63
93	54	47	53	83	67	70	41	60	40
52	63	89							

一年 B 班成績：

70	75	73	65	96	46	81	98	75	51
47	85	71	60	75	81	91	95	72	66
75	62	71	42	78	68	78	57		

請按任意鍵繼續 . . .

一年 B 班釋放記憶體

一年 A 班釋放記憶體

程式碼列表

```

1  #include <iostream>
2  #include <iomanip>
3  #include <time.h>
4  using namespace std;
5
6  class stutClass
7  {
8      private:
9          int stutNum; // 學生人數
10         int* score; // 學生成績
11
12     public:
13         string clsID; // 班級名稱
14
15         stutClass(string id) // 建構元
16         {
17             clsID = id;
18             score = NULL;
19             stutNum = 0;
20         }
21

```

```

22     bool setNumber(int num) // 設定學生數
23     {
24         if (num <= 0)
25             return false;
26
27         stutNum = num;
28         score = new int[stutNum]; // 動態配置記憶體
29         if (score == NULL)
30         {
31             stutNum = 0;
32             return false;
33         }
34         return true;
35     }
36
37     void setScore() // 設定成績
38     {
39         if (score != NULL)
40             for (int i = 0; i < stutNum; i++)
41                 score[i] = rand() % 61 + 40;
42     }
43
44     int getNumber() // 取得學生數
45     {
46         return stutNum;
47     }
48
49     int* getScore() // 取得成績的陣列
50     {
51         return score;
52     }
53
54     ~stutClass() // 解構元
55     {
56         if (score != NULL)
57         {
58             delete[]score;
59             cout << clsID << " 釋放記憶體 " << endl;
60         }
61     }
62 };
63

```

```

64 int main()
65 {
66     stutClass clsA("一年 A 班"), clsB("一年 B 班");
67     int num;
68     int* score;
69
70     srand((unsigned)time(NULL));
71
72     //----- 設定各班的資料 -----
73     cout << "輸入 " << clsA.clsID << " 的人數：";
74     cin >> num;
75     clsA.setNumber(num);
76     clsA.setScore();
77
78     cout << "輸入 " << clsB.clsID << " 的人數：";
79     cin >> num;
80     clsB.setNumber(num);
81     clsB.setScore();
82
83     //----- 顯示所有學生的成績 -----
84     score=clsA.getScore();
85     num = clsA.getNumber();
86     cout << endl << clsA.clsID << " 成績：" << endl;
87     for (int i = 1; i <= num; i++)
88     {
89         cout << setw(4) << score[i-1] << " ";
90
91         if ((i % 10) == 0)
92             cout << endl;
93     }
94     cout << endl;
95
96     score = clsB.getScore();
97     num = clsB.getNumber();
98     cout << endl << clsB.clsID << " 成績：" << endl;
99     for (int i = 1; i <= num; i++)
100    {
101        cout<< setw(4)<< score[i-1] << " ";
102
103        if ((i % 10) == 0)
104            cout << endl;
105    }
106    cout << endl;
107
108    system("pause");
109 }

```

程式講解

1. 程式碼第 1-4 行引入所需的標頭檔與宣告使用 `std` 命名空間。
2. 程式碼第 6-62 行定義類別 `stutClass`。第 8-10 行為類別的 `private` 區段，此區段宣告了 2 個資料成員：整數型別的 `stutNum`，用於表示學生人數；以及整數指標 `score`，用於儲存學生的成績。第 12-61 行為類別的 `public` 區段，此區段宣告了 1 個資料成員：字串型別的 `clsID`，表示班級名稱；以及其餘的成員函式，包括建構元與解構元。
3. 第 15-20 行為類別的建構元，並接收 1 個字串型別的參數 `id`，用於設定班級名稱。第 17-19 行分別設定資料成員的初始值。第 22-35 行為成員函式 `setNumber()` 的函式本體，並接收 1 個整數型別的參數 `num`，用於設定學生人數。

第 27 行設定資料成員 `stutNum` 的值等於參數 `num`，第 28 行使用 `new` 配置記憶體空間給指標變數 `score`；第 29-33 行判斷若記憶體配置失敗，則將學生人數 `stutNum` 重新設定為 0，並回傳 `false`；否則第 34 行回傳 `true`。

4. 程式碼第 37-42 行為類別的成員函式 `setScore()`，用於設定所有學生的成績。第 39 行判斷指標變數 `score` 若不等於 `NULL`，才表示已經配置過記憶體空間，因此第 40-41 行才使用 `for` 重複敘述替 `stutNum` 位的學生設定成績：每位學生的成績 `score[i]` 設定介於 40-100 之間的亂數作為分數。
5. 程式碼第 44-47 行為類別的成員函式 `getNumber()`，用於回傳學生人數 `stutNum`。第 49-52 行則為成員函式 `getScore()`，用於回傳學生的成績 `score`。由於資料成員 `score` 為指標變數，因此函式 `getScore()` 的函式回傳型別也是整數指標的型別。
6. 程式碼第 54-61 行為類別的解構元。第 56 行判斷若資料成員 `score` 不等於 `NULL`，表示已經配置了記憶體空間，所以第 58 行使用 `delete` 釋放 `score` 所占有的記憶體空間；因為 `score` 是一維陣列的形式，所以在 `delete` 指令之後要加上 `[]`。
7. 開始於 `main()` 主函式中撰寫程式。程式碼第 66 行宣告 `stutClass` 類別的物件 `clsA` 與 `clsB`，並且都傳入各自的班級名稱作為呼叫建構元的引數。第 67 行宣告整數變數 `num` 作為學生人數，第 68 行宣告整數型別的指標變數 `score`，作為接收學生的成績。第 70 行使用 `srand()` 函式初始化亂數產生器。
8. 程式碼第 73 行使用物件 `clsA` 的資料成員 `clsID` 顯示班級名稱，並顯示輸入學生人數的提示訊息。第 74 行讀取所輸入的學生人數，並儲存於變數 `num`。第 75 行呼叫物件 `clsA` 的成員函式 `setNumber()` 並傳入變數 `num` 作為引數，設定一年 A 班的學生人數。第 76 行呼叫物件 `clsA` 的成員函式 `setScore()` 設定每一位學生的成績。
9. 程式碼第 78-81 行與第 73-76 行的作用相同，用於設定一年 B 班的學生人數、設定每位學生的成績。

10. 程式碼第 84-94 行用於顯示一年 A 班的學生成績。第 84 行呼叫物件 `clsA` 的成員函式 `getScore()` 取得學生成績，並儲存於指標變數 `score`。第 85 行呼叫物件 `clsA` 的成員函式 `getNumber()` 取得學生人數，並儲存於變數 `num`。

第 87-93 行使用 `for` 重複敘述顯示每一位學生的成績 `score[i-1]`；因為迴圈變數 `i` 的初始值為 1，因此 `score` 陣列的索引位址為 `i-1`。爲了不讓成績顯示過長，所以第 91-92 行設定每顯示 10 個成績之後換行顯示學生成績。

11. 程式碼第 96-106 行用於顯示一年 B 班的學生成績，做法與顯示一年 A 班的方式相同。

28-3 複製建構元

複製建構元或稱爲拷貝建構元（Copy constructor）是一種特別的建構元，專門用於處理物件的複製。

預設的複製建構元

例如以下範例；程式碼第 1-6 行定義類別 `myClass`，在 `public` 區段內宣告 2 個資料成員：整數型別的變數 `a` 與 `b`，初始值分別爲 1 與 2（如果沒有設定初始值，執行時第 11、12 行會發生錯誤）。

```

1  class myClass
2  {
3      public:
4          int a = 1;
5          int b = 2;
6  };

```

接著是 `main()` 主函式。程式碼第 10-11 行宣告類別 `myClass` 的物件 `clsA` 與 `clsB`。在類別 `myClass` 的定義中並沒有提供任何的建構元，但第 11 行卻可以把物件 `clsA` 設定給另一個物件 `clsB` 作為初始值，而第 12 行也可以把一個物件 `clsA` 作為另一個物件 `clsC` 的引數；這是因爲自動執行了 Visual Studio C++ 編譯器預設的複製建構元的緣故。

由此可知，複製建構元用來複製物件的資料成員：當宣告物件時，若以另一個物件作為初始值時，複製建構元會自動執行。因此，第 14-15 行分別顯示 1 與 2；第 17-18 行也是顯示相同的結果。

```

8  int main()
9  {
10     myClass clsA;
11     myClass clsB = clsA;
12     myClass clsC(clsA);
13
14     cout << clsB.a << endl; // 顯示 1
15     cout << clsB.b << endl; // 顯示 2
16
17     cout << clsC.a << endl; // 顯示 1
18     cout << clsC.b << endl; // 顯示 2
19
20     system("pause");
21 }

```

自動呼叫了預設的複製建構元

現在提供類別 `myClass` 的建構元，如下程式碼第 7-12 行所示：

```

1  class myClass
2  {
3      public:
4          int a;
5          int b;
6
7          myClass(int a, int b)
8          {
9              cout << " 建構元被呼叫了 " << endl;
10             this->a = a;
11             this->b = b;
12         }
13 };

```

主函式 `main()` 也修改為較簡單的內容，如下所示。程式碼第 17 行宣告 `myClass` 類別的物件 `clsA`，並且傳入引數 5 與 10；因此，會執行第 7-12 行的建構函式，所以會顯示：" 建構元被呼叫了 "。

第 18 行宣告 `myClass` 類別的物件 `clsB`，並且以物件 `clsA` 作為引數。但是類別 `myClass` 並沒有相對應的建構元（只接收一個類別 `myClass` 作為參數），因此會自動呼叫 Visual Studio C++ 編譯器預設的複製建構元，所以不會執行第 7-12 行的建構元。第 20-21 行分別顯示 5 與 10。

```

15 int main()
16 {
17     myClass clsA(5,10); ← 呼叫建構元
18     myClass clsB(clsA); ← 呼叫預設的複製建構元
19
20     cout << clsB.a << endl; // 顯示 5
21     cout << clsB.b << endl; // 顯示 10
22
23     system("pause");
24 }

```

因此，此範例會輸出如下的顯示結果：

建構元被呼叫了

5
10

自訂複製建構元

既然已經有了預設的複製建構元可以使用，何必再設計自訂的複製建構元？預設的複製建構元只能處理類別裡的資料成員的初始值設定，若是類別裡所自行設計的建構元除了設定資料成員的初始值之外，還包含了其他的處理；那麼由上一個範例可以知道，當以物件做為另一個物件宣告的初始值時（上個範例的程式碼第 18 行），並不會去執行自訂的建構元；如此一來，就會發生物件有可能並沒有按照預設的方式初始化。

如下範例：類別 `myClass` 只有一個 `public` 區段，第 4 行宣告了 1 個字元指標的資料成員 `str`。第 6-11 行為建構元，接收字元指標型別的參數 `s`。第 8 行顯示呼叫建構元的訊息，第 9 行使用 `new` 指令替資料成員 `str` 配置 10 個位元組的記憶體空間，第 10 行使用 `strcpy_s()` 函式將參數 `s` 複製到資料成員 `str`。

第 13-16 行為解構元，第 15 行使用 `delete` 指令釋放在建構元裡配置給資料成員 `str` 的記憶體空間。

```

1 class myClass
2 {
3     public:
4         char *str;
5
6         myClass(const char* s)
7         {

```

```

8         cout << " 建構元被呼叫了 " << endl;
9         str = new char[10];
10        strcpy_s(str, strlen(s)+1, s);
11    }
12
13    ~myClass()
14    {
15        delete[]str;
16    }
17 };

```

接著是 `main()` 主函式。程式碼第 21 行宣告類別 `myClass` 的物件 `clsA`，並且傳入字串 "Hello" 作為建構元的參數。第 22 行宣告類別 `myClass` 的物件 `clsB`，並以物件 `clsA` 作為初始值；因為 `clsB` 是以物件 `clsA` 作為初始值，所以會使用預設的複製建構元。

第 24-25 行顯示物件 `clsA` 的資料成員 `str`，並顯示其記憶體位址；第 27-28 行顯示物件 `clsB` 的資料成員 `str`，並顯示其記憶體位址。

```

19 int main()
20 {
21     myClass clsA("Hello");
22     myClass clsB(clsA);
23
24     cout <<"clsA.str= " << clsA.str << endl;
25     cout << "clsA.str 的位址 = " << (void*)(clsA.str) << endl;
26
27     cout << "clsB.str= " << clsB.str << endl;
28     cout << "clsB.str 的位址 = " << (void*)(clsB.str) << endl;
29
30     system("pause");
31 }

```

程式碼第 24-25 行、27-28 行的輸出如下所示；注意！程式結束時會發生錯誤。

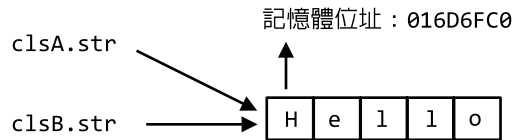
```

建構元被呼叫了
clsA.str= Hello
clsA.str 的位址 = 016D6FC0
clsB.str= Hello
clsB.str 的位址 = 016D6FC0

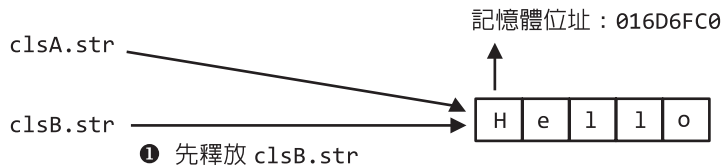
```

兩者的記憶體位址竟然相同

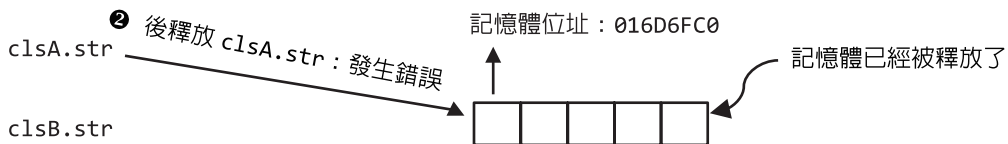
因為物件 `clsB` 是以物件 `clsA` 作為初始值，自動執行的是預設的複製建構元，而不是第 6-11 行的建構元，所以請注意物件 `clsA` 與物件 `clsB` 的資料成員 `str` 的位址是一樣的。換句話說，物件 `clsB` 的資料成員 `str` 並沒有自己的記憶體空間，而只是指向物件 `clsA` 的資料成員 `str` 的記憶體位址；如下圖所示。



在程式結束時物件 `clsB` 先被釋放，因此會執行第 13-16 行的解構元，所以資料成員 `ptr` 所指的位址的記憶體空間（物件 `clsA` 的資料成員 `str`）會被釋放；如下圖所示：



接著是釋放物件 `clsA`，所以也會執行解構元，然而物件 `clsA` 的資料成員 `str` 所佔用的記憶體空間已經被釋放過了，無法再次被釋放，所以造成了錯誤；如下圖所示。



為了避免發生如此的錯誤，就要自行設計複製建構元；複製建構元的語法如下所示：

```
類別名稱 (const 類別名稱 & 物件名稱)
{
    程式敘述;
}
```

例如，上述範例中的類別 `myClass`，新增第 13-18 行的自訂的複製建構元，如下所示；接收一個參考型別的物件 `obj`。在自訂的複製建構元中，第 16 行為資料成員 `str` 配置記憶體，並將物件參數 `obj` 的資料成員 `str` 複製到自己的資料成員 `str`。

```
1 class myClass
2 {
3     public:
4         char *str;
5
```

```

6      myClass(const char* s)
7      {
8          cout << " 建構元被呼叫了 " << endl;
9          str = new char[10];
10         strcpy_s(str, strlen(s)+1, s);
11     }
12
13     myClass(const myClass& obj)
14     {
15         cout << " 複製建構元被呼叫了 " << endl;
16         str = new char[10];
17         strcpy_s(str, strlen(obj.str) + 1, obj.str);
18     }
19
20     ~myClass()
21     {
22         delete[]str;
23     }
24 };

```

← 自訂的複製建構元

重新執行此範例，輸出的結果如下所示；並且程式能正常結束。

```

建構元被呼叫了
複製建構元被呼叫了 ← 物件 clsB 的初始化，呼叫的是自訂的複製建構元。
clsA.str= Hello
clsA.str 的位址 = 0082E0D0
clsB.str= Hello
clsB.str 的位址 = 0082E3E0

```

clsA.str 與 clsB.str 的記憶體位址已經不一樣了，2 個資料成員有各自的記憶體空間。

因此，只要在類別中的資料成員使用到動態的記憶體配置，就要提供自訂的複製建構元來處理相同的事情，以避免造成上述的問題。此外，若函式呼叫是以類別物件作為引數，也需要使用複製建構元；請參考分析與討論。

🔄 練習 3：統計骰子不同點數的次數

2 顆骰子各擲 100 次，寫一程式統計 2 顆骰子各自擲到不同點數的次數。骰子相關的資料與處理函式使用類別表示。使用此類別建立 2 顆骰子物件，並且使用第 1 顆骰子作為第 2 顆骰子的初始值。

解說

若代表骰子的類別為 `Dics`，則需要有 2 個資料成員：記錄擲出不同點數的次數、骰子的名稱，例如："骰子"、"骰子 1" 等。用於記錄擲出不同點數的次數的資料成員，則使用整數指標；此 2 個資料成員可以在執行建構元時同時設定初始值。

因為第 2 顆骰子必須使用第 1 顆骰子的資料進行初始化，所以必須提供複製建構元；以及再加上類別裡常用的成員函式：取得資料成員的函式、解構元、其他的計算函式等，則類別 `Dics` 的架構如下所示：

```
class Dics
{
    private:
        int *pips;        // 記錄骰子擲出不同點數的次數
        string title;     // 骰子的名稱

    public:
        Dics() {…}        // 建構元
        Dics(const Dics &obj) {…} // 複製建構元
        string getName() {…} // 取得骰子的名稱
        void count() {…}    // 模擬擲骰子、記錄擲出不同點數的次數
        int *getPips() {…}  // 取得資料成員 pips
        ~Dics() {…}        // 解構元
}
```

執行結果

2 顆骰子各擲 100 次的各點次數：

骰子 1: 1 點: 13 2 點: 21 3 點: 15 4 點: 16 5 點: 24 6 點: 11

骰子 2: 1 點: 22 2 點: 13 3 點: 14 4 點: 16 5 點: 15 6 點: 20

程式碼列表

```
1 #include <iostream>
2 #include <time.h>
3 using namespace std;
4
5 class Dics
6 {
7     private:
8         int *pips;
```

```
9         string title;
10
11     void clearPips()
12     {
13         pips = new int[6];
14         for (int i = 0; i < 6; i++)
15             pips[i] = 0;
16     }
17
18     public:
19     Dics()
20     {
21         clearPips();
22         title = " 骰子 ";
23     }
24
25     Dics(const Dics& obj)
26     {
27         clearPips();
28         title = obj.title;
29     }
30
31     string getName()
32     {
33         return title;
34     }
35
36     void count()
37     {
38         int no;
39
40         for (int i = 0; i < 100; i++)
41         {
42             no = rand() % 6;
43             pips[no]++;
44         }
45     }
46
47     int* getPips()
48     {
49         return pips;
50     }
```

```

51
52     ~Dics()
53     {
54         delete[] pips;
55     }
56 };
57
58 void showData(Dics &dics,int no)
59 {
60     int* pips;
61
62     cout << dics.getName() << no << ": ";
63     pips = dics.getPips();
64     for (int i = 0; i < 6; i++)
65         cout << pips[i] << " ";
66     cout << endl;
67 }
68
69 int main()
70 {
71     Dics dics1;
72     Dics dics2(dics1);
73
74     srand((unsigned)time(NULL));
75     dics1.count();
76     dics2.count();
77
78     cout << "2 顆骰子各擲 100 次的各點次數：" << endl;
79     showData(dics1,1);
80     showData(dics2,2);
81
82     system("pause");
83 }

```

程式講解

1. 程式碼第 1-3 行引入所需的標頭檔與宣告使用 `std` 命名空間。
2. 程式碼第 5-56 行定義骰子類別 `Dics`，在 `private` 區段有 2 個資料成員與 1 個成員函式。第 8 行宣告整數指標 `pips`，用於儲存骰子擲出不同點數的次數。第 9 行宣告字串型別的變數 `title`，用於儲存骰子的名稱。第 11-16 行為成員函式 `clearPips()`，用於設定整數指標 `pips` 的初始值。第 13 行替資料成員 `pips` 配置記

記憶體空間（骰子的 6 個不同的點數），`pips[0]` 表示擲出 1 點的次數、`pip[1]` 表示擲出 2 點的次數，以此類推。第 14-15 行將 `pips` 的初始值設定為 0。

3. 程式碼第 18-55 為骰子類別 `Dics` 的 `public` 區段，區段內都是成員函式。第 19-23 行為建構元，第 21 行呼叫成員函式 `clearPips()` 替資料成員 `pips` 配置記憶體空間與設定初始值。第 22 行設定骰子的名稱等於 " 骰子 "。第 25-29 行為自訂的複製建構元，其所做的事情與建構元相同。
4. 程式碼第 31-34 行為成員函式 `getName()`，用於回傳骰子的名稱 `title`。第 36-45 行為成員函式 `count()`，用於統計擲出不同骰子點數的次數。第 40-44 行使用 `for` 重複敘述模擬擲出 100 次的骰子。第 42 行使用函式 `rand()` 模擬擲出的骰子點數，並儲存於變數 `pips`。雖然骰子的點數為 1-6 點，其擲出點數分別儲存於 `pips[0]-pips[5]`，所以第 42 行特意讓產生的亂數介於 0-5，並儲存於變數 `no`；如此才剛好符合 `pips` 的索引位置。第 43 行對擲出的點數 `pips[no]` 加 1，表示次數累加 1 次。
5. 程式碼第 47-50 行為成員函式 `getPips()`，用於回傳資料成員 `pips`。因為資料成員 `pips` 的資料型別為整數指標，所以函式回傳值型別為也是整數指標。第 52-55 行為解構元，用於釋放資料成員 `pips` 所佔用的記憶體空間。
6. 程式碼第 58-67 行為自訂函式 `showData()`，用於顯示 2 顆骰子各自擲出不同點數的統計次數。此自訂函式接收 2 個參數；第 1 個參數是參考型別的 `Dics` 物件 `dics`，用於顯示其資料成員 `pips`。第 2 個參數為整數型別 `no`，表示要顯示第幾顆骰子。

第 62 行顯示 `dics.getName()` 的回傳值與參數 `no`，用來顯示這是第幾顆骰子。第 63 行呼叫參數 `dics` 的成員函式 `getPips()` 取得資料成員 `pips`，並儲存於變數 `pips`。第 64-65 行使用 `for` 重複敘述，顯示第 `no` 顆骰子不同點數的擲出次數 `pips[i]`。

7. 開始於 `main()` 主函式中撰寫程式。程式碼第 71 行宣告 `Dics` 類別的物件 `dics1`，第 72 行宣告 `Dics` 類別的物件 `dics2`，並使用物件 `dics1` 作為引數；因此，會自動執行第 25-29 的複製建構元。第 74 行呼叫 `srand()` 函式初始化亂數產生器。

第 75-76 行分別呼叫物件 `dics1` 與 `dics2` 的成員函式 `count()`，統計骰子擲出不同點數的次數。第 79-80 行呼叫自訂函式 `showData()`，並分別傳入 `dics1` 與 `dics2` 作為引數，以及第幾顆骰子的編號，並顯示 2 顆骰子各自擲出不同點數的次數。

三、範例程式解說

1. 建立專案，程式碼第 1-2 行引入所需要的標頭檔與宣告使用 `std` 命名空間。

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

2. 程式碼第 4-67 行定義販售咖啡的類別 `Stand`。第 6-10 行為 `private` 區段，此區段內宣告了 4 個資料成員。第 7 行宣告字串指標 `coffee`，用於儲存咖啡的名稱。第 8 行宣告整數指標 `price`，用於儲存咖啡的單價。第 9 行宣告整數指標 `total`，用於儲存咖啡的販售數量。第 10 行宣告整數變數 `number`，用於儲存販售幾種咖啡。

```
4 class Stand
5 {
6     private:
7         string* coffee;    // 咖啡名稱
8         int* price;        // 咖啡單價
9         int* total;        // 販售數量
10        int number;        // 販賣的咖啡總類
```

3. 程式碼第 12-66 行為類別 `Stand` 的 `public` 區段。第 13 行宣告字串型別的資料成員 `name`，用於儲存攤位的名稱。第 15-30 行為類別的建構元，並接收 2 個參數：字串型別的參數 `str`，以及整數型別的參數 `num`；分別代表攤位名稱以及販售多少種咖啡。

第 17-20 行設定資料成員的初始值，第 21 行判斷若參數 `num` 大於 0，則第 22-27 行配置適當的記憶體空間給資料成員 `coffee`、`price` 與 `total`；否則將資料成員 `number` 再設定為 0。因此，也可以藉由資料成員 `number` 判斷若等於 0，表示資料成員 `coffee`、`price` 與 `total` 尚未初始化，無法被使用。

```
12     public:
13         string name;    // 店名
14
15         Stand(string str,int num) // 建構元
16         {
17             name = str;
18             coffee = NULL;
19             price = NULL;
20             total = NULL;
21             if (num > 0)
22             {
23                 number = num;
```

```

24             coffee = new string[number];
25             price = new int[number];
26             total = new int[number];
27         }
28         else
29             number = 0;
30     }

```

4. 程式碼第 32-35 行為成員函式 `getCoffee()`，用於回傳第 `index` 個的咖啡名稱 `coffee[index]`。第 37-40 行為成員函式 `getPrice()`，用於回傳第 `index` 個的咖啡單價 `price[index]`。第 42-45 行為成員函式 `getTotal()`，用於回傳第 `index` 個咖啡的銷售數量 `total[index]`。

```

32     string getCoffee(int index) // 取得咖啡名稱
33     {
34         return coffee[index];
35     }
36
37     int getPrice(int index) // 取得咖啡的單價
38     {
39         return price[index];
40     }
41
42     int getTotal(int index) // 取得咖啡的販售數量
43     {
44         return total[index];
45     }

```

5. 程式碼第 47-51 行為成員函式 `setCoffee()`，用於設定咖啡的名稱與單價；因此，接受 3 個參數：第 1 個參數用於表示欲設定第 `index` 種咖啡的資料；第 2 個參數 `name` 與第 3 個參數 `price`，分別代表咖啡名稱與單價。第 49-50 分別將咖啡名稱 `name` 與單價 `price` 設定給類別的資料成員 `coffee[index]` 與 `this->price[index]`。

第 53-56 行為成員函式 `setTotal()`，用於設定咖啡的銷售數量；參數 `index` 與 `total`，分別代表欲設定第 `index` 種咖啡與其銷售數量。第 55 行將咖啡的銷售數量 `total` 設定給第 `index` 種咖啡的銷售數量 `this->coffee[index]`。

第 58-66 行為解構元，第 60-65 行判斷咖啡數量若大於等於 1，則將在建構元中所配置給資料成員 `coffee`、`price` 與 `total` 的記憶體空間釋放。

```

47     void setCoffee(int index,string name,int price)
48     { // 設定咖啡名稱與單價
49         coffee[index] = name;
50         this->price[index] = price;
51     }
52
53     void setTotal(int index, int total) // 設定販售數量
54     {
55         this->total[index] = total;
56     }
57
58     ~Stand() // 解構元
59     {
60         if (number >= 1)
61         {
62             delete[] coffee;
63             delete[] price;
64             delete[] total;
65         }
66     }
67 };

```

6. 開始於 `main()` 主函式中撰寫程式。程式碼第 71-74 行宣告變數，第 71 行宣告整數型別的 `coffeeNum`，初始值等於 3；表示一共有 3 種咖啡。第 72 行宣告 `Stand` 類別的物件 `coff`，並以字串 " 咖啡小館 " 與 `coffeeNum` 作為初始值之引數。第 73-74 行所宣告之變數 `price`、`total` 與 `name`，分別表示咖啡的單價、銷售數量與名稱。

```

71 int coffeeNum = 3;
72 Stand coff(" 咖啡小館 ", coffeeNum);
73 int price, total;
74 string name;

```

7. 程式碼第 77-82 行用於設定 3 種咖啡的名稱與單價。第 77-82 行為 `for` 重複敘述，因為 `coffeeNum` 等於 3，所以 `for` 重複敘述會執行 3 次。第 80 行讀取所輸入的咖啡名稱與單價，並儲存於變數 `name` 與 `price`，第 81 行呼叫物件 `coff` 的成員函式 `setCoffee()` 設定第 `i` 種咖啡的名稱與單價。

```

76 // 輸入咖啡的名稱與單價
77 for (int i = 0; i < coffeeNum; i++)
78 {
79     cout << " 輸入咖啡 " << i + 1 << " 的名稱與售價 : ";
80     cin >> name >> price;
81     coff.setCoffee(i, name, price);
82 }

```

8. 程式碼第 85-90 行用於設定 3 種咖啡的銷售數量。第 85-90 行為 for 重複敘述，因為 coffeeNum 等於 3，所以 for 重複敘述會執行 3 次。第 88 行讀取所輸入的咖啡銷售數量，並儲存於變數 total，第 89 行呼叫物件 coff 的成員函式 setTotal() 設定第 i 種咖啡的銷售數量。

```

84 // 輸入咖啡的販售數量
85     for (int i = 0; i < coffeeNum; i++)
86     {
87         cout << coff.getCoffee(i) << " 的販售數量 : ";
88         cin >> total;
89         coff.setTotal(i, total);
90     }

```

9. 程式碼第 94 行先顯示攤位的名稱 coff.Name。第 95-100 行為 for 重複敘述，用於顯示 3 種咖啡的資料。第 97 行呼叫物件 coff 的成員函式 getCoffee()，顯示第 i 種咖啡的名稱，第 98 行呼叫物件 coff 的成員函式 getPrice()，顯示第 i 種咖啡的單價，第 99 行呼叫物件 coff 的成員函式 getTotal()，顯示第 i 種咖啡的銷售數量。

```

92 // 顯示咖啡販售的相關資訊
93 system("cls");
94 cout << "==== " << coff.name << " =====> << endl;
95 for (int i = 0; i < coffeeNum; i++)
96 {
97     cout << coff.getCoffee(i) << " 咖啡，單價 : ";
98     cout << coff.getPrice(i) << "，賣了 ";
99     cout << coff.getTotal(i) << " 杯。" << endl;
100 }
101
102 system("pause");

```

重點整理

1. 建構元在物件被建立時會自動執行，通常是用來初始化物件的初始狀態、設定資料成員初始值。
2. 解構元在物件被釋放時會自動執行。在類別中若有動態記憶體配置，除了在成員函式中需要負責自行釋放之外，也能在解構元中釋放尚未被釋放的記憶體空間。
3. 使用物件做為另一個物件宣告時的初始值時，並且在建構元中有處理記憶體配置時，就需要提供自訂的複製建構元，處理相同的動態記憶體配置。

分析與討論

1. 練習 1 中比較「誰的購買金額比較多」的成員函式 `comsMore()`，可以有不同的設計方法。設計此成員函式需要特別留意一點，以物件作為引數傳遞的預設方式為傳值呼叫，所以引數和參數只是內容相同的 2 個獨立的物件。

假設這樣的情形：自己 `this` 和參數 `com` 比較購買金額之後，若參數 `com` 的購買金額比較高，因此要回傳參數 `com`。但參數 `com` 其實並不是原來的那一個引數，而只是引數的一份拷貝而已；雖然結果並沒有差別（因為引數和參數的內容相同）。因此，若想要回傳的參數 `com` 是原本所傳入的那個引數，則可以使用傳址呼叫或是參考呼叫的方式。

2. 練習 3 中的程式碼第 79-80 行呼叫 `showData()` 顯示擲骰子的資料，並傳遞骰子物件作為引數，而第 58-67 自訂函式 `showData()` 則以參考呼叫的方式接收此物件。若以傳值呼叫的方式傳遞引數，如下所示：

```
59 void showData( Dics dics ,int no)
60 {
    :      :
68 }
```

則輸出的結果如下所示；2 顆骰子所有不同點數的擲出次數都等於 0。

```
2 顆骰子各擲 100 次的各點次數：
骰子 1: 1 點: 0 2 點: 0 3 點: 0 4 點: 0 5 點: 0 6 點: 0
骰子 2: 1 點: 0 2 點: 0 3 點: 0 4 點: 0 5 點: 0 6 點: 0
```

這是因為若使用傳值呼叫的方式傳遞物件時，會自動呼叫複製建構元：因為傳值呼叫的特性是將物件複製一份相同的內容給自訂函式作為參數，所以若類別裡有提供複製建構元，便會自動執行複製建構元。而複製建構元的程式碼的 27 行執行 `clearPips()` 函式，所以 `pips` 的內容都被設定為 0；因此，才會顯示所有擲出不同點數的次數都等於 0。

因此，當類別中有使用到動態記憶體配置的程式敘述時，當以類別物件作為引數或是參數，並以傳值呼叫的方式呼叫函式，則類別裡必須提供複製建構元來處理記憶體配置的事情，否則程式會發生錯誤；請參考本章習題第 5、6 題。

3. 定義類別時，除了建構元、成員函式可以多載之外，'+', '-', '>', '<', '=' 等運算子也能多載；稱為運算子多載。這些運算子多載之後，並不會改變原有的功能，但能增加自行設計的額外作用。

由於在類別裡將運算子多載之後容易和一般的數值運算混淆，所以在實際運用上並不見得實用，並且運算子多載所要做到的功能，也可以自行設計成員函式來達到相同的作用；因此，運算子多載就不見得一定要使用了。

然而，在某些特定的情形之下則需要使用運算子多載，否則會出現錯誤的情形，請參考範例 29 的討論與說明第 2 點說明；運算子多的範例如下所式。

在類別 `myClass` 的程式碼第 9-12 行為 '>' 運算子的多載。此運算子多載要比較的是 `int` 型別的資料成員 `num` 的大小，因此運算子多載函式的函式回傳值型別也是 `int` 型別；之後接著關鍵字 `operator`，然後是要多載的運算子 '>'；其參數為參考型別的類別物件。程式碼的 11 行則是真正要進行判斷比較的程式敘述。

```

1  class myClass
2  {
3      public:
4          int num=0;
5          myClass(int n) :num(n)
6          {
7          }
8
9          int operator >(myClass& obj)
10         {
11             return num > obj.num;
12         }
13 };

```

← '>' 運算子多載

接下來是主函式 `main()`，程式碼第 17 行分別宣告 `myClass` 類別的物件 `clsA` 與 `clsB`，並傳入數值 10 與 20 作為引數。第 19 行判斷物件 `clsA` 是否大於 `clsB`，因為類別 `myClass` 提供了 '>' 運算子的多載，因此會執行類別的第 9-12 行的 '>' 運算子的多載函式。因此，此程式的執行結果會顯示：`"clsA.num < clsB.num"`。

```

15 int main()
16 {
17     myClass clsA(10), clsB(20);
18
19     if (clsA > clsB)
20         cout << "clsA.num > clsB.num" << endl;
21     else
22         cout << "clsA.num < clsB.num" << endl;
23
24     system("pause");
25 }

```

4. 使用傳址呼叫或參考呼叫的物件，並不會自動呼叫複製建構元，請參考練習檔案 `extra_a`。
5. 將物件設定給另一個物件時，例如物件 `clsA` 設定給物件 `clsB`：`clsB=clsA`，並不會自動執行複製建構元。若類別裡的資料成員有使用動態記憶體配置，則當這些物件不再使用時，便會發生重複釋放記憶體的錯誤請，請參考練習檔案 `extra_b`。

如下程式碼第 10 行所示，在類別 `myClass` 的建構元裡配置記憶體空間給資料成員 `num5`。第 14-17 行為解構元，用於釋放 `num5` 所佔用的記憶體空間。

```

1  class myClass
2  {
3      public:
4          int a;
5          int num3[3] = { 11,12,13 };
6          int* num5;
7
8          myClass()
9          {
10             num5 = new int[5];
11             cout << "呼叫建構元" << endl;
12         }
13
14         ~myClass()
15         {
16             delete[] num5;
17         }
18 };

```

在主函式 `main()` 中，程式碼第 22-23 分別宣告 `myClass` 類別的物件 `clsA` 與 `clsB`。第 25-26 行分別顯示資料成員 `clsA.num5` 與 `clsB.num5` 的位址，此 2 個資

料成員的位址並不會一樣。然而，當執行第 28 行 `clsB=clsA` 之後，`clsB` 仍然會有自己的資料成員 `a` 與 `num3`，但是資料成員 `num5` 變成指向了 `clsA.num5`；因此，第 29-30 行再次顯示 `clsB.num5` 的位址時，便會發現其位址已經和 `clsA.num5` 的位址一樣了。

```

20 int main()
21 {
22     myClass clsA;
23     myClass clsB;
24
25     cout << "clsA.num5 的位址：" << (void*)clsA.num5 << endl;
26     cout << "clsB.num5 的位址：" << (void*)clsB.num5 << endl;
27
28     clsB = clsA;
29     cout << "clsB=clsA 之後，clsB.num5 的位址：" <<
30         (void*)clsB.num5 << endl;
31
32     system("pause");
33 }

```

當程式結束釋放物件 `clsA` 與 `clsB`，資料成員 `num5` 便會被釋放 2 次，因此造成了錯誤。此程式的輸出結果如下所示：

```

呼叫建構元
呼叫建構元
clsA.num5 的位址：000002A88813F4D0 ← 記憶體位址相同
clsB.num5 的位址：000002A88813FB60
clsB=clsA 之後，clsB.num5 的位址：000002A88813F4D0

```

要修正這種錯誤，可以在類別裡增加 '=' 指定運算子多載來解決這個問題，如下所示；請參考練習檔案 `extra_c`。

```

class myClass
{
public:
    :
    void operator = (const myClass& obj)
    {
        a = obj.a;
        memcpy(num3, obj.num3, sizeof(int) * 3);
        memcpy(num5, obj.num5, sizeof(int) * 5);
    }
    :
};

```

若是使用指標型別來宣告物件，把物件指定給指標物件，則不會發生上述的錯誤；如下所示。請參考練習檔案 `extra_d`。

```
myClass clsA;
myClass* clsB;

clsB = &clsA;
```

程式碼列表

```
1 #include <iostream>
2 using namespace std;
3
4 class Stand
5 {
6     private:
7         string* coffee; // 咖啡名稱
8         int* price;      // 咖啡單價
9         int* total;      // 販售數量
10        int number;      // 販賣的咖啡總類
11
12    public:
13        string name;     // 店名
14
15        Stand(string str,int num) // 建構元
16        {
17            name = str;
18            coffee = NULL;
19            price = NULL;
20            total = NULL;
21            if (num > 0)
22            {
23                number = num;
24                coffee = new string[number];
25                price = new int[number];
26                total = new int[number];
27            }
28            else
29                number = 0;
30        }
31
32        string getCoffee(int index) // 取得咖啡名稱
33        {
```

```
34         return coffee[index];
35     }
36
37     int getPrice(int index) // 取得咖啡的單價
38     {
39         return price[index];
40     }
41
42     int getTotal(int index) // 取得咖啡的販售數量
43     {
44         return total[index];
45     }
46
47     void setCoffee(int index,string name,int price) // 設定咖啡名稱與單價
48     {
49         coffee[index] = name;
50         this->price[index] = price;
51     }
52
53     void setTotal(int index, int total) // 設定販售數量
54     {
55         this->total[index] = total;
56     }
57
58     ~Stand() // 解構元
59     {
60         if (number >= 1)
61         {
62             delete[] coffee;
63             delete[] price;
64             delete[] total;
65         }
66     }
67 };
68
69 int main()
70 {
71     int coffeeNum = 3;
72     Stand coff(" 咖啡小館 ", coffeeNum);
73     int price, total;
74     string name;
75
```



```
76 // 輸入咖啡的名稱與單價
77 for (int i = 0; i < coffeeNum; i++)
78 {
79     cout << " 輸入咖啡 " << i + 1 << " 的名稱與售價：";
80     cin >> name >> price;
81     coff.setCoffee(i, name, price);
82 }
83
84 // 輸入咖啡的販售數量
85 for (int i = 0; i < coffeeNum; i++)
86 {
87     cout << coff.getCoffee(i) << " 的販售數量：";
88     cin >> total;
89     coff.setTotal(i, total);
90 }
91
92 // 顯示咖啡販售的相關資訊
93 system("cls");
94 cout << "==== " << coff.name << " ====" << endl;
95 for (int i = 0; i < coffeeNum; i++)
96 {
97     cout << coff.getCoffee(i) << " 咖啡，單價：";
98     cout << coff.getPrice(i) << "，賣了";
99     cout << coff.getTotal(i) << " 杯。" << endl;
100 }
101
102 system("pause");
103 }
```

本章習題

1. 設計一個用於執行 2 個整數相加的類別，並提供多載之建構元。
2. 設計一個用於多個整數加總的類別。輸入要加總的整數數量，並設計成員函式將這些整數加總；每個整數的值以亂數的方式產生介於 1-100 之間的值。
3. 設計一個類別 `myClass`，有 2 個資料成員：`string` 型別的指標變數 `str`、整數變數 `num`；`num` 用來初使化 `str` 的容量。於類別中設計成員函式用於輸入 `num` 個字串，並儲存於變數 `str`，以及顯示 `num` 個字串。寫一程式，宣告 2 個 `myClass` 類別的指標物件 `clsA` 與 `clsB`。輸入字串的數量之後，用來初始化指標物件 `clsA`；並使用指標物件 `clsA` 初始化指標物件 `clsB`。
4. 修改範例 28，撰寫成員函式用於計算各種咖啡的販售金額，以及所有咖啡的販售金額。
5. 設計一個類別 `myClass`，用於產生指定數量的 1-100 之間的亂數，並將亂數儲存於指標成員 `number` 中。輸入欲產生亂數的個數 `num`，作為宣告 `myClass` 物件的引數；例如，宣告 `myClass` 類別的物件 `clsA`，並產生 10 個亂數：

```
myClass clsA(10);
```

類別 `myClass` 中的資料成員都宣告在 `private` 區段內，在其建構元中產生 `num` 個亂數，並儲存於陣列 `numbers` 中。寫一自訂函式 `showNumbers()`，接收一個類別物件作為參數，並顯示此參數物件的 `numbers` 內容。

6. 如同第 5 題之類別。設計一自訂函式 `genNumbers()`，於其中宣告 `myClass` 類別之物件，並將物件回傳給呼叫者。在 `main()` 主函式中呼叫自訂函式 `genNumbers()`，接收其回傳的物件，並顯示物件所產生的 10 個亂數。