

Example

29

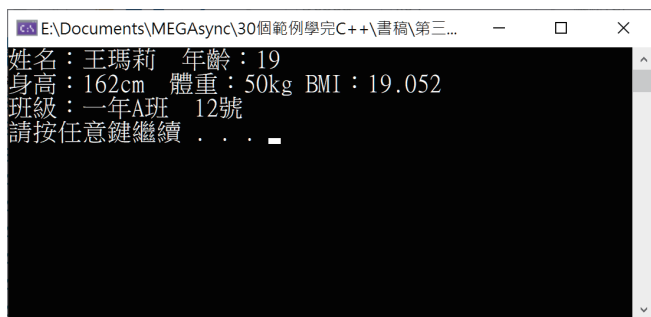
類別與物件：繼承

有以下類別：Person、Basic 與 Student。類別 Person 記錄姓名與年齡，有 2 個成員函式：setNameAge() 與 showInfo()，分別用於設定與顯示姓名與年齡。類別 Basic 記錄身高與體重，有一個純虛擬函式 getBMI() 用於計算 BMI 值。類別 Student 繼承 Person 與 Basic，並記錄學生的班級與號碼。寫一程式用於設定與顯示學生的所有資料。

一、學習目標

類別繼承（Inheritance）是物件導向程式設計的特色之一，不僅讓既有的類別得以再利用，也能透過繼承的特性，在既有的類別之上新增、修改原有的功能，進而成為新類別。如此一來，不僅能快速開發符合新需求的類別，也能讓軟體的開發如同硬體一般，利用既有的模組去設計更複雜、不同應用的新模組。

二、執行結果

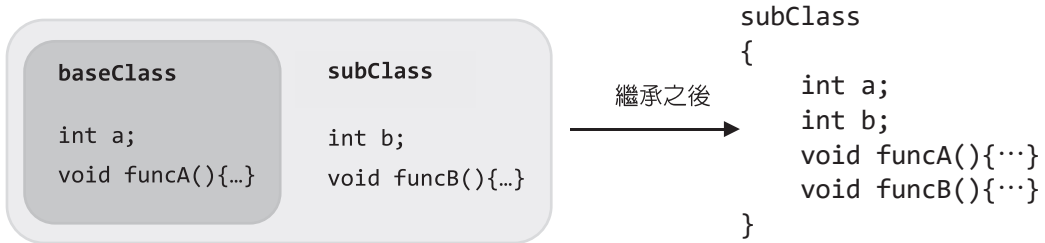


```
E:\Documents\MEGAsync\30個範例學完C++\書稿\第三...
姓名：王瑪莉 年齡：19
身高：162cm 體重：50kg BMI：19.052
班級：一年A班 12號
請按任意鍵繼續 . . .
```

29-1 繼承關係

被繼承的類別稱為基礎類別（Base class）或父類別（Parent class、Super class），繼承父類別的類別稱為衍生類別（Derived class）或是子類別（Child class、Sub class）。當 2 個類別彼此定義了繼承關係之後，父類別的資料成員與成員函式可在子類別中使用。以下是一個基本的繼承關係概念圖，用以說明父類別與子類別之間的繼承關係。

下圖中類別 `baseClass` 為父類別，有 1 個資料成員 `a` 與 1 個成員函式 `funcA()`。`subClass` 類別繼承 `baseClass` 類別，而在 `subClass` 類別自身中也有 1 個資料成員 `b`，以及 1 個成員函式 `funcB()`。當繼承之後，在 `subClass` 類別中視同有了 2 個資料成員 `a`、`b`，以及 2 個成員函式 `funcA()` 與 `funcB()`。



但根據在父類別中的資料成員與成員函式宣告在不同的存取控制區段，會在子類別中有著不一樣存取權限，以及不同的特性。此外，建構元、解構元、`friend` 函式以及運算子 `"="` 多載法無被繼承（但可以被使用）；也就是子類別必須要撰寫自己的建構元與解構元，以及運算子 `"="` 多載；`friend` 函式也要重新再宣告一次。

定義繼承關係

類別繼承的語法如下所示。在子類別的名稱之後，以分號 `:"` 連接父類別的名稱。繼承權限修飾字則有：`public`、`private` 與 `protected`，用來表示使用哪種存取權限來繼承父類別的成員；也就是決定了父類別裡的成員該如何繼承給子類別。

```
class 子類別的名稱 : 繼承權限修飾字 父類別的名稱
{
    ...
};
```

例如，有 2 個類別 `baseClass` 與 `subClass`，並且類別 `subClass` 繼承 `baseClass` 類別，則定義繼承關係的方式如下所示。程式碼第 1-4 行為類別 `baseClass` 的定義，第 6-9 行則為類別 `subClass` 的定義，並且以 `public` 繼承權限的方式繼承 `baseClass` 類別。

```
1 class baseClass
2 {
3     :
4 };
5
6 class subClass :public baseClass ← 繼承 baseClass 類別
7 {
8     :
9 };
```

例如：以下的範例中（範例檔案 01-1），類別 `subClass` 繼承 `baseClass` 類別；如下程式碼所示。在父類別 `baseClass` 中有 1 個字串型別的資料成員 `strA`，以及一個成員函式 `showMsg()`，用於顯示資料成員 `strA`。子類別 `subClass` 有 1 個字串型別的資料成員 `strB`，以及一個成員函式 `sub_showMsg()`，用於顯示資料成員 `strB` 與字串 "How are you?"。

由於類別 `subClass` 以 `public` 繼承權限的方式繼承了 `baseClass` 類別；因此，在 `subClass` 中也能直接使用 `baseClass` 類別中的資料成員 `strA`，以及成員函式 `showMsg()`。

```

1  class baseClass // 基礎類別
2  {
3      public:
4          string strA;
5
6          void showMsg()
7          {
8              cout << strA;
9          }
10 };
11
12 class subClass : public baseClass // 子類別
13 {
14     public:
15         string strB;
16
17         void sub_showMsg()
18         {
19             cout << strB + "How are you?" << endl;
20         }
21 };

```

接著，主函式 `main()` 中程式碼第 25 行宣告 `subClass` 類別的物件 `myCls`，第 27 行將繼承自 `baseClass` 類別的資料成員 `strA` 設定為 "Hello, "；雖然在物件 `subCls` 中並沒有資料成員 `strA`，但因為繼承了 `baseClass` 類別，所以可以使用資料成員 `strA`。相同的道理，第 28 行呼叫了物件 `subCls` 的成員函式 `showMsg()`，所以第 28 行會顯示："Hello, "。

```

23 int main()
24 {
25     subClass myCls;
26
27     myCls.strA = "Hello, ";

```

```

28     myCls.showMsg();
29     myCls.strB = "Mary. ";
30     myCls.sub_showMsg();
31 }

```

程式碼第 29 行設定物件 `myCls` 的資料成員 `strB` 等於字串 "Mary. "，第 30 行呼叫了物件 `subCls` 的成員函式 `sub_showMsg()`，所以第 30 行會顯示："Mary. How are you?"。因此，第 28、30 行一併輸出顯示結果：

Hello, Mary. How are you?

重載類別的成員

在類別繼承關係中，若子類別中的成員名稱與父類別中的成員名稱相同時，則父類別的這個成員將被遮蔽，無法在子類別中使用；此種情形稱為重載或覆載（Overriding）。重載與多載很類似，在同一個範圍的同名函式稱為多載；若是在類別的繼承關係中，則稱為重載。透過重載的特色，可以修改或是重寫父類別中的成員，以符合子類別的需要。

例如，將類別 `subClass` 的成員函式 `sub_showMsg()` 改寫如下所示：

```

12 class subClass : public baseClass // 子類別
13 {
14     public:
15         string strB;
16
17         void showMsg()
18         {
19             cout << strB + "How are you?" << endl;
20         }
21 };

```

如上述程式碼第 17 行所示，子類別 `subClass` 的成員函式與父類別的成員函式有著相同的函式名稱：`showMsg()`；因此，在 `main()` 主函式中，程式碼第 29 行物件 `myCls` 所呼叫的成員函式 `showMsg()` 就是類別 `subClass` 的成員函式 `showMsg()`；如下所示。因此，第 29 行會顯示："Mary. How are you?"。

```

23 int main()
24 {
25     subClass myCls;
26

```

```

27     myCls.strA = "Hello. ";
28     myCls.strB = "Mary. ";
29     myCls.showMsg(); ← 類別 subClass 的成員函式 showMsg()
30 }

```

即使是這樣的情形，若還是要執行父類別中相同名稱的成員函式，可以使用以下的方式（範例檔案 01-2）。第 1 種方式在子類別中呼叫父類別相同名稱的成員，則需要在成員名稱之前加上 " 父類別名稱 :: "：

父類別名稱 :: 成員名稱

例如：在類別 `subClass` 的成員函式 `showMsg()` 中呼叫父類別中同名稱的成員函式 `showMsg()`，如下程式碼第 19 行所示：

```

12 class subClass : public baseClass // 子類別
13 {
14     public:
15         string strB;
16
17         void showMsg()
18         {
19             baseClass::showMsg();
20             cout << strB + "How are you?" << endl;
21         }
22 };

```

第 2 種方式則是使用強制轉型的方式，讓子類別的物件轉型為父類別之後，再存取父類別的成員。如下程式碼第 29 行所示，因此，第 29 行會顯示："Hello, "。

```

23 int main()
24 {
25     subClass myCls;
26
27     myCls.strA = "Hello. ";
28     myCls.strB = "Mary. ";
29     ((baseClass)myCls).showMsg();
30 }

```

🔗 練習 1：設計計算矩形與長方體之類別

設計用於計算矩形面積之類別 `Rectangle`：資料成員用於表示矩形的長度與寬度，以及用於計算矩形面積的成員函式 `compute()`。再設計可用於計算長方體體積之類別 `Cuboid`，此類別繼承 `Rectangle` 類別，並重載 `compute()` 成員函式，使之用於計算長方體之體積。

■ 解說

`Rectangle` 的類別的架構大致如下所示。在類別裡只有 1 個 `public` 區段，區段內有 2 個 `double` 型別的資料成員 `width` 與 `length`，分別表示矩形的寬與長。成員函式 `compute()` 則用於計算矩形的面積。

```
class Rectangle
{
    public:
        double width, length;  // 寬與長

        double compute(){...}
}
```

`Cuboid` 類別的架構則大致如下所示。此類別繼承 `Rectangle` 類別，在類別裡只有 1 個 `public` 區段，區段內有 1 個 `double` 型別的資料成員 `height`，表示長方體的高；至於長方體的寬與長可以從父類別 `Rectangle` 繼承過來。成員函式 `compute()` 用於計算長方體的體積。因為計算矩形面積與計算長方體體積的公式不同，所以 `Cuboid` 類別重載了成員函式 `compute()`，重新定義了 `compute()` 的內容以符合計算長方體的體積公式。

```
class Cuboid: public Rectangle
{
    public:
        double height;  // 高

        double compute(){...}
}
```

■ 執行結果

```
矩形面積等於：120
長方體體積等於：960
```

程式碼列表

```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle
5  {
6      public:
7          double width = 0, length = 0; // 寬與長
8
9          double compute()
10         {
11             return width * length;
12         }
13 };
14
15 class Cuboid : public Rectangle
16 {
17     public:
18         double height=0; // 高
19
20         double compute()
21         {
22             return Rectangle::compute() * height;
23         }
24 };
25
26 int main()
27 {
28     Cuboid cub;
29
30     cub.length = 10;
31     cub.width = 12;
32     cout << "矩形面積等於：" << ((Rectangle)cub).compute() << endl;
33
34     cub.height = 8;
35     cout << "長方體體積等於：" << cub.compute() << endl;
36
37     system("pause");
38 }
```

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 4-13 行定義類別 `Rectangle`，用於計算矩形的面積。類別裡只有一個 `public` 區段，第 7 行宣告 2 個資料成員 `width` 與 `length`，用於表示矩形的寬與長。第 9-12 行宣告成員函式 `compute()`，用於計算矩形的面積；第 11 行計算並回傳矩形的面積。
3. 程式碼第 15-24 行定義類別 `Cuboid`，此類別繼承 `Rectangle` 類別。類別中只有一個 `public` 區段，第 18 行宣告資料成員 `height`，用於表示長方體之高度；而長方體的寬與長，則繼承自類別 `Rectangle` 的資料成員 `width` 與 `length`。

因為計算長方體體積的公式與計算矩形面積的公式不同；因此，第 20-23 行重載 `compute()` 成員函式，第 22 行計算並回傳長方體的體積：呼叫 `Rectangle()` 類別的 `compute()` 函式計算並取得矩形的面積，接著乘上高度 `height` 便能計算出長方體的體積。

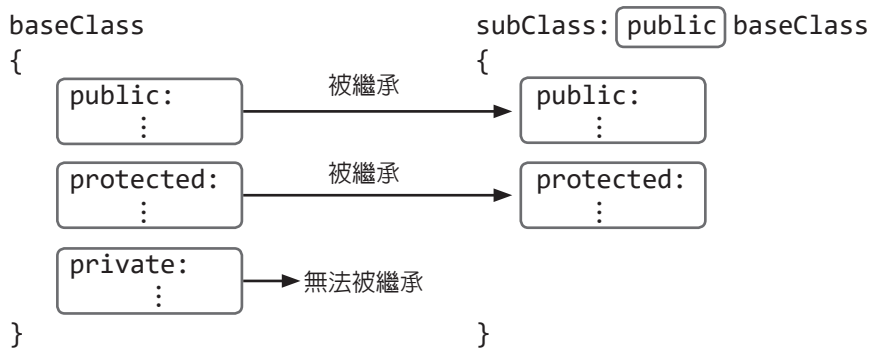
4. 開始於 `main()` 主函式中撰寫程式。程式碼第 28 行宣告 `Cuboid` 類別的物件變數 `cub`，第 30-31 行設定物件 `cub` 的資料成員 `width` 與 `length`。第 32 行將物件 `cub` 轉型為 `Rectangle` 類別，並呼叫 `Rectangle` 類別的成員函式 `compute()` 便可計算矩形的面積。
5. 程式碼第 34 行設定物件 `cub` 的資料成員 `height`；在步驟 4 宣告的物件 `cub` 已經設定了長方體的寬與長，因此第 35 行可以呼叫成員函式 `compute()` 計算並取得長方體的體積。

不同繼承權限的效果

在定義類別繼承關係時，父類別中的成員會依據子類別繼承時所使用的不同的繼承權限修飾字，決定是否可以被繼承到子類別，以及被繼承到子類別的何種存取控制的區段中。若有 2 個類別 `baseClass` 與 `subClass`，並且 `subClass` 繼承 `baseClass` 類別，則各種繼承權限的特點如下說明。

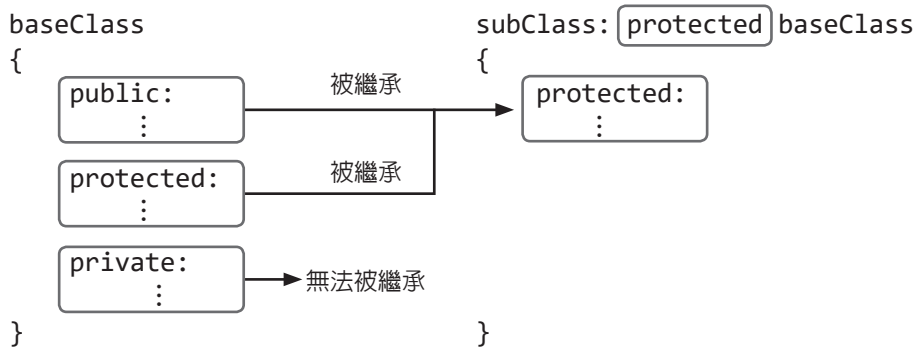
► `public` 繼承權限

在父類別中除了 `private` 區段內的成員無法被繼承之外，`public` 與 `protected` 區段的成員都可以被繼承到子類別各自的區段；如下圖所示。因此，若希望類別成員可以一直被繼承下去，就要使用 `public` 繼承權限。



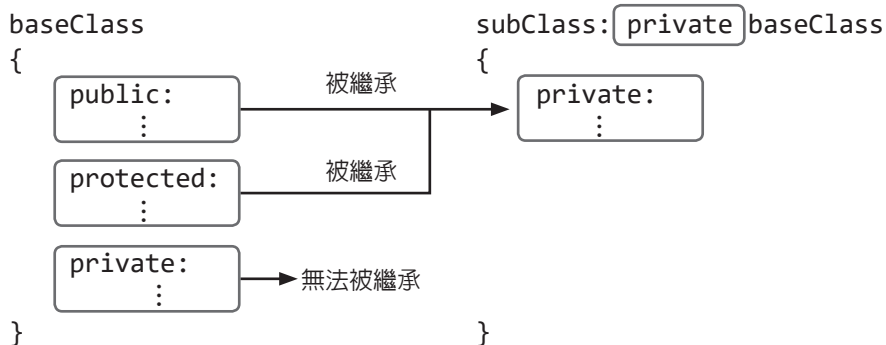
► protected 繼承權限

在父類別中除了 `private` 區段內的成員無法被繼承之外，`public` 與 `protected` 區段的成員都可以被繼承到子類別的 `protected` 區段；如下圖所示。因此，若只想被繼承之後的成員也只能在類別中使用，或是再次繼承的類別中使用，便可以使用 `protected` 繼承權限。



► private 繼承權限

在父類別中除了 `private` 區段內的成員無法被繼承之外，`public` 與 `protected` 區段的成員都可以被繼承到子類別的 `private` 區段；如下圖所示。因此，若不想被繼承之後的成員再次被繼承，就可以使用 `private` 繼承權限。



29-2 建構元、複製建構元、解構元

在類別的繼承關係中，建構元、複製建構元與解構元也會自動被執行。然而，有了類別的繼承關係之後，執行的方式與時機會有所不同，以及需要特別注意的地方。

父、子類別建構元的執行順序

建立子類別的物件時不僅會執行自己的建構元，也會自動執行父類別的建構元；如下範例所示（範例檔案 02-1）。程式碼第 1-11 行為父類別 `baseClass`，在 `public` 區段內有 1 個整數型別的資料成員 `varA`，以及第 6-10 行是沒有帶參數的建構元；第 8 行將資料成員 `varA` 的初始值設定為 10。第 9 行顯示訊息，表示執行的是父類別的建構元。

第 13-20 行為子類別 `subClass`，並以 `public` 繼承權限繼承 `baseClass` 類別；因此，也會繼承父類別的資料成員 `varA`。在 `public` 區段內第 16-19 行是沒有帶參數的建構元，第 18 行顯示訊息，表示執行的是子類別的建構元。

```

1  class baseClass
2  {
3      public:
4          int varA;
5
6          baseClass()
7          {
8              varA = 10;
9              cout << "執行父類別 baseClass 建構元 " << endl;
10         }
11 };
12
13 class subClass : public baseClass
14 {
15     public :
16         subClass()
17         {
18             cout << "執行子類別 subClass 建構元 " << endl;
19         }
20 };
21
22 int main()
23 {
24     subClass clsA;
25
26     cout << "繼承自父類別的資料成員 varA=" << clsA.varA << endl;
27 }

```

在主函式 `main()` 裡，程式碼第 24 行宣告 `subClass` 類別的物件 `clsA`。此時會先去執行父類別 `baseClass` 的建構元，然後再執行 `subClass` 的建構元。因此，會依序顯示以下的輸出結果：

```
執行父類別 baseClass 建構元
執行子類別 subClass 建構元
```

先執行父類別的建構元是合理的；因為既然是繼承關係，所以子類別中的部分成員是繼承於父類別，所以必須先執行父類別的建構元來初始化這些成員。因此，程式碼第 26 行顯示繼承自父類別的資料成員 `varA`，才會顯示其值等於 `10`；如下所示。

```
繼承自父類別的資料成員 varA=10
```

呼叫父類別特定的建構元

將上述的範例修改如下所示（範例檔案 02-2）。程式碼第 1-17 行定義父類別 `baseClass`，在 `private` 區段內有 1 個字串型別的資料成員 `name`，預設值等於父類別的名稱 "`baseClass`"。在 `public` 區段內有 2 個建構元：第 7-10 行與第 12-16 行。第 1 個建構元沒有接收任何參數，並顯示由 `name` 執行了父類別建構元的訊息。第 2 個建構元接收 1 個字串型別的參數 `str`，並將參數 `str` 設定給資料成員 `name`，並顯示由 `name` 執行了父類別建構元的訊息。

程式碼第 19-26 行定義子類別 `subClass`，在 `public` 區段內只有 1 個建構元，此建構元接收 1 個字串型別的參數 `str`；第 24 行顯示由 `str` 執行了子類別 `subClass` 建構元的訊息。

```

1  class baseClass
2  {
3      private:
4          string name = "baseClass";
5
6      public:
7          baseClass()
8          {
9              cout << name << " 執行父類別 baseClass 建構元 " << endl;
10         }
11
12         baseClass(string str)
13         {
14             name = str;
15             cout << name << " 執行父類別 baseClass 建構元 " << endl;
16         }
17     };
18
19  class subClass : public baseClass

```

```

20 {
21     public:
22         subClass(string str)
23         {
24             cout << str << " 執行子類別 subClass() 建構元 " << endl;
25         }
26 };
27
28 int main()
29 {
30     subClass subObj("subObj");
31 }

```

程式碼第 28-31 行為主函式 `main()`，第 30 行宣告 `subClass` 類別的物件 `subObj`，並將物件的名稱 `"subObj"` 作為引數。執行後顯示如下之結果：

```

baseClass 執行父類別 baseClass 建構元
subObj 執行子類別 subClass 建構元

```

因為子類別所宣告的物件會自動執行父類別的建構元，雖然第 30 行物件 `subObj` 在宣告時帶有 1 個字串型別的引數；然而，並沒有去執行父類別正確的建構元：第 12-16 行帶有 1 個字串型別參數的建構元。

由此可知，父類別會被自動執行的都是預設的（在沒有提供自訂建構元的情形下）或是沒有帶參數的建構元。若要執行父類別特定的建構元，便要自行在子類別的建構元中，自行指定要執行父類別中的哪個建構元；如下所示（範例檔案 02-3）：

```

19 class subClass : public baseClass
20 {
21     public:
22         subClass(string str) : baseClass(str)
23         {
24             cout << str << " 執行子類別 subClass() 建構元 " << endl;
25         }
26 };

```

如上述程式碼第 22 行所示，在宣告子類別的建構元時，加上 `": baseClass(str)"` 表示指定呼叫帶有 1 個字串型別參數的父類別建構元，並將參數 `str` 作為引數傳遞給父類別的建構元。因此，更改後的範例輸出如下的結果：

```

subObj 執行父類別 baseClass 建構元
subObj 執行子類別 subClass 建構元

```

由輸出結果可以清楚地知道子類別物件 `subObj` 呼叫了父類別正確的建構元之後，再執行自己的建構元。

複製建構元與解構元

子類別所宣告的物件會自動執行父類別的複製建構元與解構元，並且子類別的解構元先執行之後，才會執行父類別的解構元；如下範例所示（範例檔案 02-4）。

程式碼第 1-24 行定義父類別 `baseClass`；第 3-4 行 `private` 區段內只有 1 個整數型別的指標變數 `numbers`，初始值等於 `NULL`。第 6-23 行為 `public` 區段，第 7-11 行與第 13-17 行分別為建構元與拷貝建構元；此 2 個建構元都配置記憶體空間給資料成員 `numbers`。第 19-23 行為解構元，用於釋放資料成員 `numbers` 所佔用的記憶體空間。

程式碼第 26-33 行定義子類別 `subClass`，以 `public` 繼承權限繼承 `baseClass` 類別。`public` 區段內為建構元，只顯示呼叫 `subClass` 建構元的訊息。

```

1  class baseClass
2  {
3      private:
4          int *numbers=NULL;
5
6      public:
7          baseClass()
8          {
9              cout << "baseClass 建構元被呼叫了 " << endl;
10             numbers = new int[3];
11         }
12
13         baseClass(const baseClass &obj)
14         {
15             cout << "baseClass 複製建構元被呼叫了 " << endl;
16             numbers = new int[3];
17         }
18
19         ~baseClass()
20         {
21             cout << "baseClass 解建構元被呼叫了 " << endl;
22             delete[] numbers;
23         }
24 };
25
26 class subClass : public baseClass

```

```

27 {
28     public:
29         subClass()
30         {
31             cout << " 執行 subClass 建構元 " << endl;
32         }
33 };
34
35 int main()
36 {
37     subClass clsA;
38     subClass clsB(clsA);
39 }

```

程式碼第 35-39 行為主函式 `main()`，第 37 行宣告 `subClass` 類別的物件 `clsA`，因此會先執行 `baseClass` 類別的建構元（第 7-11 行），接著再執行 `subClass` 類別的建構元。因此，會顯示如下的結果：

```
baseClass 建構元被呼叫了
執行 subClass 建構元
```

第 38 行宣告 `subClass` 類別的物件 `clsB`，並傳入物件 `clsA` 作為引數；因此會自動呼叫 `baseClass` 類別的複製建構元（第 13-17 行）。因此，此範例會顯示如下的結果：

```
baseClass 複製建構元被呼叫了
```

第 38 行執行了父類別的複製建構元之後，為何沒有呼叫子類別 `subClass` 的建構元？其實是執行了 Visual Studio C++ 所提供的預設複製建構元。因此，若 `subClass` 類別有提供自訂的複製建構元，則第 38 行便會執行此自訂的複製建構元。所以，將子類別 `subClass` 的定義修改為如下所示（範例檔案 02-5）：第 34-37 行增加類別 `subClass` 的複製建構元。

```

26 class subClass : public baseClass
27 {
28     public:
29         subClass()
30         {
31             cout << " 執行 subClass 建構元 " << endl;
32         }
33
34         subClass(const subClass& obj)
35         {

```

```

36         cout << " 執行 subClass 的複製建構元 " << endl;
37     }
38 };
39
40 int main()
41 {
42     subClass clsA;
43     subClass clsB(clsA);
44 }

```

則第 43 行執行結果如下所示：

baseClass 建構元被呼叫了
執行 subClass 的複製建構元

這次的執行結果的確執行了子類別 `subClass` 的複製建構元，但卻沒有執行父類別的複製建構元，反而執行了父類別的一般建構元而已。若在父類別的複製建構元中有動態配置記憶體的操作時，會因為沒有執行到此複製建構元而發生錯誤。

因此，可由子類別的複製建構元呼叫父類別的複製建構元；再次修改 `subClass` 類別的複製建構元如下所示（範例檔案 02-6）：增加呼叫父類別 `baseClass(obj)` 的複製建構元。

```

26 class subClass : public baseClass
27 {
28     public:
29         subClass()
30         {
31             cout << " 執行 subClass 建構元 " << endl;
32         }
33
34         subClass(const subClass& obj) : baseClass(obj)
35         {
36             cout << " 執行 subClass 的複製建構元 " << endl;
37         }
38 };
39
40 int main()
41 {
42     subClass clsA;
43     subClass clsB(clsA);
44 }

```

呼叫父類別的
複製建構元。

修改後之後，第 43 行執行之後正確地輸出如下的結果：

```
baseClass 複製建構元被呼叫了
執行 subClass 的複製建構元
```

練習 2：產生亂數並計算其總和與平均

設計 2 個類別的繼承關係。父類別 `baseClass` 用於儲存所產生的亂數，以及記錄亂數的數量。子類別 `subClass` 用於計算父類別裡所產生的亂數的總和與平均。寫一程式輸入欲產生亂數的數量，呼叫自訂函式 `genNumbers()`，並於其中宣告類別 `subClass` 的物件，再將物件作為引數傳遞給另一個自訂函式 `showData()` 計算所產生的亂數的總和與平均。

■ 解說

父類別 `baseClass` 用於產生指定數量的亂數，因此應該會有 2 個資料成員：儲存產生的亂數以及記錄亂數的數量。由於產生亂數的數量是由使用者所輸入，因此無法預先設定儲存亂數的資料成員的長度，必須使用動態記憶體配置；所以，需要在建構元、複製建構元與解構元裡處理配置記憶體與釋放記憶體。依照題意需將物件作為引數，並傳遞給自訂函式去計算亂數的總和與平均；因此，才會需要複製建構元。

子類別只負責計算亂數的總和與平均；因此，除了繼承父類別之外，還需要提供自己的建構子與計算總分與平均的成員函式；其餘所需的資料成員與成員函式就從父類別取得就行了。

■ 執行結果

輸入要產生 8 個介於 1-100 之間的亂數。因為宣告的是子類別的物件，因此會先呼叫父類別的建構元之後，才執行子類別的建構元；如同輸出結果的第 2-3 行所示。接著將物件作為引數傳遞給自訂函式 `showData()`，因此會呼叫複製建構元，如同輸出結果第 4 行所示。

```
輸入要產生幾個亂數：8
執行 baseClass 建構元
執行 subClass 建構元
執行 baseClass 複製建構元
72 71 19 39 88 58 5 69
總和 = 421
平均 = 52.625
```


程式碼列表

```

1  #include <iostream>
2  #include <time.h>
3  using namespace std;
4
5  class baseClass
6  {
7      private:
8          int* numbers=NULL;
9          int num = 0;
10
11      public:
12          baseClass(int num) // 建構元
13          {
14              cout << "執行 baseClass 建構元 " << endl;
15              numbers = new int[num];
16              this->num = num;
17              for (int i = 0; i < num; i++)
18                  numbers[i] = rand() % 100 + 1;
19          }
20
21          baseClass(const baseClass& obj) // 複製建構元
22          {
23              cout << "執行 baseClass 複製建構元 " << endl;
24              num = obj.num;
25              numbers = new int[num];
26              memcpy(numbers, obj.numbers, sizeof(int)*num);
27          }
28
29          void ShowData() // 顯示 numbers 的內容
30          {
31              for (int i = 0; i < num; i++)
32                  cout << numbers[i] << " ";
33
34              cout << endl;
35          }
36
37          int* getNumbers() // 取得 numbers
38          {
39              return numbers;
40          }
41

```

```

42         void getNum(int &num) // 取得亂數的個數
43         {
44             num = this->num;
45         }
46
47         ~baseClass() // 解構元
48         {
49             if (num != 0)
50                 delete[] numbers;
51         }
52     };
53
54     class subClass :public baseClass
55     {
56     public:
57         float sum, avg; // 總和、平均
58
59         subClass(int num):baseClass(num)
60         {
61             cout << " 執行 subClass 建構元 " << endl;
62             sum = 0;
63             avg = 0;
64         }
65
66         void compute()
67         {
68             int* numbers;
69             int num;
70
71             numbers = getNumbers();
72             getNum(num);
73
74             for (int i = 0; i < num; i++)
75                 sum += numbers[i];
76             avg = sum / (float)num;
77         }
78     };
79
80     void showData(subClass cls)
81     {
82         cls.ShowData();
83         cls.compute();

```

```

84     cout << " 總和 = " << cls.sum << endl;
85     cout << " 平均 = " << cls.avg << endl;
86 }
87
88 void genNumbers(int num)
89 {
90     subClass cls(num);
91
92     showData(cls);
93 }
94
95 int main()
96 {
97     int num;
98
99     srand((unsigned)time(NULL));
100
101     cout << " 輸入要產生幾個亂數：";
102     cin >> num;
103     if (num <= 0)
104         cout << " 輸入錯誤 " << endl;
105     else
106         genNumbers(num);
107
108     system("pause");
109 }

```

程式講解

1. 程式碼第 1-3 行引入 `iostream` 標頭檔與宣告使用 `std` 命名空間。
2. 程式碼第 5-52 行為父類別 `baseClass` 的定義。第 7-9 行在 `private` 區段內有 2 個資料成員：整數指標型別的變數 `numbers` 用於儲存所產生的亂數，整數變數 `num` 用於記錄產生的亂數數量。
3. 第 11-51 行為 `public` 區段。第 12-19 行為建構元，並接收一個整數型別的參數 `num`，此參數表示欲產生的亂數數量。第 15 行配置 `num` 個整數型別的記憶體空間給資料成員 `numbers`，第 16 行將參數 `num` 設定給類別的資料成員 `num`。第 17-18 行使用 `for` 重複敘述產生 `num` 個介於 1-100 的亂數，並儲存於資料成員 `numbers` 中。

第 21-27 行為複製建構元，與建構元差別的地方在於第 26 行的程式敘述，使用 `memcpy()` 函式將所傳入的物件 `obj` 的資料成員 `numbers` 拷貝給資料成員 `numbers`。

4. 第 29-35 行為成員函式 `showData()`，用於顯示儲存於資料成員 `numbers` 裡的亂數。第 37-40 行為成員函式 `getNumbers()`，函式回傳值型別為 `int *`，用於回傳資料成員 `numbers`。第 42-45 行為成員函式 `getNum()`，此函式使用參考呼叫的方式回傳資料成員 `num`。第 47-51 行為解構元，用於釋放資料成員 `numbers` 所佔用的記憶體空間。
5. 程式碼第 54-78 行為子類別 `subClass` 的定義，並使用 `public` 繼承權限繼承 `baseClass` 類別。在 `public` 區段內第 57 行宣告浮點數型別的資料成員 `sum` 與 `avg`，分別代表總和與平均。第 59-64 行為建構元，並且呼叫了父類別的建構元 `baseClass(num)`；整數型別的參數 `num` 用於指定產生的亂數數量。
第 66-77 行為成員函式 `compute()`，用於計算總和與平均。第 68-69 行宣告整數指標 `numbers` 以及整數變數 `num`，分別用於接收第 71-72 行呼叫父類別的成員函式 `getNumbers()` 與 `getNum()` 的回傳值。第 74-75 行使用 `for` 重複敘述計算總和，並儲存於資料成員 `sum`。第 76 行計算平均，並儲存於資料成員 `avg`。
6. 程式碼第 80-86 行為自訂函式 `showData()` 的程式本體，用於顯示所產生的亂數以及計算總和與平均；並接收 1 個 `subClass` 類別的參數 `cls`。第 82-83 行呼叫參數 `cls` 的成員函式 `showData()` 與 `compute()`，分別用於顯示所產生的亂數，以及計算這些亂數的總和與平均。第 84-85 行顯示總和與平均。
7. 程式碼第 88-93 行為自訂函式 `genNumbers()`，用於產生亂數以及計算總和與平均；並接收 1 個整數參數 `num`，用於表示要產生的亂數數量。第 90 行宣告 `subClass` 類別的物件 `cls`，並傳入變數 `num` 作為引數。第 92 行呼叫自訂函式 `showData()` 來顯示所產生的亂數，以及計算這些亂數的總和與平均。
8. 開始撰寫主函式 `main()`。程式碼第 97 行宣告整數變數 `num`，用於表示亂數的數量。第 99 行使用 `srand()` 函式初使化亂數產生器。第 101-102 行顯示輸入的提示，以及將輸入的資料儲存於變數 `num`。第 103-106 為 `if...else` 判斷敘述，若輸入的亂數數量小於 0，則顯示錯誤；否則，呼叫自訂函式 `genNumbers()` 來產生亂數並計算總和與平均。

29-3 多重繼承

C++ 所提供的類別繼承功能，允許子類別可以同時繼承多個父類別，稱之為多重繼承（Multiple inheritance）。透過多重繼承的特性，可以將不同功能的類別組合在一起，形成一個符合所需的新類別。

多承繼承的語法與關係

多重繼承的語法如下所示：

```
class 子類別的名稱 : 繼承權限修飾字 父類別 1, 繼承權限修飾字 父類別 2, ...
{

};
```

例如下面的例子，類別 `baseClass1` 與 `baseClass2` 作為父類別，而類別 `subClass` 則以 `public` 繼承權限繼承了 `baseClass1`，以及使用 `protected` 繼承權限繼承了 `baseClass2`；因此，此 2 個父類別的建構元、複製建構元與解構元一樣會依照繼承關係的方式自動被執行。

```
class baseClass1 // 父類別 1
{
    :
};
class baseClass2 // 父類別 2
{
    :
};

class subClass : public baseClass1, protected baseClass2
{
    :
};
```

繼承 2 個父類別
↓

再看一個實際的例子（參考範例檔案 03-1）。程式碼第 1-10 行為計算圓形面積的類別 `Circle`，只有一個 `public` 區段。第 4 行為資料成員 `radius`，用於表示圓的半徑。第 6-9 行為成員函式 `computeCircle()`，用於計算並回傳圓形的面積。

第 12-21 行為計算矩形面積的類別 `Rectangle`，也只有一個 `public` 區段。第 15 行為資料成員 `length` 與 `width`，用於表示矩形的長與寬。第 17-20 行為成員函式 `computeRectangle()`，用於計算並回傳矩形的面積。

```
1 class Circle // 圓形類別
2 {
3     public:
4         double radius = 0; // 半徑
5
6         double computeCircle() // 計算圓形面積
7         {
```

```

8         return (radius * radius)* 3.14159;
9     }
10 };
11
12 class Rectangle // 矩形類別
13 {
14     public:
15         double length=0, width=0; // 長與寬
16
17         double computeRectangle() // 計算矩形面積
18         {
19             return length * width;
20         }
21 };

```

第 23-40 行為計算基礎形狀的面積的類別 `PriShapes`，只有一個 `public` 區段，並以 `public` 繼承權限繼承 `Circle` 與 `Rectangle` 此 2 個類別。第 26 行資料成員 `sel` 用於代表選擇要計算圓形面積或是矩形面積。第 28-39 行為成員函式 `computeSize()` 用於計算圓形或是矩形的面積。第 30-38 行根據資料成員 `sel` 的值，分別呼叫並回傳父類別 `Circle` 的成員函式 `computeCircle()`，或父類別 `Rectangle` 的成員函式 `computeRectangle()`；若不是要計算圓形或矩形的面積，則回傳 0。

```

23 class PriShapes : public Circle, public Rectangle
24 {
25     public:
26         int sel=0; //1: 計算圓形面積 2: 計算矩形面積
27
28         double computeSize()
29         {
30             if (sel == 1)
31                 return computeCircle();
32             else
33             {
34                 if (sel == 2)
35                     return computeRectangle();
36                 else
37                     return 0;
38             }
39         }
40 };

```

在 `main()` 主函式中，程式碼第 44 行宣告子類別 `PriShapes` 的物件 `shapes`，第 46-47 行設定圓形的半徑等於 20.3，並設定物件 `shapes` 的資料成員 `sel` 等於 1，表示要計算的是圓的面積，第 48 行執行物件 `shapes` 的成員函式 `computeSize()` 計算並顯示圓形的面積。因為類別 `PriShapes` 繼承了 `Circle` 類別，因此也能使用它的資料成員 `radius` 以及成員函式 `computeCircle()`。

```

42 int main()
43 {
44     PriShapes shapes;
45
46     shapes.radius = 20.3;
47     shapes.sel = 1;
48     cout << " 圓形面積 = " << shapes.computeSize() << endl;
49
50     shapes.length = 12;
51     shapes.width = 26;
52     shapes.sel = 2;
53     cout << " 矩形面積 = " << shapes.computeSize() << endl;
54 }

```

第 50-52 行設定矩形的長與寬分別等於 12 與 26，並設定物件 `shapes` 的資料成員 `sel` 等於 2，表示要計算矩形的面積，第 53 行執行物件 `shapes` 的成員函式 `computeSize()` 計算並顯示矩形的面積。因為類別 `PriShapes` 也同時繼承了 `Rectangle` 類別，因此也能使用它的資料成員 `length` 與 `width`，以及成員函式 `computeRectangle()`。

```

    圓形面積 = 1294.62
    矩形面積 = 312

```

模稜兩可的語法錯誤

在多重繼承關係中的父類別，若不同父類別中有相同名稱的成員時，因為無法區分在子類別中所使用到的父類別的成員，是來自哪個父類別；此時 Visual Studio C++ 便會發出警告。

例如，將上述例子中的 `Circle` 類別的成員函式 `computeCircle()` 改成 `compute()`，`Rectangle` 類別的成員函式 `computeRectangle()` 也改成 `compute()`，如下所示：

```

1 class Circle
2 {
3     public:
4         double radius = 0;
5

```

```

6      double compute()
7      {
8          return (radius * radius)* 3.14159;
9      }
10 };
11
12 class Rectangle
13 {
14     public:
15         double length=0, width=0;
16
17         double compute()
18         {
19             return length * width;
20         }
21 };

```

名稱相同

則在子類別 `PriShapes` 的成員函式 `computeSize()` 中，呼叫父類別的成員函式 `compute()` 時，便無法分辨是要呼叫的是 `Circle` 類別還是 `Rectangle` 類別的 `compute()`；如以下程式碼第 31 和 35 行所示。

```

23 class PriShapes : public Circle, public Rectangle
24 {
25     public:
26         int sel=0;
27
28         double computeSize()
29         {
30             if (sel == 1)
31                 return compute();
32             else
33             {
34                 if (sel == 2)
35                     return compute();
36                 else
37                     return 0;
38             }
39         }
40 };

```

出現 "模稜兩可" 的語法錯誤：不知道要執行哪個父類別的 `compute()` 成員函式。

在 Visual Studio C++ 的程式碼編輯視窗也會出現如下的錯誤警告：

```
double computeSize()
{
    if (sel == 1)
        return compute();
    else
    {
        if (sel ==
            return compute();
        else
            return 0;
    }
}
```

double Circle::compute()
"priShapes::compute" 模稜兩可

若爲了統一計算形狀的面積的成員函式的名稱，所以子類別 **PriShapes** 又使用重載的方式將成員函式 **computeSize()** 的名稱改成 **compute()**，則會形成反覆呼叫自己的情形，造成了無窮的迴圈；如下程式碼第 28、31 與 35 行所示：

```
23 class PriShapes : public Circle, public Rectangle
24 {
25     public:
26         int sel=0;
27
28         double compute()
29         {
30             if (sel == 1)
31                 return compute();
32             else
33             {
34                 if (sel == 2)
35                     return compute();
36                 else
37                     return 0;
38             }
39         }
40     };
```

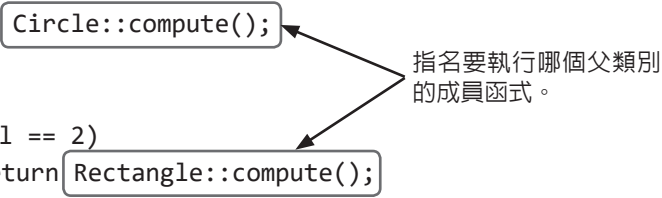
出現反覆執行自己的情形。

解決此種模稜兩可的錯誤，可以直接在父類別的成員函式名稱之前加上父類別的名稱，藉以指名要執行哪個父類別的成員函式（參考範例檔案 03-2）；如下所示：

```

23 class PriShapes : public Circle, public Rectangle
24 {
25     public:
26         int sel=0;
27
28         double compute()
29         {
30             if (sel == 1)
31                 return Circle::compute();
32             else
33             {
34                 if (sel == 2)
35                     return Rectangle::compute();
36                 else
37                     return 0;
38             }
39         }
40 };

```



指名要執行哪個父類別的成員函式。

練習 3：顯示送貨訂單

有 3 個類別：Product、Customer 與 Order，分別表示產品資料、客戶資料與訂單。送貨訂單上需顯示：物品名稱、貨款、購買數量、客戶姓名、送貨地址與聯絡電話。寫一程式，輸入產品資料、客戶資料，並顯示送貨訂單。

解說

送貨訂單上需要顯示的資料即為類別 Product 與 Customer 的資料；因此，可以將此 3 個類別作如下的設計，並將類別 Order 繼承 Product 與 Customer 類別。

► Product 類別

只需要 1 個 public 區段，有 3 個資料成員：物品名稱、貨款與購買數量。有 1 個成員函式 showInfo()，用於顯示 3 個資料成員。Product 類別的架構大致如下所示：

```

class Product
{
    public:
        物品名稱、貨款、購買數量；

        void showInfo()
        {
            顯示產品資料；
        }
};

```

► Customer 類別

只需要 1 個 `public` 區段，有 3 個資料成員：客戶姓名、送貨地址與聯絡電話。有 1 個成員函式 `showInfo()`，用於顯示 3 個資料成員。`Customer` 類別的架構大致如下所示：

```
class Customer
{
    public:
        客戶姓名、送貨地址、連絡電話；

        void showInfo()
        {
            顯示客戶資訊；
        }
};
```

► Order 類別

以 `protected` 繼承權限繼承 `Product` 與 `Customer` 類別，如此一來，2 個父類別的資料成員與成員函式便只能在子類別 `Order` 中存取，藉以達到資料封裝與隱藏。類別 `Order` 只有 1 個 `public` 區段與 3 個成員函式：用於填寫物品資料 `fillProduct()`、填寫客戶資料 `fillCustomer()` 與顯示送貨訂單 `showInfo()`。

此外，成員函式 `fillProduct()` 與 `fillCustomer()` 則需要接收物品與客戶的資料，才能初始化父類別 `Product` 與 `Customer` 中的資料成員。`Order` 的架構大致如下所示：

```
class Customer: protected Product, protected Customer
{
    public:
        void fillProduct( 物品名稱 , 貨款 , 購買數量 )
        {
            設定產品資料；
        }

        void fillCustomer( 客戶姓名 , 送貨地址 , 連絡電話 )
        {
            設定客戶資料；
        }

        void showInfo()
        {
            顯示送貨訂單資訊；
        }
};
```

執行結果

如下所示，分別輸入物品與客戶的資料之後，便顯示完整的送貨訂單。

```
輸入物品名稱、貨款與購買數量：藍芽耳機 2300 2
輸入客戶姓名、送貨住址與連絡電話：王小明 新北市文化路一段 313 號 0922345678
===== 送貨訂單 =====
商品名稱：藍芽耳機 貨款：2300 數量：2
姓名：王小明
送貨地址：新北市文化路一段 313 號
連絡電話：0922345678
```

程式碼列表

```
1 #include <iostream>
2 using namespace std;
3
4 class Product
5 {
6     public:
7         string item="";
8         int price=0;
9         int number = 0;
10
11         void showInfo()
12         {
13             cout << " 商品名稱：" << item <<
14                 " 貨款：" << price << " 數量：" << number << endl;
15         }
16 };
17
18 class Customer
19 {
20     public:
21         string name = "";
22         string address="";
23         string phone = "";
24
25         void showInfo()
26         {
27             cout << " 姓名：" << name << endl;
28             cout << " 送貨地址：" << address << endl;
29             cout << " 連絡電話：" << phone << endl;
30         }
31 }
```

```

31 };
32
33 class Order : protected Product, protected Customer
34 {
35     public:
36         void fillProduct(string item="", int price=0, int num=0)
37         {
38             this->item = item;
39             this->price = price;
40             number = num;
41         }
42
43         void fillCustomer(string name="", string address="", string phone="")
44         {
45             this->name = name;
46             this->address = address;
47             this->phone = phone;
48         }
49
50         void showInfo()
51         {
52             Product::showInfo();
53             Customer::showInfo();
54         }
55 };
56
57 int main()
58 {
59     Order ord;
60     string item, name, address, phone;
61     int price, number;
62
63     cout << " 輸入物品名稱、貨款與購買數量：";
64     cin >> item >> price >> number;
65     ord.fillProduct(item, price, number);
66
67     cout << " 輸入客戶姓名、送貨地址與連絡電話：";
68     cin >> name >> address >> phone;
69     ord.fillCustomer(name, address, phone);
70
71     cout << "===== 送貨訂單 =====" << endl;
72     ord.showInfo();
73
74     system("pause");
75 }

```

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 4-16 行為類別 `Product` 的定義，只有 1 個 `public` 區段，此類別用於表示物品的資料。第 7-9 行為資料成員 `item`、`price` 與 `number`，分別表示物品的名稱、貨款與購買數量。第 11-15 行為成員函式 `showInfo()`，用於顯示此 3 個資料成員。
3. 程式碼第 18-31 行為類別 `Customer` 的定義，只有 1 個 `public` 區段，此類別用於表示客戶的資料。第 21-23 行為資料成員 `name`、`address` 與 `phone`，分別表示客戶的姓名、送貨地址與聯絡電話。第 25-30 行為成員函式 `showInfo()`，用於顯示此 3 個資料成員。
4. 程式碼第 33-55 行為類別 `Order`，以 `protected` 繼承權限繼承 `Product` 與 `Customer` 類別，用於表示送貨的訂單。類別只有 1 個 `public` 區段，第 36-41 行為成員函式 `fillProduct()`，用於填寫物品資料。函式接收 3 個參數 `item`、`price` 與 `num`，分別表示物品名稱、貨款與購買數量。第 38-40 行將此 3 個參數設定給類別相對應的資料成員。

第 43-48 行為成員函式 `fillCustomer()`，用於填寫客戶資料。函式接收 3 個參數 `name`、`address` 與 `phone`，分別表示客戶姓名、送貨地址與聯絡電話。第 45-47 行將此 3 個參數設定給類別相對應的資料成員。

第 50-54 行為成員函式 `showInfo()`，用於顯示送貨訂單。函式分別呼叫父類別 `Product` 與 `Customer` 的成員函式 `showInfo()` 顯示各自的資料。

5. 開始撰寫主函式 `main()`。程式碼第 59 行宣告 `Order` 類別的物件 `ord`，第 60 行宣告字串變數 `item`、`name`、`address` 與 `phone`，分別代表物品名稱、客戶姓名、送貨地址與聯絡電話。第 61 行宣告整數變數 `price` 與 `number`，代表貨款與購買數量。

第 63 行顯示輸入物品資料的提示訊息，第 64 行取得輸入的物品名稱、貨款與購買數量，並儲存於變數 `item`、`price` 與 `number`。第 65 行執行物件 `ord` 的成員函式 `fillProduct()` 並將變數 `item`、`price` 與 `number` 作為引數。

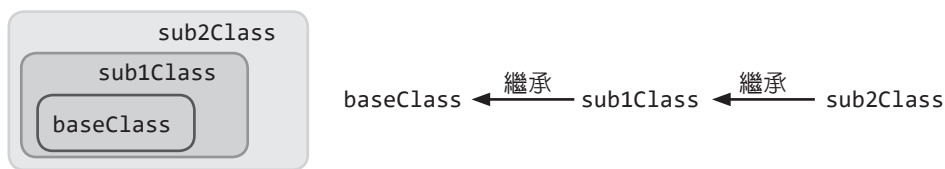
第 67 行顯示輸入客戶資料的提示訊息，第 68 行取得輸入的客戶姓名、送貨地址與聯絡電話，並儲存於變數 `name`、`address` 與 `phone`。第 69 行執行物件 `ord` 的成員函式 `fillCustomer()` 並將變數 `name`、`address` 與 `phone` 作為引數。

第 72 行執行物件 `ord` 的成員函式 `showInfo()` 顯示送貨的訂單資料。

父類別與子類別之轉型

子類別繼承自父類別，因此由子類別所宣告的物件，可以直接指定給父類別之物件，而不需要經過轉型。因此，若以父類別來宣告自訂函式的參數時，便可以接收任何繼承此父類別的子類別物件；並且只要經過適當的子類別轉型，便可以存取該子類別的成員。例如以下範例（參考範例檔案 03-3）：

有 3 個類別 `baseClass`、`sub1Class` 與 `sub2Class`，類別 `sub1Class` 繼承 `baseClass` 類別，而類別 `sub2Class` 又繼承 `sub1Class`；其繼承關係如下圖所示：



此 3 個類別的定義分別為程式碼第 1-9 行、第 11-19 行第 21-29 行。每個類別內都有 1 個字串型別的資料成員 `str`，以及 1 個成員函式 `show()`。

```

1  class baseClass
2  {
3      public:
4          string str = "base";
5          void show()
6          {
7              cout << "From baseClass" << endl;
8          }
9  };
10
11 class sub1Class :public baseClass
12 {
13     public:
14         string str = "sub1";
15         void show()
16         {
17             cout << "From sub1 Class" << endl;
18         }
19 };
20
21 class sub2Class :public sub1Class
22 {

```

```

23     public:
24         string str = "sub2";
25         void show()
26         {
27             cout << "From sub2 Class" << endl;
28         }
29     };

```

程式碼第 31-50 行為自訂函式 `show()` 的程式本體，接收 2 個參數。第 1 個參數為 `baseClass` 類別的參考型別的物件 `obj`，第 2 個參數為整數型別 `type`，用來表示第 1 個參數是哪種類別的物件。第 33-49 行為 `switch...case` 敘述，第 35-38 行為 `type` 等於 0 的程式區塊，表示參數 `obj` 為 `baseClass` 型別，第 36-37 行分別顯示資料成員 `str` 與執行成員函式 `show()`。

第 40-43 行為 `type` 等於 1 的程式區塊，表示參數 `obj` 為 `sub1Class` 型別；因此，可以透過轉型 `"(sub1Class &)"` 將物件 `obj` 轉型為原來的 `subClass` 型別。第 41-42 行分別顯示資料成員 `str` 與執行成員函式 `show()`。

第 45-48 行為 `type` 等於 2 的程式區塊，表示參數 `obj` 為 `sub2Class` 型別；使用相同的方式可以透過轉型 `"(sub2Class &)"` 將物件 `obj` 轉型為原來的 `sub2Class` 型別。第 46-47 行分別顯示資料成員 `str` 與執行成員函式 `show()`。

```

31 void show(baseClass &obj, int type)
32 {
33     switch (type)
34     {
35         case 0: //baseClass
36             cout << "str= " << obj.str << ", ";
37             obj.show();
38             break;
39
40         case 1: //sub1Class
41             cout << "str= " << ((sub1Class &)obj).str << ", ";
42             ((sub1Class&)obj).show();
43             break;
44
45         case 2: //sub2Class
46             cout << "str= " << ((sub2Class&)obj).str << ", ";
47             ((sub2Class&)obj).show();
48             break;
49     }

```



```

50 }
51
52 int main()
53 {
54     baseClass baseCls;
55     sub1Class sub1Cls;
56     sub2Class sub2Cls;
57
58     show(baseCls, 0);
59     show(sub1Cls, 1);
60     show(sub2Cls, 2);
61 }

```

在主函式 `main()` 中，程式碼第 54-56 分別宣告類別 `baseClass`、`sub1Class` 與 `sub2Class` 的物件變數 `baseCls`、`sub1Cls` 與 `sub2Cls`。第 58-60 行呼叫自訂函式 `show()`，並分別將此 3 個物件變數作為引數傳入自訂函式 `show()`。雖然自訂函式 `show()` 所接收的是 `baseClass` 型別的參數，因為類別 `sub1Class` 與 `sub2Class` 都繼承自 `baseClass` 類別，因此並不需要轉型為 `baseClass` 型別，就能被自訂函式 `show()` 所接受。此程式的顯示結果如下所示：

```

str= base, From baseClass
str= sub1, From sub1 Class
str= sub2, From sub2 Class

```

29-4 虛擬函式與抽象類別

虛擬函式或虛擬函數（Virtual function），是類別中特定的成員函式，若某函式在被繼承之後有可能被重載，或是特別設計給繼承的子類別中去重新定義此函式的內容，則此函式適合宣告為虛擬函式。定義虛擬函式只需要在函式的最前面加上 `virtual` 關鍵字即可，例如：

```
virtual void func(){...} // 將成員函式 func() 宣告為虛擬函式
```

在子類別中重載或是重新定義此虛擬函式 `func()` 的成員函式，並不需要在函式前面加上 `virtual` 關鍵字；但通常為了能夠一目了然成員函式之間的繼承關係，所以習慣上也會加上 `virtual` 關鍵字。虛擬函式基本上有以下 3 種用途：簡化父 - 子類別之間的轉型、呼叫正確的成員函式與純虛擬函式。

簡化父、子類別之間的轉型

在 29-3 節中「父類別與子類別之轉型」的範例中，自訂函式 `show()` 接收父類別 `baseClass` 型別的參數 `obj`，並可以透過轉型成為不同的子類別 `sub1Class` 與 `sub2Class` 的物件，然後再存取子類別的成員，如程式碼第 40-48 行所示。透過虛擬函式的方式，便可以省略如此麻煩的型別轉換；修改此範例後的程式碼如下所示（參考範例檔案 04-1）。

類別 `sub1Class` 繼承 `baseClass` 類別，然後類別 `sub2Class` 再繼承 `sub1Class` 類別。2 個子類別中的成員函式 `show()` 都加上了關鍵字 `virtual`，表示重載或重新定義了父類別的 `show()` 函式。自訂函式 `show()` 與主函式 `main()` 如下所示。

```

1  class baseClass
2  {
3      public:
4          string str = "base";
5          virtual void show() ← 虛擬成員函式
6          {
7              cout << str << ", from baseClass" << endl;
8          }
9  };
10
11 class sub1Class :public baseClass
12 {
13     public:
14         string str = "sub1";
15         virtual void show() ← 虛擬成員函式
16         {
17             cout << str << ", from sub1 Class" << endl;
18         }
19 };
20
21 class sub2Class :public sub1Class
22 {
23     public:
24         string str = "sub2";
25         virtual void show() ← 虛擬成員函式
26         {
27             cout << str << ", from sub2 Class" << endl;
28         }
29 };

```

第 31 行自訂函式 `show()` 的第 1 個參數雖然接收 `baseClass` 類別的參考物件 `obj`，但第 33 行卻能依照原本傳入自訂函式中的引數型別（第 42-44 行）正確呼叫不同類別的 `show()` 成員函式；這正是因為將父類別 `baseClass` 的成員函式 `show()` 宣告為虛擬函式的作用。

```

31 void show(baseClass &obj, int type)
32 {
33     obj.show();
34 }
35
36 int main()
37 {
38     baseClass baseCls;
39     sub1Class sub1Cls;
40     sub2Class sub2Cls;
41
42     show(baseCls, 0);
43     show(sub1Cls, 1);
44     show(sub2Cls, 2);
45
46     system("pause");
47 }

```

主函式 `main()` 中，程式碼第 38-40 行分別宣告類別 `baseClass`、`sub1Class` 與 `sub2Class` 的物件 `baseCls`、`sub1Cls` 與 `sub2Cls`。第 42-44 行呼叫自訂函式 `show()`，並且將這 3 個不同類別的物件作為引數。

在自訂函式 `show()` 的第 1 個參數接收的是 `baseClass` 類別的物件參數，所以物件 `sub1Cls` 與 `sub2Cls` 傳入自訂函式 `show()` 後都被轉型為 `baseClass` 型別的物件，所以第 33 行應該執行的是類別 `baseClass` 的成員函式 `show()`。然而，因為成員函式 `show()` 被宣告為虛擬函式，所以第 33 行的參數 `obj` 仍然可以正確地連結到原本引數所屬類別的成員函式 `show()`；此種特色又稱為動態連結。因此，此範例輸出的結果如下所示：

```

base, from baseClass
sub1, from sub1 Class
sub2, from sub2 Class

```

呼叫正確的成員函式

下面的例子示範了子類別呼叫繼承自父類別的成員函式，所發生的混淆情形；若改使用虛擬函式便能避免如此的情況（參考範例檔案 04-2）。

程式碼第 1-12 行與第 14-21 行分別為類別 `baseClass` 與 `subClass`，並且類別 `subClass` 繼承 `baseClass` 類別。在父類別 `baseClass` 中有 2 個成員函式 `showAll()` 與 `show()`；函式 `showAll()` 中呼叫函式 `show()` 來顯示字串 "From baseClass"。子類別 `subClass` 中只有一個重載的成員函式 `show()`，用於顯示字串 "From subClass"。因為類別 `subClass` 繼承 `baseClass` 類別，因此也繼承了父類別的成員函式 `showAll()`。

```

1  class baseClass
2  {
3      public:
4          void showAll()
5          {
6              show();
7          }
8          void show()
9          {
10             cout << "from baseClass";
11         }
12 };
13
14 class subClass: public baseClass // 繼承 baseClass
15 {
16     public:
17         void show()
18         {
19             cout << "from sub1Class";
20         }
21 };
22
23 int main()
24 {
25     subClass subCls;
26
27     subCls.showAll();
28
29     system("pause");
30 }

```

在主函式 `main()` 中，程式碼第 25 行宣告 `subClass` 類別的物件 `subCls`，第 27 行呼叫其成員函式 `showAll()`。因為成員函式 `showAll()` 繼承自類別 `baseClass`，因此在函式 `showAll()` 中呼叫的 `show()` 函式是類別 `baseClass` 中的成員函式 `show()`，這是早在編譯階段時就已經決定了，稱之為靜態連結，因此無法更改。所以，即使是由 `subClass` 類別的

物件 `subCls` 來呼叫 `showAll()` 函式，也無法讓函式 `showAll()` 執行類別 `subClass` 的成員函式 `show()`。因此，此範例的輸出結果為：

From baseClass

若是要讓物件 `subCls` 呼叫成員函式 `showAll()` 時，所執行的 `show()` 函式是類別 `subClass` 的成員函式 `show()`，則只要將類別 `baseClass` 中的成員函式 `show()` 宣告為虛擬函式，便可以修正此種情形；如下所示（參考範例檔案 04-3）：

```

1  class baseClass
2  {
3      public:
4          void showAll()
5          {
6              show();
7          }
8          virtual void show() ← 宣告為虛擬函式
9          {
10             cout << "from baseClass";
11         }
12 };
13
14 class subClass: public baseClass // 繼承 baseClass
15 {
16     public:
17         void show() ← 可以或不需要宣告為虛擬函式
18         {
19             cout << "from sub1Class";
20         }
21 };

```

因此，在程式碼第 27 行執行物件 `subCls` 的成員函式 `showAll()` 時，便會知道 `showAll()` 函式中要執行的函式 `show()` 是類別 `subClass()` 的成員函式 `show()`。此種方式又稱為動態連結：等到要真正執行函式 `show()` 的時候才去判斷要執行的是哪個類別的成員函式 `show()`。因此，經過修改程式之後的執行結果為：

From subClass

純虛擬函式與抽象類別

在設計類別中的成員函式，若只是爲了讓繼承的子類別對父類別中的特定函式重載或是重新定義；此時只需要把此成員函式定義爲虛擬函式，並不需要真正去撰寫此成員函式的內容；此種虛擬函式稱爲純虛擬函數（Pure virtual function），而包含純虛擬函式的類別便稱爲抽象類別（Abstract class）。

因此，抽象類別與純虛擬函式如同規定了類別中該要有哪些成員、要怎麼做的樣板，讓子類別有一定的規範去設計純虛擬函式的內容，以及讓繼承抽象類別的子類別都有一定要提供的成員。純虛擬函式的語法爲：

```
virtual 函式回傳值型別 函式名稱() = 0;
```

例如，宣告類別中的成員函式 `func()` 爲純虛擬函式：

```
virtual int func() = 0;
```

使用純虛擬函式與抽象類別需注意的是：

1. 抽象類別不能用來直接宣告物件變數，因爲它的純虛擬函式並沒有真正的程式內容。
2. 子類別中，若未對父類別中的純虛擬函式撰寫實際的程式內容，則必須在子類別中繼續宣告此函式爲純虛擬函式。
3. 抽象類別中也能宣告一般的虛擬函式，此虛擬函式也能被執行。

例如下列的例子（參考範例檔案 04-4）：程式碼第 1-9 行爲父類別 **Vehicle**，作爲衍生各種車輛的抽象類別；因此，在第 4-6 行提供了車輛的基本判斷條件的資料成員：輪子數量 `wheels`、乘坐人數 `number` 與是否有引擎 `engine`。並且第 8 行提供了顯示車輛名稱的純虛擬函式 `show()`，讓繼承此類別的子類別去重新定義此函式的內容，用於判斷車輛的種類。

```

1 class Vehicle // 車輛
2 {
3     public:
4         int wheels = 0; // 輪子數量
5         int number = 0; // 乘坐人數
6         bool engine = false; // 是否有引擎
7
8         virtual void show() = 0; ← 純虛擬函式
9 };

```

接下來，定義 2 個繼承 **Vehicle** 類別的子類別 **Bike** 與 **Scooter**，分別用於判斷是否爲腳踏車還是摩托車；如程式碼第 11-21 行與第 23-33 行。因爲此 2 個子類別繼承了父類別

`Vehicle` 中的所有資料成員，因此不用再各別宣告輪子數量、乘坐人數與是否有引擎的資料成員。但是在父類別 `Vehicle` 中的成員函式 `show()` 是純虛擬函式，只是函式的原型但沒有實際的內容；因此第 14-20 行、第 26-32 行則重新定義了虛擬函式 `show()` 的實際內容，分別用於判斷是否為腳踏車與判斷是否為摩托車。

```

11 class Bike :public Vehicle // 腳踏車
12 {
13     public:
14         virtual void show() ← 重新定義虛擬函式 show()
15         {
16             if (wheels == 2 && number == 1 && !engine)
17                 cout << " 這是一輛腳踏車 " << endl;
18             else
19                 cout << " 這不是一輛腳踏車 " << endl;
20         }
21 };
22
23 class Scooter :public Vehicle // 摩托車 ( 速克達 )
24 {
25     public:
26         virtual void show() ← 重新定義虛擬函式 show()
27         {
28             if (wheels == 2 && number == 2 && engine)
29                 cout << " 這是一輛機車 " << endl;
30             else
31                 cout << " 這不是一輛機車 " << endl;
32         }
33 };

```

🔄 練習 4：計算不同形狀之面積

設計一個形狀的抽象類別與純虛擬函式，可用於計算矩形與圓形之面積。

■ 解說

此抽象類別是作為矩形和圓形的父類別；因此，在此抽象類別中可以提供計算矩形與圓形面積的資料成員：長度、寬度與半徑，並讓矩形與圓形的類別繼承這些資料成員。此外，計算矩形與圓形面積的公式不同，所以可以在父類別中宣告計算面積的純虛擬函式，然後在矩形與圓形的類別中，各自重新定義此計算面積的虛擬函式。

執行結果

矩形面積 = 288
圓形面積 = 452.389

程式碼列表

```
1  #include <iostream>
2  using namespace std;
3
4  class Shapes // 父類別
5  {
6      public:
7          double length = 0, width = 0; // 長、寬
8          double raduis = 0; // 半徑
9
10         virtual double getSize() = 0; // 純虛擬函式
11     };
12
13     class Rectangle : public Shapes // 矩形類別
14     {
15         public:
16             virtual double getSize() // 重新定義虛擬函式 getSize()
17             {
18                 return length * width;
19             }
20     };
21
22     class Circle :public Shapes // 圓形類別
23     {
24         public:
25             virtual double getSize() // 重新定義虛擬函式 getSize()
26             {
27                 return (raduis * raduis)* 3.14159;
28             }
29     };
30
31     int main()
32     {
33         Rectangle rect;
34         Circle cir;
35     }
```



```

36     rect.length = 12;
37     rect.width = 24;
38     cout << " 矩形面積 = " << rect.getSize() << endl;
39
40     cir.raduis = 12;
41     cout << " 圓形面積 = " << cir.getSize() << endl;
42
43     system("pause");
44 }

```

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 4-11 行為父類別 `Shapes`，用於作為各種形狀的抽象類別，只有 1 個 `public` 區段。第 7-8 行宣告多數形狀都會使用到的資料成員：長度 `length`、寬度 `width` 與半徑 `radius`。第 10 行宣告純虛擬函式 `getSize()`，作為定義計算形狀的面積，因此函式回傳型別為 `double`。
3. 程式碼第 13-20 行為矩形類別 `Rectangle` 的定義，並繼承類別 `Shapes`，只有 1 個 `public` 區段。因為繼承了 `Shapes` 類別，所以可直接使用繼承自 `Shapes` 類別的資料成員：`length`、`width`。第 16-19 行重新定義虛擬函式 `getSize()`，才能計算矩形的面積。
4. 程式碼第 22-29 行為圓形類別 `Circle` 的定義，並繼承類別 `Shapes`，只有 1 個 `public` 區段。因為繼承了 `Shapes` 類別，所以可直接使用繼承自 `Shapes` 類別的資料成員 `radius`。第 25-28 行重新定義虛擬函式 `getSize()`，才能計算圓形的面積。
5. 開始撰寫主函式 `main()`。程式碼第 33-34 行分別宣告 `Rectangle` 與 `Circle` 類別的物件 `rect` 與 `cir`。第 36-37 行設定物件 `rect` 的長度與寬度，第 38 行呼叫物件 `rect` 的成員函式 `getSize()` 計算並顯示矩形的面積。

第 40 行設定物件 `cir` 的半徑，第 41 行呼叫物件 `cir` 的成員函式 `getSize()` 計算並顯示圓形的面積。

虛擬函式與指標物件

因為類別繼承與虛擬函式的特性，使得在使用指標型別的物件時，容易發生一些模稜兩可的問題。例如，有 2 個類別如下所示（請參考範例檔案 04-5）。

程式碼第 1-13 行為父類別 `baseClass`，在其 `public` 區段內有 1 個成員函式 `show()`，用於顯示訊息 `"baseClass"`。第 9-12 行為解構元，用於顯示執行的是哪個類別的解構元。第 15-27 行為子類別 `subClass`，繼承 `baseClass` 類別。在其 `public` 區段內一樣有 1 個成員函

式 `show()`，以及解構元。

```

1  class baseClass
2  {
3      public:
4          void show()
5          {
6              cout << "baseClass." << endl;
7          }
8
9          ~baseClass()
10         {
11             cout << "呼叫 baseClass 解構元 " << endl;
12         }
13 };
14
15 class subClass :public baseClass
16 {
17     public:
18         void show()
19         {
20             cout << "subClass." << endl;
21         }
22
23         ~subClass()
24         {
25             cout << "呼叫 subClass 解構元 " << endl;
26         }
27 };

```

接著在 `main()` 主函式中，第 31-32 行宣告類別 `baseClass` 與 `subClass` 的指標物件 `baseCls` 與 `subCls`。

```

31 baseClass *baseCls;
32 subClass* subCls;
33
34 baseCls = new baseClass();
35 baseCls->show();
36 delete baseCls;
37
38 subCls =new subClass();
39 subCls->show();
40 delete subCls;

```

第 34 行使用 `new` 指令配置類別 `baseClass()` 的實體給指標物件 `baseCls`。第 35 行呼叫其成員函式 `show()`，第 36 行使用 `delete` 釋放指標物件 `baseCls`；因此會輸出結果：

```
baseClass
呼叫 baseClass 解構元
```

第 38 行使用 `new` 指令配置類別 `subClass()` 的實體給指標物件 `subCls`。第 39 行呼叫其成員函式 `show()`，第 40 行使用 `delete` 釋放指標物件 `subCls`；因此會輸出如下結果。因為類別 `subClass` 繼承 `baseClass` 類別，所以在釋放指標物件 `subCls` 時，會先執行類別 `subClass` 的解構元之後，再執行 `baseClass` 的解構元。

```
subClass
呼叫 subClass 解構元
呼叫 baseClass 解構元
```

以上的輸出結果如預期之中。現在將程式碼第 34 行改為如下所示，改用 `subClass` 類別初始化指標物件 `baseCls`：使用子類別初始化父類別所宣告的物件。

```
34 baseCls = new subClass();
```

則第 35-36 行的輸出結果會變得如何？是執行父類別 `baseClass` 的成員函式 `show()` 與及解構元，還是應該執行子類別 `subClass` 的成員函式 `show()` 與及解構元？第 35-36 行的輸出結果為：

```
baseClass
呼叫 baseClass 解構元
```

其實無論是執行父類別 `baseClass` 或是子類別 `subClass` 的成員函式和解構元，都是合理的執行結果；端視想輸出何種結果。如果想輸出的是子類別 `subClass` 的成員函式與解構元，則修改父類別 `baseClass` 的成員函式 `show()` 為虛擬函式，解構元修改為虛擬解構元；如下所示（請參考範例檔案 04-6）：

```
1 class baseClass
2 {
3     public:
4         virtual void show()
5         {
6             cout << "baseClass." << endl;
7         }
```

```

8
9     virtual ~baseClass()
10    {
11        cout << " 呼叫 baseClass 解構元 " << endl;
12    }
13 };

```

在 `subClass` 中的成員函式 `show()` 與解構元，無論是否有加上 `virtual` 關鍵字，都會被視為是虛擬的函式與解構元。修改程式碼後的輸出結果如下所示：

```

subClass
呼叫 subClass 解構元
呼叫 baseClass 解構元

```

因為在 `baseClass` 類別中的成員函式 `show()` 是虛擬函式，解構元也是虛擬解構元；因此，第 35-36 行都會以動態連結的方式先去執行子類別 `subClass` 的成員函式 `show()` 與解構元；然後再執行父類別的虛擬解構元。

三、範例程式解說

1. 建立專案，程式碼第 1-2 行引入 `iostream` 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 4-21 行定義 `Person` 類別。第 6-8 行的 `private` 區段內宣告了 2 個資料成員 `name` 與 `age`，分別表示姓名與年齡。第 10-20 行為 `protected` 區段，此區段內定義了成員函式 `setNameAge()` 與 `showInfo()`。第 11-15 行為成員函式 `setNameAge()` 用於設定姓名與年齡，因此接收了 2 個參數：`name` 與 `age`，分別代表姓名與年齡。第 13-14 行將此 2 個參數設定給類別的資料成員 `this->name` 與 `this->age`。第 17-20 行為成員函式 `showInfo()`，用於顯示姓名 `name` 與年齡 `age`。

```

4 class Person // 基本資料：姓名與年齡
5 {
6     private:
7         string name = ""; // 姓名
8         int age = 0; // 年齡
9
10    protected:
11        void setNameAge(string name, int age)

```

```

12     {
13         this->name = name;
14         this->age = age;
15     }
16
17     void showInfo() // 顯示資料
18     {
19         cout << " 姓名：" << name << " 年齡：" << age << endl;
20     }
21 };

```

3. 程式碼第 23-30 行定義類別 **Basic**。在 **public** 區段內第 26-27 行宣告 2 個 **double** 型別的資料成員 **height** 與 **weight**，分別表示身高 (cm) 與體重 (kg)。第 29 行定義純虛擬函式 **getBMI()**，讓繼承此類別的子類別重新定義此函式的內容。

```

23 class Basic // 基本資料：身高與體重
24 {
25     public:
26         double height = 0; // 身高
27         double weight = 0; // 體重
28
29         virtual double getBMI() = 0; // 計算 BMI
30 };

```

4. 程式碼第 32-63 行定義 **Student** 類別，此類別以 **public** 與 **protected** 繼承權限，分別繼承 **Person** 與 **Basic** 類別；此類別只有 1 個 **public** 區段。第 36-37 行定義 2 個資料成員 **classNo** 與 **schNo**，分別代表班級與號碼。

第 39-42 行為成員函式 **setNameAge()**，此函式重載類別 **Person** 的 **setNameAge()** 成員函式；因此，第 41 行呼叫 **Person** 類別的成員函式 **setNameAge()** 來設定姓名與年齡。第 44-48 行為成員函式 **setH_W()**，此函式用於設定身高與體重；因此，接收 2 個參數 **height** 與 **weight**，分別表示身高與體重。第 46-47 行使用此 2 個參數設定資料成員 **this->height** 與 **this->weight**；此 2 個資料成員繼承自 **Basic** 類別。

```

32 class Student :public Person, protected Basic // 學生資料
33 {
34
35     public:
36         string classNo = ""; // 班級
37         int schNo = 0; // 號碼

```

```

38
39         void setNameAge(string name, int age) // 設定姓名與年齡
40     {
41         Person::setNameAge(name, age);
42     }
43
44         void setH_W(double height, double weight) // 設定身高與體重
45     {
46         this->height = height;
47         this->weight = weight;
48     }

```

第 50-54 行重新定義與實作類別 **Basic** 中的純虛擬函式 **getBMI()**，此函式用於計算 BMI 值；第 52-53 行計算並回傳 BMI 值。第 56-62 行重載類別 **Person** 的成員函式 **showInfo()**，第 58 行先呼叫類別 **Person** 的成員函式 **showInfo()** 顯示姓名與年齡。第 59-61 行顯示其他的基本資料；其中第 60 行呼叫 **getBMI()** 計算並取得 BMI 值。

```

50         virtual double getBMI() // 重新定義 getBMI()
51     {
52         double h = height / 100.0;
53         return weight/ (h * h);
54     }
55
56         void showInfo() // 顯示資料
57     {
58         Person::showInfo();
59         cout << " 身高：" << height << "cm 體重：" << weight << "kg ";
60         cout << "BMI：" << getBMI() << endl;
61         cout << "班級：" << classNo << " " << schNo << "號" << endl;
62     }
63 };

```

5. 在主函式 **main()** 中，程式碼第 67 行宣告 **Student** 類別的物件 **Mary**。第 69 行呼叫物件 **Mary** 的成員函式 **setNameAge()**，並傳入字串 " 王瑪莉 " 與數字 19 此 2 個引數設定姓名與年齡。

第 70 行呼叫物件 **Mary** 的成員函式 **setH_W()**，並傳入 162 與 50 此 2 個引數設定身高與體重。第 71-72 行分別設定物件 **Mary** 的班級與號碼。第 74 行呼叫物件 **Mary** 的成員函式 **showInfo()** 顯示所有的基本資料。

```

65 int main()
66 {
67     Student Mary;
68
69     Mary.setNameAge("王瑪莉", 19); // 設定姓名與年齡
70     Mary.setH_W(162, 50);           // 設定身高與體重
71     Mary.classNo = "一年 A 班";    // 設定班級
72     Mary.schNo = 12;                // 設定號碼
73
74     Mary.showInfo();                // 顯示學生基本資料
75
76     system("pause");
77 }

```

重點整理

1. 子類別宣告物件之後，只會自動執行父類別中沒有帶任何參數的建構元，或是 Visual Studio C++ 的預設建構元；此外的父類別建構元都要自行指定執行。
2. 類別裡的資料成員若需要動態記憶體配置、建構元裡需要處理動態記憶體配置時，則類別需要提供複製建構元來處理相同的事情。
3. 自動呼叫複製建構元的時機會發生在：以類別物件作為另一個物件宣告時的引數、以傳值呼叫的方式將物件作為引數傳遞給其他函式。
4. 物件若以傳址呼叫、參考呼叫的方式作為引數傳遞時，並不會自動呼叫複製建構元。
5. 類別中資料成員無法宣告為虛擬資料成員。
6. 在類別中把成員函式宣告為虛擬函式之後，則繼承此類別的子類別，所定義的虛擬函式，必須函式名稱、函式回傳值型別與參數列都必須相同，否則會被視為是 2 個不同的函式；如此一來，就無法達到繼承虛擬函式的效果了。
7. 類別中以虛擬函式的方式進行動態連結，以達到呼叫正確的成員函式。此種方式也是物件導向程式設計的一個特色：同名異式（Polymorphism，或稱為多形）。

分析與討論

1. 在 29-3 節多重繼承中討論「父類別與子類別之轉型」，其中第 31-50 行的自訂函式 `show()` 接收的是參考呼叫的 `baseClass` 類別的物件。若要接收的是傳址呼叫的物件，則自訂函式 `show()` 應該改為：

```
31 void show(baseClass *obj, int type)
32 {
33     switch (type)
34     {
35         case 0: //baseClass
36             cout << "str= " << obj->str << ", ";
37             obj->show();
38             break;
39
40         case 1: //sub1Class
41             cout << "str= " << ((sub1Class *)obj)->str << ", ";
42             ((sub1Class *)obj)->show();
43             break;
44
45         case 2: //sub2Class
46             cout << "str= " << ((sub2Class *)obj)->str << ", ";
47             ((sub2Class *)obj)->show();
48             break;
49     }
50 }
```

在主函式 `main()` 中，呼叫自訂函式 `show()` 時，所傳入的物件也應該為傳入物件的位址：

```
52 int main()
53 {
54     baseClass baseCls;
55     sub1Class sub1Cls;
56     sub2Class sub2Cls;
57
58     show(&baseCls, 0);
59     show(&sub1Cls, 1);
60     show(&sub2Cls, 2);
61 }
```

程式碼列表

```

1  #include <iostream>
2  using namespace std;
3
4  class Person // 基本資料：姓名與年齡
5  {
6      private:
7          string name = ""; // 姓名
8          int age = 0; // 年齡
9
10     protected:
11         void setNameAge(string name, int age)
12         {
13             this->name = name;
14             this->age = age;
15         }
16
17         void showInfo() // 顯示資料
18         {
19             cout << "姓名：" << name << " 年齡：" << age << endl;
20         }
21 };
22
23 class Basic // 基本資料：身高與體重
24 {
25     public:
26         double height = 0; // 身高
27         double weight = 0; // 體重
28
29         virtual double getBMI() = 0; // 計算 BMI
30 };
31
32 class Student :public Person, protected Basic // 學生資料
33 {
34
35     public:
36         string classNo = ""; // 班級
37         int schNo = 0; // 號碼
38
39         void setNameAge(string name, int age) // 設定姓名與年齡
40         {

```

```

41         Person::setNameAge(name, age);
42     }
43
44     void setH_W(double height, double weight) // 設定身高與體重
45     {
46         this->height = height;
47         this->weight = weight;
48     }
49
50     virtual double getBMI() // 重新定義 getBMI()
51     {
52         double h = height / 100.0;
53         return weight/ (h * h);
54     }
55
56     void showInfo() // 顯示資料
57     {
58         Person::showInfo();
59         cout << " 身高：" << height << "cm 體重：" << weight << "kg ";
60         cout << "BMI：" << getBMI() << endl;
61         cout << " 班級：" << classNo << " " << schNo << " 號 " << endl;
62     }
63 };
64
65 int main()
66 {
67     Student Mary;
68
69     Mary.setNameAge(" 王瑪莉 ", 19); // 設定姓名與年齡
70     Mary.setH_W(162, 50); // 設定身高與體重
71     Mary.classNo = " 一年 A 班 "; // 設定班級
72     Mary.schNo = 12; // 設定號碼
73
74     Mary.showInfo(); // 顯示學生基本資料
75
76     system("pause");
77 }

```

本章習題

1. 設計 1 個磅轉公克的基礎類別，再設計 1 個公克轉公斤的衍生類別，並繼承此基礎類別。使用此 2 個類別寫一輸入磅並轉成公斤的程式。
2. 設計用於計算圓形面積之類別 `Circle`：有 1 個表示圓半徑之資料成員 `radius`，以及用於計算圓面積的成員函式 `compute()`。再設計計算圓柱體體積之類別 `Cylinder`，此類別繼承 `Circle` 類別，並重載 `compute()` 成員函式，使之用於計算圓柱體之體積。宣告 1 個 `Cylinder` 類別之物件 `cls`，並由 `cls` 計算圓形面積與圓柱體體積。
3. 承第 2 題，宣告 2 個 `Cylinder` 類別之物件：`clsA` 與 `clsB`，以及一個指標型別的物件 `clsC`；如下所示。修改 `Circle` 與 `Cylinder` 類別，由物件 `clsA`、`clsB` 與 `clsC` 計算圓柱體體積。

```
Cylinder clsA;
Cylinder clsB(clsA);
Cylinder* clsC;
```

```
clsC = &clsA;
```

4. 有 3 個類別：`Tea`、`Milk` 與 `Tea_Milk`，分別代表泡茶、沖泡牛奶與製作奶茶，並且 `Tea_Milk` 類別繼承 `Tea` 與 `Milk` 此 2 個類別。`Tea` 類別用於讓使用者挑選茶葉、泡茶的濃度，`Milk` 類別用於讓使用者選擇牛奶、沖泡牛奶的濃度。寫一程式宣告類別 `Tea_Milk` 物件，讓使用者挑選茶的種類、沖泡濃度、牛奶的種類與沖泡濃度。

