國立屏東大學 資訊工程學系 C++程式設計入門教材

# 16. 類別與物件

在前一章中，我們已經為讀者說明了在應用程式構思階段的兩件重要的抽象化工作：決定代表應用程式的物件導向世界要由哪些物件所組成，並決定每個物件的屬性、行為及物件與物件間的關係。為了要建構物件導向世界，我們首先必須依據構思階段的決定來定義類別(Class)，然後再用類別來產生所需的物件(Object)。本章將說明並示範如何使用C++語言來完成類別的定義，以及如何產生物件。

## 16.1 類別定義

完成前一節的構思後，我們已經掌握了在物件導向世界裡應該會有哪些物件存在，但問題是，在你動手建構這個世界之前，你有沒有先定義這些物件的屬性與行為呢？換句話說，有沒有先定義其所屬的類別呢？更具體來說，在構思完成後，你還有以下的工作要進行：

- 將物件所屬的類別先加以定義 — 依據構思時所決定的物件屬性與行為，將真實或虛擬世界中的人、事、時、地、物等抽象化為程式語言中的類別。定義各物件所屬的類別。
- 宣告與建立各式類別的物件 — 依據需求在程式中宣告並產生各個類別的物件。我們把依據類別定義產生物件的動作實體化(Instantiate)，至於產生出的物件被稱為物件實體(Object Instance)
- 實現程式所需提供的各式功能 — 透過設定或改變物件的屬性數值、呼叫物件的函式、進行物件與物件間的互動或是訊息的傳遞等，來加以實現(當然這部份就必須等我們學到更多C++語言類別與物件的相關語法後，才能繼續加以討論)。

至於類別的定義其實並不困難，正如我們已經在之前的討論中所看到過的一樣，只要使用類似結構體的定義方式，將類別相關的屬性及行為加以定義即可。但在開始介紹類別定義語法前，先讓我們說明一下與類別相關的術語，因為物件導向在不同程式語言所使用的術語並不一致，所以有時會發生同樣一個東西在不同地方卻會看到不一樣的名詞的困擾；table 1將在談論物件導向概念時，類別的屬性與行為這兩個名詞，在C++語言、Java語言以及UML所使用的名詞加以彙整，請讀者自行加以參考；其中對我們現階段最為重要的就是C++語言，它是使用「資料成員(Data Member)」與「成員函式(Member Function)」來稱呼類別的屬性與行為。

| 概念 | UML | C++語言 | Java語言 |
|---|---|---|---|
| 屬性(Attribute) | 屬性(Attribute) | 資料成員(Data Member) | 欄位(Field) |
| 行為(Behavior) | 操作(Operation) | 成員函式(Member Function) | 方法(Method) |

Tab. 1: 物件導向術語彙整

### 類別定義語法

現在，請參考以下的C++語言類別定義語法：

**類別定義語法**

```
class 類別名稱
{
public:
   // 資料成員宣告
   [資料型態 變數名稱[, 變數名稱]*];*

   // 成員函式宣告(與實作)
   [回傳型態 函式名稱([參數型態 參數名稱 [, 參數型態
參數名稱]*]?)
    [{
      // 函式實作
      程式敘述;*
    }]?
};
```

其語法上大致與結構體的定義相同，除了必須提供類別名稱以外，還要記得在大括號裡的開頭處寫下public:(關於此點我們將在下一章進行說名)，另外就是在其大括號的內部宣告相關的資料成員與成員函式，請參考以下的說明：

- 資料成員(Data Member)此部份就如同結構體一樣，可以定義多個與此類別相關的資料項目，但要注意的是它並不允許在宣告變數時給定初始值(但從C++11起已允許初始值給定)。
- 成員函式(Member Function):此部份語法就如同原本C++語言的函式定義一樣，只不過現在是寫在類別內部。

**世上大多數的程式設計師都曾犯過的錯！！！**

請特別注意上述的語法，在類別定義完成後，在其右大括號的後方還必須加上一個分號做為定義敘述的結尾。但包含筆者在內，幾乎所有的程式設計師可能都曾不小心忘了加上分號，導致發生編譯時的語法錯誤！由於這個錯誤實在太常見了，因此才會在此特別提醒大家，請別忘了：

類別定義完成後要記得加分號、類別定義完成後要記得加分號、類別定義完成後要記得加分號！

### 類別定義範例

以下我們使用本章前面所使用的客戶與學生範例，示範如何使用C++語言進行Customer類別與Student類別的定義：

```
class Customer
{
public:
   string name;
   string ID;
   string accountNo;
   int    balance;
```

```
   int interest(double rate)
   {
      return balance*rate;
   }
};
```

```
class Student
{
public:
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
   }
};
```

上面的例子相信讀者應該很容易理解，在此不另行說明。不過前面我們也提到C++語言已經把結構體升級為2.0，所以以下的結構體定義也是正確的：

```
struct Customer
{
   string name;
   string ID;
   string accountNo;
   int    balance;
   int interest(double rate)
   {
      return balance*rate;
   }
};
```

```
struct Student
{
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
```

```
   }
};
```

雖然你現在不論怎麼看，都只會覺得C++語言的類別與結構體定義除了一個用class一個用struct以外，一個有寫public一個沒寫以外，它們根本是完全相同的！但是筆者要提醒讀者，它們兩者間還是存在著其它差異，但現在還不是為你揭曉的時候，等到下一章我們介紹到存取控制後，自然就能向你說明、讓你明白。不過如果是過去已經學過C語言的讀者，你現在至少看得出結構體1.0和2.0的差別了 — 1.0只能定義結構體相關的資料項目，2.0還可以定義專供其使用的函式！

本節最後要提醒讀者，完成類別的定義以後，我們就可以把類別視為模具，未來就可以在程式裡面用來生產物件了！

# 16.2 物件變數與實體

接續前一小節的類別定義，現在我們就可以將類別視為模具，並用來生產物件了。請參考以下的程式碼，雖然它看起來就好像將Student類別視為型態，並宣告了一個變數bob但是它其實是幫我們產生出一個“Student類別的物件 ”：

```
Student bob;
```

更具體來說，上面這行宣告敘述將會在記憶體裡產生Student類別的物件實體，並且使用物件變數名稱bob做為後續存取該物件實體的資料成員與呼叫其成員函式的識別字。

> 等等… 完全看不懂你在說什麼…
>
> 別擔心，看不懂是正常的。本節的目的就是要讓你能夠看懂它…

筆者想先問讀者們一個問題：

「我們將Student類別視為型態所宣告的變數bob到底是什麼？」

> 前面不是有說過，不就是物件嗎？
>
> 嗯，這個回答算是一半對、一半不對！ 讓我們先回想一下變數到底是什麼？請參考以下的程式碼：

```
int x;
...
x=38;
```

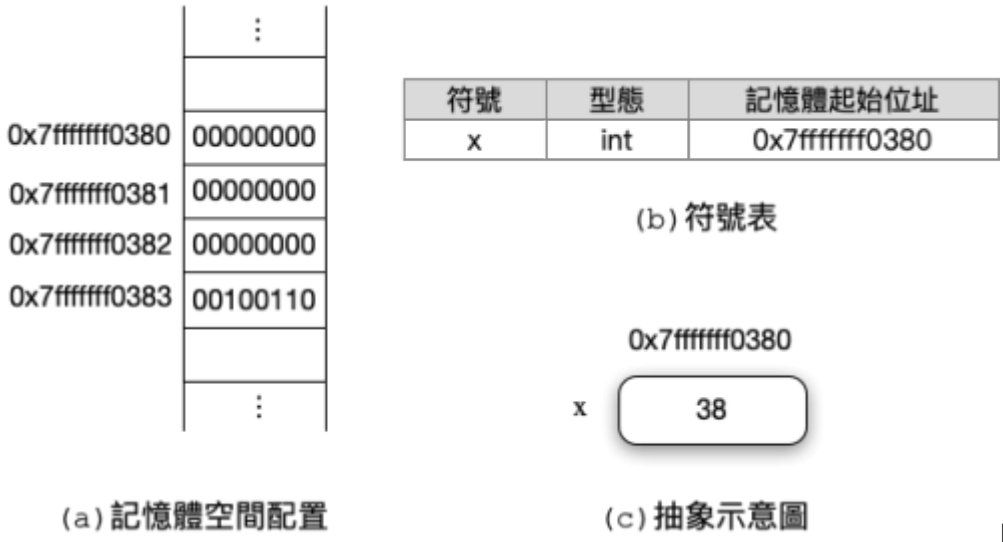



Fig. 1: 變數x記憶體配置。

在上面這段簡單的程式裡，其實有一些編譯時的細節你必須明瞭。還記得我們在第14章 淺談記憶體管理所介紹過的自動變數嗎？此處所宣告的int x;就是一個自動變數，假設int整數佔4個Bytes當編譯器編譯到int x;這行宣告時，會在記憶體裡面配置一塊連續4個Bytes的空間供其使用，假設是配置在從0x7fffffff0380到0x7fffffff0383的記憶體位址裡(請參考figure 1(a))完成記憶體配置後，編譯器會把|x|int|0x7fffffff0380|等資訊寫入到符號表裡(如figure 1(b))

當後續處理到運算敘述x=38;時，編譯器(其實完全不知道x是什麼東東，所以)必須去符號表裡搜尋看看能不能找到x當然在此例裡，其搜尋結果顯示x是一個int整數且被配置到0x7fffffff0380位址開頭的記憶體空間，因此編譯器才有能力完成x=38;的運算敘述 — 將整數值38以二進制的方式保存在記憶體位址0x7fffffff0380到0x7fffffff0383裡面。

至此，我們完成了上述程式碼在編譯過程中的細節說明。現在，你覺得變數是什麼？其實在上述程式碼裡面的x對於編譯器來說，它只不過是一個符號而已，就像每個人一個都有一個“名字”，“名字”本身並不是“真正的你”，只不過是一個“用來識別你的方法”而已。所以在程式碼中的x就只是一個「識別字(Identifier)」我們所有透過x所進行的動作，最終都必須作用在“所配置的記憶體空間」裡；就好比不管”真正的你“長高了幾公分，你的名字也不會跟著長高一樣x=38;的作用，並不是把x變成38，而是把38放到x所代表的那塊記憶體空間裡。

所以到底變數是什麼？變數是一塊在記憶體裡的連續空間，空間的大小由其型態決定，並可用來存放特定的數值。那什麼又是變數名稱呢？ 變數名稱只是一個識別字，用以幫助我們以及編譯器去識別那些在記憶體裡面真正用來存放數值的連續空間。不過在習慣上，絕大部份的程式設計師都是用抽象的方式，直接將變數名稱就視為是變數(例如直接把x視為變數)，請參考figure 1(c)儘管你現在已經學習到「變數x真正的意涵應該是「在記憶體裡的某塊可以使用x加以識別的連續空間」，但是筆者還是建議你維持原本的稱呼方法，還是直接簡單一點用抽象的方式稱呼為「變數x就好 — 這就是抽象的意思，把複雜的細節隱藏起來，讓我們更容易於溝通與表達並專注於邏輯上的意涵。試想以x=x+y;為例，你是想要完全不抽象、完完整整的說成：「把在記憶體裡的某兩塊可以使用x與y加以識別的連續空間裡的數值相加，並把結果放到在記憶體裡的某塊可以使用x加以識別的連續空間裡。」，還是選擇抽象一點、簡化一點，直接說成「把變數x和y的數值相加，並把結果放到變數x裡」？ 我想大部份的人都會選擇後者。

現在，讓我們回到用以產生Student類別的物件的宣告敘述:

```
Student bob;
```

經過前面對於變數x的討論，讓我們再回到開頭所討論的問題：

「我們將Student類別視為型態所宣告的變數bob到底是什麼？」

> 如果問題裡的 ” 變數bob“指得是宣告後所配置到的記憶體空間，那麼我的答案是「物件」！

因為像是Student bob;這樣的宣告敘述，其實會在記憶體裡面得到一塊可以放得下”Student類別的物件 “ 的記憶體空間(如果你對於Student類別的物件佔多少記憶體空間感到好奇，你可以使用sizeof去找到答案)，並以bob做為識別字，幫助我們在後續的程式裡去使用配置給”Student類別的物件 “ 的記憶體空間，包含去存取它的資料成員以及呼叫它的成員函式。

> 但如果問題裡的 ” 變數bob“指得就是bob那麼很抱歉，我的答案是它們是變數名稱、是識別字，是用來幫助我們(與編譯器)去找到、去使用在記憶體裡配置給物件的空間。

其實像是Student bob;這樣的宣告敘述，和int x;一樣是使用自動配置的記憶體空間，所以它會被 ” 自動 “ 的配置到一塊適合Student類別大小的空間供其使用，而這個所配置到的空間裡的內容就是依據Student類別的定義所配置的，裡面可用來存放其所有相關的資料成員，以及它的成員函式的程式碼，請參考figure 2 — 這塊在記憶體空間才 ” 物件 “ 的本體，依物件導向的術語還可以將其稱為「物件實體(Object Instance)」至於那個宣告時所使用的bob只是一個識別字，它只是用來幫助我們在後續的程式裡去使用 ” 物件實體 “ (也就是幫我們去存取那塊配置給物件使用的記憶體空間)。

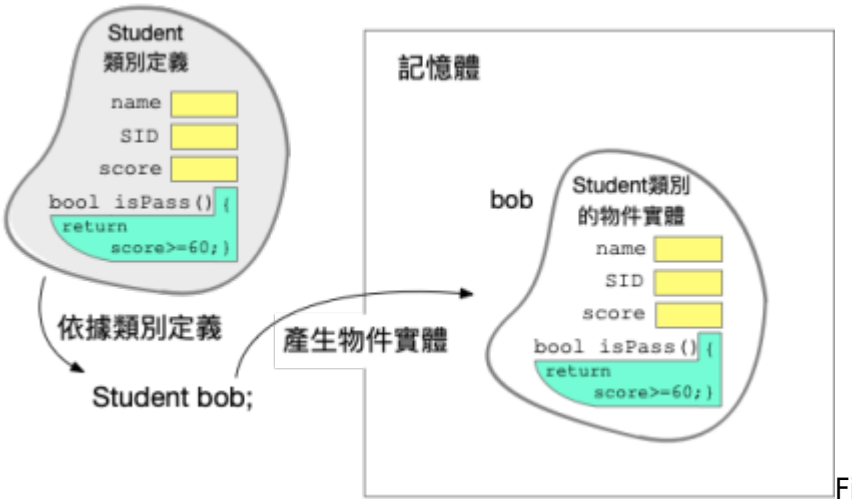


Fig. 2: Student類別與其物件實體

釐清了相關觀念後，現在讓我們再回答一次：

「我們將Student類別視為型態所宣告的變數bob到底是什麼？」

> bob是什麼？bob是一個識別字，它可以用來幫我們存取在記憶體裡Student類別的物件實體的資料成員，或是呼叫其成員函式。

最後，就如同我們所討論過的「變數x與「在記憶體裡的某塊可以使用x加以識別的連續空間」一樣，儘管我們現在已經明白Student bob;的相關細節，但如果每次都把物件說成「在記憶體裡的某塊依據Student類別所加以配置的連續空間」、把bob說成「可以用來存取在記憶體裡的某塊依據Student類別所加以配置的連續空間的資料成員，以及呼叫該塊連續空間裡的成員函式的識別字」，似乎也太過於麻煩和太過於精確了。筆者建議使用以下的方法稱呼它們:

- 物件/物件實體：在記憶體裡配置的空間叫做物件，這點是沒有疑問的，但可以稱為「物件實體」會更為精確。
- 物件變數：至於宣告類別物件時所使用的名稱，儘管只是個識別字，但就像「變數x一樣，還是有人也抽象地它為「物件」，不過筆者覺得這樣很容易和 ” 物件實體 “ 混淆，所以建議的稱呼是「物

件變數」。

好了，最後讓我們再回答一次：

「我們將Student類別視為型態所宣告的變數bob到底是什麼？」

> bob是在記憶體裡面的一個Student類別的物件實體的變數名稱
>
> > 嗯，回答得很好！可以再…再簡化一點嗎？
>
> bob是Student類別的物件變數！
>
> > 太棒了！聽起來你已經完全搞懂了！

# 16.3 物件實體化

依據類別定義在記憶體裡配置空間給其物件使用這件事情，就稱為物件的實體化(Object Instantiate) — 就是把物件實體產生出來的意思！本節將說明C++語言如何進行物件的實體化，並介紹如何為其資料成員進行初始值給定。

## 16.3.1 自動物件變數

首先第一種方式就是宣告物件變數，可在定義類別後將類別視為型態進行變數宣告，或是在定義類別的同時進行宣告(就好像結構體一樣)：

```
class Student
{
public:
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
   }
};

Student bob;
```

```
class Student
{
public:
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
   }
} bob;
```

正如我們在上一節所說明過的，上面的做法將會 ” 自動 “ 地配置一塊適合Student類別的物件的記憶體空間 — 換句話說，Student類別的一個物件實體被 ” 自動 “ 地產生出來了。此部份的概念，也可以參考figure 2。

### 16.3.2 全域與區域

如同變數的宣告一樣，類別的定義以及物件的宣告在程式中的位置就決定了它們可以被使用的地方，以下是將類別定義為全域的(意即將它宣告於所有函式之外)例子：

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

class Student // Student類別定義是全域的
{
public:
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
   }
} bob; // 使用全域的Student類別宣告全域的bob物件變數

Student robert; // 使用全域的Student類別宣告全域的robert物件變數

int main()
{
   Student alice; // 使用全域的Student類別宣告區域的alice物件變數
...
 // 此處除alice區域物件變數外，另有bob與robert全域物件變數可以使用
}
```

在上面的例子裡，由於Student類別被定義於所有的函式以外，所以它是屬於所謂的「全域類別定義(Global Class Definition)」，可以在程式的任何地方被使用 — 例如我們分別在第14、16與19行使用此全域類別來宣告物件變數bob、robert與alice，又因為它們被宣告的位置不同，其中bob與robert是全域的物件變數，至於alice則是宣告在main()函式裡的區域變數。

既然有全域類別，那當然也會有所謂的區域類別(Local Class)，請參考下面的例子：

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

Student robert; // 此行會發生編譯錯誤，因為全域的Student類別定義並不存在

int main()
{
   class Student // 定義在main()函式內部的Student類別，是僅限於main()函式裡使用的區域類別
   {
   public:
```

```
      string name;
      string SID;
      int    score;
      bool isPass()
      {
         return (score>=60);
      }
   } bob; // 使用區域的Student類別宣告區域的bob物件變數

   Student alice; // 使用區域的Student類別宣告區域的alice物件變數
...
 // 此處僅有bob與alice區域物件變數可以使用
}
```

### 16.3.3 匿名類別

有的時候，我們在程式裡所定義的類別(不論全域或區域類別)可能只有被用來進行一次物件宣告，且在後續的程式碼裡都沒有再次使用到該類別；要知道，儘管它是一次性的，但在定義時，我們還是必須完整地為此類別命名、定義資料成員與成員函式C++語言針的這種情況，提供 " 可省略類別名稱 " 的做法，我們將其稱為「匿名類別(Anonymous Class) — 沒有名字的類別(或是叫做 " 不需要名字的類別 " 更為恰當啦。請參考以下的範例：

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

int main()
{
   class // 在main()函式裡定義一個匿名的區域類別
   {
   public:
      string name;
      string SID;
      int    score;
      bool isPass()
      {
         return (score>=60);
      }
   } bob; // 宣告此匿名類別的區域物件變數
...
}
```

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

```
class // 定義一個匿名的全域類別
{
public:
   string name;
   string SID;
   int    score;
   bool isPass()
   {
      return (score>=60);
   }
} bob; // 宣告此匿名類別的全域物件變數

int main()
{
...
}
```

### 16.3.4 資料成員預設值

讓我們再回顧一下figure 2

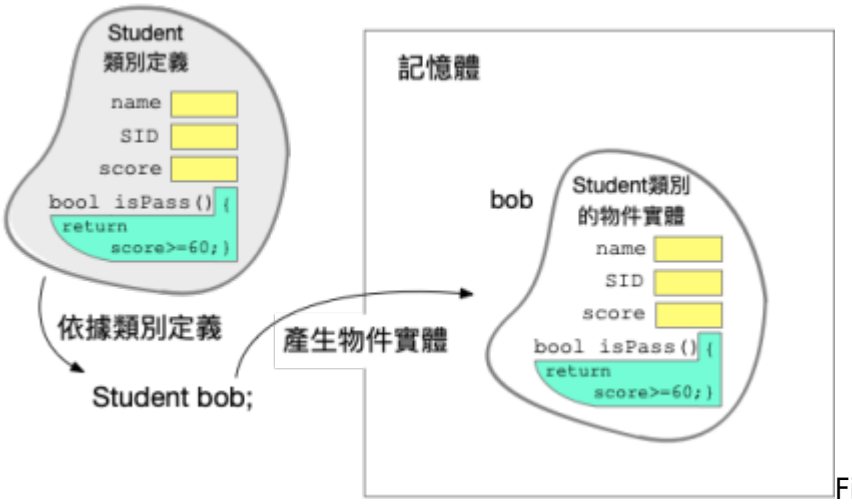
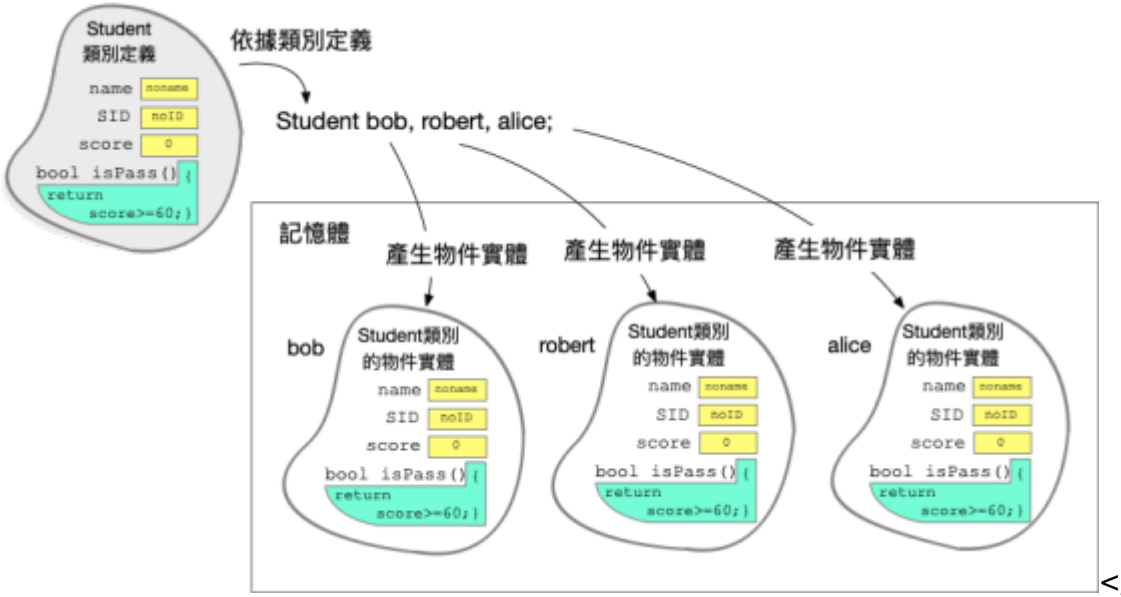


Fig. 2: Student類別與其物件實體

細心的讀者應該已注意到在figure 2裡，物件實體裡的資料成員都是空白的，這代表它們還沒有被賦與任何的初始值。自C++11起，支援給定類別資料成員預設值(Default Member Value)請參考下例(必須使用-std=C++11或-std=gnu11編譯器參數) <sxh cpp; title:類別資料成員預設值範例 defaultMemberValue.cpp; highlight:[7-9,14-17]> #include <iostream> using namespace std; class Student { public: string name="noname"; string SID="noID"; int score=0; bool isPass() { return (score>=60); } void showInfo() { cout << name << "(" << SID << ") " << score << endl; } }; int main() { Student bob, robert, alice; bob.showInfo(); robert.showInfo(); alice.showInfo(); } </sxh> 此程式在第7至9行為所有的資料成員設定了數值，這些數值被稱為預設值，請參考<imgref Fig_defaultvalue> 你可以看出現在依據Student類別所產生的所有物件實體的資料成員都會具有這些數值 <WRAP center> <imgcaption Fig_defaultvalue center 依據類別定義(含成員預設值)產生的物件實體>



</imgcaption>
</WRAP> 另外，為了便利理解此程式的執行結果，我們在第14-19行新增了一個成員函式showInfo()，它會將物件的資料成員的數值加以輸出，其執行結果如下：

```
noname (noID) 0
noname (noID) 0
noname (noID) 0
```

### 16.3.4.1 資料成員初始值給定

不同於上一小節所提到的資料成員預設值，本節將介紹的資料成員初始值給定(Data Member Initialization)是指在產生物件時對於資料成員設定的初始數值(Initial Value)，我們在13.1.1 結構體變數裡所提到的結構體變數的初始值給定方法，也適用於類別的物件實體 — 在類別物件變數宣告時以「={…}」方式，依類別定義資料成員的順序進行給定即可，請參考下面的例子

```
class Student
{
public:
   string name="noname";
   string SID="noID";
   int    score=0;
   bool isPass()
   {
      return (score>=60);
   }
} bob={"Bob Dylan", "CBB11100000", 100};

Student robert={"Robert", "CBB11100001", 59};
Student alice; // 未給定初始值，所以依資料成員預設值給定
```

在此例中所宣告的3個Student類別的物件變數bob，robert與alice，其中bob與robert有依據資料成員的宣告順序給定初始值，只於alice的部份沒有給定所以依上一小節的資料成員預設值給定。此程式的執行結果可以參考figure 4。

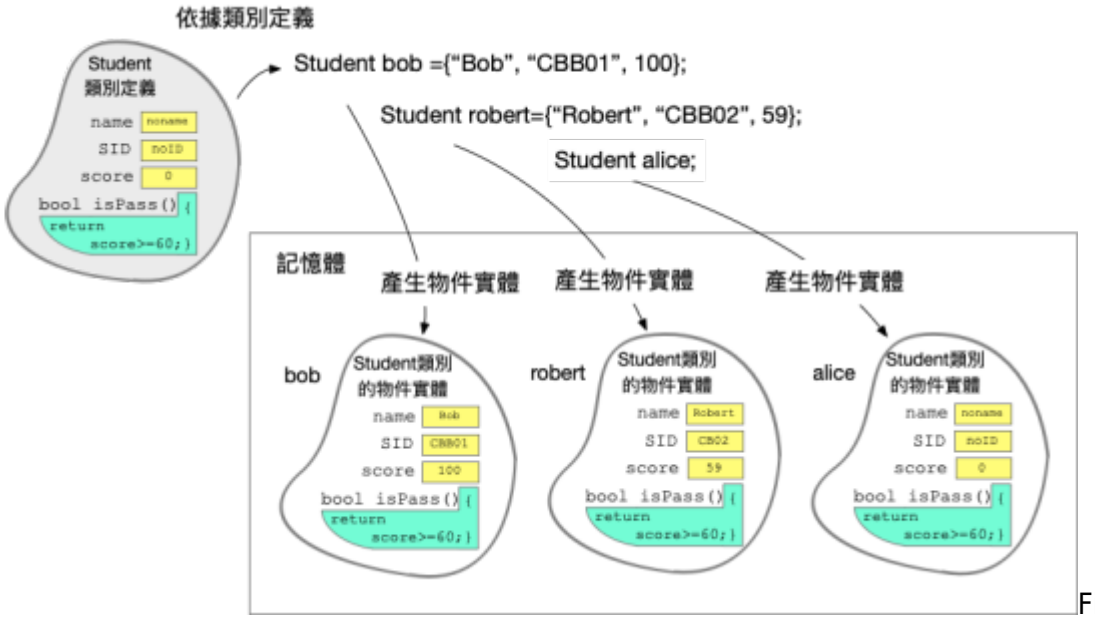


Fig. 4: 物件實體的資料成員初始值給定

若不想依據宣告時的順序，或是有想要忽略的資料成員，那麼就如同結構體一樣可以使用指定初始子(Designated Initializer)來加以完成，請參考以下的例子：

```
Student bob={.name="Bob", .SID="CBB01", .score=100}; // 可使用指定初始子註明所要設定初始值的資料成員
Student robert={.score=59, .SID="CBB02", .name="Robert"}; // 可不依原宣告順序
Student alice={.SID="CBB03", .name="Alice"}; //缺少初始值給定的score資料成員將會依資料成員預設值給定
```

### 16.3.5 動態記憶體配置

除了利用物件變數宣告讓C++幫我們 ” 自動 “ 產生物件實體以外，還有另外一種產生物件實體的方法，透過new與delete來自行決定何時要產生物件實體所需的記憶體空間，何時又要將其加以收回 — 我們把這種做法稱為「動態記憶體配置(Dynamic Memory Allocation)」或是稱為「物件實體化(Object Instantiation)」關於new與delete可參考本書14.3 動態儲存。 以下的程式碼依據Student類別的定義，在記憶體裡面動態配置一塊空間：

```
new Student;
```

當然，經過本章前面到目前為止的討論，相信讀者應該可以理解此處所配置的空間就是Student類別的物件實體；更簡單來說，上面這行程式，動態產生了一個Student類別的物件實體。在完成記憶體空間配置後，new將會把所配置到的記憶體空間的起始位址傳回，請參考以下的程式碼：

```
cout << hex << (new Student);
```

其可能的執行結果(由於每次執行時程式所配置到的記憶體位置並不相同，此記憶體位址依實際執行結果而定)如下：

```
0x60000144c200
```

好了，我們動態產生了一個物件實體，它被配置在0x60000144c200的位址，然後呢？我們要如何使用它

呢？答案是用指標，請參考下面的程式碼：

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

class Student
{
public:
   string name="noname";
   string SID="noID";
   int    score=0;
   bool isPass()
   {
      return (score>=60);
   }
   void showInfo()
   {
      cout << name << "(" << ID << ") " << score << endl;
   }
};

int main()
{
   Student *bob = new Student;
   bob->showInfo();
   (*bob).showInfo();

}
```

在上述的程式裡，第22行的Student *bob = new Student;做了以下的事情： - 等號左邊的Studnet *bob宣告了一個指標bob(指標其實也就是變數，假設它被配置到0x7fffffff0680的位址)，它應該要指向在記憶體裡某個Student類別的物件實體所在的位址； - 等號右邊的new Student將會使用動態記憶體配置方法來產生了一個Student類別的物件實體； - 最後，等號把右邊傳回的記憶體位址賦與給左邊。 好了，大功告成，請參考figure 5，它顯示了指標bob的值為Student類別的物件實體所在的記憶體位址：

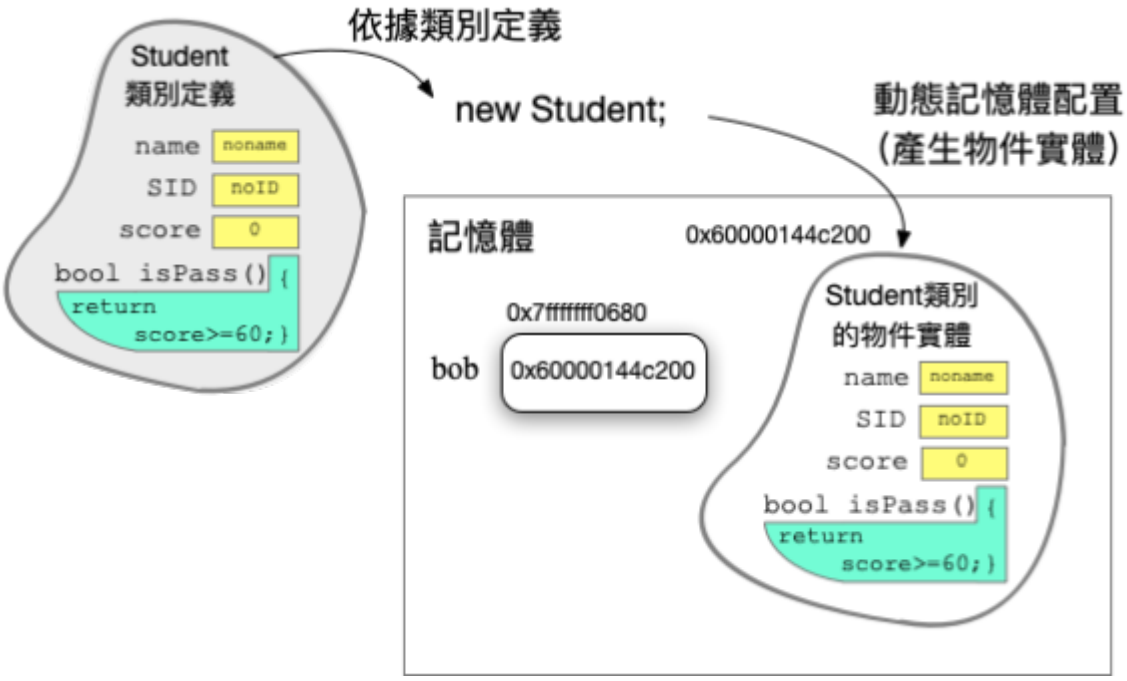



Fig. 5: 透過指標來使用動態配置的物件實體

當然，我們也可以用比較抽象的方式來看待，請參考figure 6

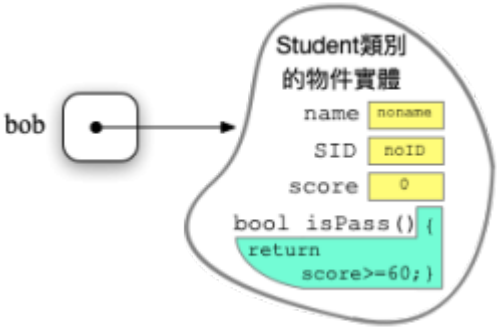



Fig. 6: 指向物件實體的指標

另外，要提醒讀者注意的是，當透過指標來使用物件實體時，就好比透過指標來使用結構體時一樣，要存取其資料成員與成員函式時，必須改用 -> 成員選取運算子(Member Selection Operator)又叫作成員選取子(Member Selector)例如在上述程式中的第23行，我們透過bob指標去使用Student類別的物件實體裡面的showInfo()成員函式 —bob->showInfo();當然，我們也可以先用 * 間接存取運算子(Indirect Access Operator)來取得指標所指向之處的值(此處的 " 值 " 為物件實體)，然後就可以使用原本的 . 來呼叫它的成員函式，例如第24行的(*bob).showInfo();

## 16.3.6 動態物件實體陣列

當然，我們也可以動態建立物件實體的陣列：

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

class Student
{
```

```
public:
   string name="noname";
   string SID="noID";
   int    score=0;
   bool isPass()
   {
      return (score>=60);
   }
   void showInfo()
   {
      cout << name << "(" << ID << ") " << score << endl;
   }
};

int main()
{
  Student *cise1A = new Student [50];
  Student *someone;

  csie1A[0].name="Amy";  //陣列元素是物件實體，所以使用.
  caie1A[0].showInfo();
  someone=csie1A;
  someone++;
  someone->name="Jacky";  //someone是指標，所以用->
  (*someone).showInfo();
  delete [] csie1A;
  return 0;
}
```

上述程式所進行的工作並沒有特別的意義，主要只是做為產生動態物件實體陣列的示範。

# 16.4 類別定義與實作架構

通常，我們會將類別的定義與實作分開放在不同的檔案： === 類別定義 ===

```
#ifndef _STUDENT_
#define _STUDENT_

class Student
{
public:
  string name;
  string SID;
  int    score;
  bool   isPass();
  void   showInfo();
};
#endif
```

=== 類別實作 === 我們也可以在類別定義以外的地方進行成員函式的實作，只要記得在實作成員函式時，在函式名稱前加上「類別名稱::」表明其所屬的類別即可。

```
#include <iostream>
#include "student.h"
using namespace std;

bool Student::isPass()
{
   return score>=60;
}

void Student::showInfo()
{
  cout << name << "(" << SID << ") " << score << endl;
}
```

=== 在含有main()函式的程式中使用類別 ===

```
#include <iostream>
#include "student.h"
using namespace std;

int main()
{
  Student *bob = new Student;
  bob->name="Bob";
  bob->SID="CBB01";
  bob->score=100;
  bob->showInfo();
}
```

=== 使用Makefile === 通常，將類別的定義與實作分開後，最麻煩的就是編譯時的步驟變多了，例如：

```
g++ -c student.cpp
g++ main.cpp student.o -o main
```

這時只要搭配Makefile就可以輕鬆地完成編譯的動作：

```
all: student.o
        g++ main.cpp studnet.o -o main

student.o: student.cpp
        g++ -c student.cpp

clean:
        rm -f *.o main *.*~ *~
```

本系系友Kito寫了一個不錯的入門網頁，大家可以去參考參考。

**建立inline的成員函式**

在C/C++語言中，我們可將函式以inline方式宣告，來增進效能。當我們呼叫一個inline函式時，編譯器會幫助我們將函式內的程式碼「複製」一份到其所被呼叫之處，並在該處執行程式碼，省略了函式「呼叫與傳回」的動作。因此，嚴格來說，inline函式並不是函式，而是類似#define一樣，是由編譯器進行前置的程式碼替代動作。

在C++中，我們也可以將類別的成員函式以inline方式宣告，並有以下兩種方式：

1.宣告時不須註明，而是在實作時前使用inline保留字即可，請參考下列的範例：

```
class Student
{
public:
  string name;
  string SID;
  int    score;
  bool   isPass();
  void   showInfo();
};

inline void Student::showInfo()
{
  cout << name << "(" << SID << ") " << score << endl;
}
```

2.直接宣告並實作於類別定義內，但在宣告時同樣不須註明 — 因為C++會幫我們把所有實作在類別定義內的成員函式都自動轉換為inline 函式。請參考下列的例子：

```
class Student
{
public:
  string name;
  string SID;
  int    score;
  bool   isPass()
  {
     return score>=60;
  }
  void   showInfo()
  {
```

```
    cout << name << "(" << SID << ") " << score << endl;
  }
};
```

要注意的是inline函視並不是絕對可以使用，一切必須視所使用的編譯器而定，例如：

- 一些編譯器並不支援在inline function中使用迴圈、switch或是goto等敘述；
- inline函式不可以設計為遞迴型式；
- inline函式內不支援宣告static的變數；
- 還有，最後最重要的一點，大部份的編譯器其實都沒有完全支援inline函式。

# 16.5 建構與解構函式

請回顧一下前面使用Student類別的物件實體的程式，若使用動態配置記憶體的方法，則通常會有以下這一行：

```
Student *bob = new Student;
```

請把它代換為：

```
Student *bob = new Student();
```

沒錯，就是在後面加上一組括號。請修改完成後再重新編譯與執行程式，看看有沒有什麼差別？嗯，沒錯，沒有任何差別！ 其實不論是new Student();或是new Student;除了其中的new Student是用來配置Student類別的物件實體所需的記憶體空間外，它還會幫我們呼叫一個名為Student()的成員函式 — 重點是不論你在new Student的後面有沒有加上括號都會做一樣的事情。 » 可是我沒有定義這個成員函式啊？ > 沒錯，我們的確沒有定義這個成員函式，可是C++的編譯器預設會幫我們在類別定義內產生一個與類別同名的成員函式，我們將其稱之為「建構函式(constructor)」，或稱為「建構子」。 > 例如Student類別會自動包含以下的程式碼

```
Student() { }
```

> 這是一個特殊的成員函式，沒有(也不能有)傳回型態的宣告，且其內容為空白(什麼都沒做)；在我們使用new Student();時(或者是new Student;)這個建構函式Student()會被自動呼叫! » 呼叫就呼叫，它的內容是空白的，那不就沒用處？ > 嗯，沒錯，空白是沒用。但是還記得C++讓我們可以視需要，自己寫一個新的版本來替代它！例如，我們可以寫一個用來給定資料成員初始值的建構函式，來代換掉那個空白的預設版本：

```
Student()
{
   name="unknown";
   SID ="unknown";
   score=0;
}
```

要注意的是，此處用來給定資料成員初始值的建構函式，其用途與前面我們所談論的資料成員預設值是類似的作用，都可以在沒有給定初始值的情況下，讓物件實體的資料成員能夠有初始的數值。但是資料成員預設值是從C++11後才被支援的，建構函式則是從一開始就有支援。

> 還記得第10章 函式介紹過的「同名異式」函式多載嗎？你也可以使用這個方法來提供不同版本的建構函式，例如:

```
Student (string n, string i)
{
   name=n;
   SID=i;
   score=0;
}

Student (string n, string i, int s)
{
   name=n;
   SID=i;
   score=s;
}
```

> 如此一來，你就可以在物件實體化的同時，順便把其資料成員做相關的初始值設定，而且可以視你手上有些什麼資料來呼叫不同的版本。例如以下的宣告:

```
Student *bob = new Student("Bob", "CBB01");
Student *robert = new Student("Robert", "CBB02", 99);
```

> 不過要特別注意，如果你定義了自己的建構函式(不論是有沒有參數的版本)，編譯器就不再幫我們產生預設的建構函式，因此以下的程式碼是錯誤的：

```
#ifndef _STUDENT_
#define _STUDENT_

class Student
{
public:
  string name;
  string SID;
  int    score;
  Student (string n, string i, int s);
  bool   isPass();
  void   showInfo();
};
#endif
```

```
#include <iostream>
```

```
#include "student.h"
using namespace std;

Student::Student(string n, string i, int s)
{
  name=n;
  SID=i;
  score=s;
}

bool Student::isPass()
{
   return score>=60;
}
void Student::showInfo()
{
   cout << name << "(" << SID << ") " << score << endl;
}
```

```
#include <iostream>
#include "student.h"
using namespace std;

int main()
{
  Student *bob = new Bob;
  bob->showInfo();
}
```

» 瞭解了！所以如果有自定建構函式時，我會小心實體化時呼叫的版本是否正確！ > 嗯，沒錯，不過建議讀者可以養成一個習慣 — 永遠為你的類別自己提供一個預設的版本！例如：

```
Student::Student()
{
   do whatever you like!
}
```

> 這樣一來，就萬無一失了！

# 16.6 成員初始化串列

學會了類別的建構函式該如何撰寫後，本節將介紹一個專屬於建構函式的「成員初始化串列(Member Initializer List)」用以在建構出物件實體時將指定的資料成員進行初始值的設定。成員初始化串列的語法是在建構函式後，使用一個冒號:並將所要設定初始值的資料成員的名稱及數值依下列語法進行設定：

**成員初始化串列語法定義**

:資料成員名稱(運算式),[資料成員名稱(運算式)]*

依據上述的語法，若要設定一個以上的資料成員初始值時，可以使用逗號加以分隔。另外，語法中的「運算式」的運算元可以是數值、建構函式的參數以及該類別的資料成員，但其運算結果必須與所要設定的資料成員型態一致(否則將啟動自動型態轉換，且當無法轉換為所需的資料型態時，就會得到編譯錯誤)。請參考下面的範例：

```
Student::Student(string n, string i, int s): name(n), SID(i),
score(s>100?100:s)
{
}
```

在上例中的成員初始化串列裡，我們使用了參數n及i做為資料成員name與SID的數值(最簡單的運算式就是不包含任何運算子，僅包含一個運算元的形式)，並且使用運算式s>100?100:s來設定資料成員score的數值，並確保大於100分的分數以100分計！ > 給讀者一個簡單的練習：請用同樣的成員初始化串列的方法，在Student類別建構函式裡除了將score設定為大於100分以100分計，還要檢查是否小於0分，若小於0分則以0分計。 請再看下面的例子：

```
Student::Student(): name("unknown"), SID(name), score(0)
{
}
```

在此例的成員初始化串列裡，先以”unknown“設定為資料成員name的數值，然後再以剛設定好的name做為SID的數值。但請讀者要特別注意的是，列示在成員初始化串列中要進行初始化的資料成員，其初始化的順序並不是依照在串列中的順序，而是依照在類別內宣告的先後順序進行。所以下面的做法在編譯時將會得到警告訊息：

```
Student::Student(): SID("unknown"), name(SID), score(0)
{
}
```

因為成員初始化串列的執行順序是以當初宣告在Student類別裡的順序加以決定，所以首先會執行的是name(SID)，然後才是SID(“unknown”)與score(0)，所以在執行第一個name(SID)時，由於另一個資料成員SID的值其實還沒給定，所以就會造成潛在的問題。

# 16.7 解構函式/解構子

除了建構函式之外，C++也會在我們進行delete以回收不再需要的物件實體時、或是當物件實體的生命週期結束時(例如在函式內的區域物件變數離開函式範圍，或是全域類別變數遇到程式執行結束時)，呼叫一個特別的成員函式，稱為「解構函式(destructor)」或「解構子」。如果你並沒有提供解構函式，那麼編譯器也會自動幫我們產生下面的函式：

```
Student::~Student() {  }
```

> 沒錯，又是空白的內容。所以你也可以實作自己的解構函式：

```
Student::~Student()
{
   cout << "Bye!" << endl;
}
```

解構函式通常不是用來做上面這種 ” 不知道算不算是很無聊 “ 的事情，它最常見的用途是幫我們將物件實體裡動態配置的記憶體空間加以回收，例如：

```
#ifndef _STUDENT_
#define _STUDENT_

class Student
{
public:
  char *name;
  char *SID;
  int   score;
  Student();
  ~Student();
  bool   isPass();
  void   showInfo();
};#endif
```

```
#include <iostream>
#include "student.h"
using namespace std;

Student::Student()
{
  name = new char[20];
  SID = new char[10];
}

Student::~Student()
{
  delete [] name;
  delete [] SID;
}

bool Student::isPass()
{
   return score>=60;
}
void Student::showInfo()
{
   cout << name << "(" << SID << ") " << score << endl;
}
```

```
#include <iostream>
#include "student.h"
using namespace std;

int main()
{
```

```
  Student *bob = new Bob;
  bob->showInfo();
  delete bob;
}
```

> 好了，打完收工。