國立屏東大學 資訊工程學系 C++程式設計入門教材

15. 走向物件導向世界

從現在起，本書正式朝向「物件導向世界」邁進。

C語言自1973年誕生以來，透過循序、選擇與重複等控制結構，再加上使用函式所能達成的模組化設計，有效滿足了當時絕大多數程式設計的功能需求。然而在過去一甲子裡，電腦系統的功能與使用者需求不斷地提升，為了因應隨之大幅增加的軟體開發困難度與複雜性，程式語言與程式設計的方法、工具與技術也隨之不斷演化。支援物件導向程式設計(Object-Oriented Programming)的C++語言即為一例，它不但承襲了來自C語言的特點，還引入了物件導向的特性，顛覆了以往程式設計以「功能」為核心的做法，轉變成為以「物件」為導向的思維方法。歷四十餘年的發展，C++語言已經成為資訊產業最主流的程式語言之一，同時也是支援物件導向的程式語言當中最為強大、但也最為複雜的語言。在開始進入物件導向的學習之前，本章將先為讀者說明物件導向技術是如何用來解決傳統程式設計既有的一些問題，並就物件導向程式設計的特性與思維方式加以簡介。

15.1 思維的演進

從有程式語言以來，程式設計師的工作就是創作程式來解決問題或滿足使用者的需求，但受到硬體環境的快速改變與使用者需求的持續提升的影響，我們的工作內容也變得愈來愈困難、愈來愈複雜。從C語言到C++語言，程式語言與程式設計的方法也都處於持續演化發展的階段，然而任憑程式語言的功能、特性如何地強大，最終只是一個被人用來設計程式的工具而已 — 人才是程式設計的主體，程式語言只是工具而已；既然程式語言是工具，那麼使用工具的方法就變得重要了起來。一直以來，不斷地有人提出包含IPO模型、結構化程式設計(Structured Programming)、函式程式設計(Functional Programming)、程序式程式設計(Procedure Programming)等數十種不同的方法曾先後被提出，然而程式設計問題的多樣性與複雜性，不太可能有一個單一的方法能滿足所有不同程式的設計需求；甚至更有趣的一點是，儘管有這麼多的程式設計方法被人發明出來，但絕大部份的程式設計師，都不會依照特定的設計方法來進行程式的開發！因為對於程式設計師而言，採用何種方法其實並不重要，重要的是如何解決問題與滿足使用者的需求而已！

大部份的程式設計師在累積一定數量的程式開發經驗後，或許是受限於傳統結構化程式語言的特性，往往都會在無形當中自然養成一種以“ 功能(Function)”為導向的思考方式[1)] — 針對所要開發的程式，首先思考的通常都是“ 程式該提供哪些功能 ”?後續才是思考“ 該如何用程式碼來完成這些功能 ”?例如一個銀行的客戶系統與學校的成績處理系統，分別需要為客戶計算其銀行帳戶餘額利息，以及判斷學生的修課成績是否及格；這些需求很直覺地就可以被轉換成程式所應該提供的功能，後續就可以使用程式語言的循序、選擇與重複等控制結構，依功能的邏輯與流程將這些功能開發為interest()與isPass()等函式。

> 就好比以英語為母語的人，並不需要刻意學習發音的方法一樣，筆者將這種 ” 自然而然 ” 形成的方法視為是“ 自然程式設計法 ” ！
>
> 好吧！我承認聽起來不怎麼樣，那就改稱為“ 以功能為導向的程式設計思維方法 ” 吧！這樣有沒有“ 水準 ” 高一些呢？

功能導向程式設計

「功能導向程式設計(Function-Oriented Programming)」一詞與一些既有的程式設計方法的名稱或有相似，例如程序導向程式設計(Procedure-Oriented Programming)與

函式程式設計(Functional Programming)等，但筆者在此要強調此名詞為本書自創，僅用以表示重視程式功能的開發方法(先確認程式所需的功能，再以循序、選擇與重複等控制結構來實現程式功能，並常常將功能開發為函式)，並無意引用其它程式設計方法的名稱或內涵，請讀者不要被它們看起來相似的名稱給混淆了。

本節後續將從這種“習慣成自然”的“功能導向程式設計思維方法”開始，帶領讀者一覽其優缺點，並逐步討論物件導向的思維方式是如何演化以因應相關問題。

15.1.1 功能導向程式設計方法

功能導向程式設計方法(Function-Oriented Programming Paradigm)可以視為是一種以目的為驅動的方法，針對所要開發的程式，首要的工作是思考“程式該提供哪些功能”?後續再思考“該如何完成這些功能”?考慮到模組化設計的好處，在大部份的情況下，我們都應該將所要開發的功能設計為函式；但是要提醒讀者注意的是，函式只是實作程式功能的一種具有模組性的做法，但不是強制性或必要的做法。事實上，本書所使用的功能導向一詞是用來表示程式的開發是功能為主體，但不限定實作的方法。

現在，讓我們以功能導向程式設計為例，考慮一個銀行的客戶系統與學校的成績處理系統，假設它們分別需要計算客戶的銀行帳戶餘額利息，以及判斷學生的修課成績是否及格等功能，我們可以將這兩個功能開發為兩個函式interest()與isPass()。

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

bool isPass(int score)
{
   return (score>=60);
}
```

假設程式裡有代表銀行客戶Amy的銀行帳戶餘額變數amy_balance。以及代表學生Bob的修課成績變數bob_score。我們可以將amy_balance及利率1%做為引數，對interest()函式進行呼叫，以得到其應得之利息金額；另外將bob_score做為引數，去呼叫isPass()函式，就可以得知Bob是否及格。請參考下的程式碼：

```
int amy_balance;
int bob_score;

cout << interest(amy_balance, 0.01); // 呼叫interest()取得Amy應得的利息金額
if (isPass(bob_score))               // 呼叫isPass()判斷Bob是否及格
   cout "Bob Pass" << endln;
```

我們可以把上述的這種做法，稱為「功能導向」並歸納如下：

功能導向的程式設計方法，是以“功能“為主體，將所需要的”功能“設計為函式，

再將相關的資料做為引數進行函式呼叫。

在功能導向的思維裡，功能是主角，提供功能所需的資料或是會受到這些功能影響的(例如客戶與學生)則稱為 ” 對象 ” — 它們只是配角。

錯誤使用函式

採用功能導向的程式設計方法時，為了實現特定功能所開發的函式，也可以在不同的程式裡再次使用，例如計算利息的interest()與及格與否的isPass()函式，未來也可以在銀行的定期存款(需要為客戶計算定存可實現利息收入)與學校的期中成績預警系統(需要針對期中成績不及格同學發出預警)中再次使用。函式可以重複使用的好處是顯而易見的(例如可以縮短軟體的開發時程、降低開發成本)，但卻也可能造成新的問題，例如同樣呼叫這些函式但卻提供了不正確的引數，雖然在語法上是正確的(所以能夠通過編譯)，但卻會帶來無意義(或不正確的)運算結果 — 這種情況就是主角與配角的組合出了問題，又可稱為「語法正確但語意錯誤」，是比單純語法錯誤更難發現及改正的錯誤。例如以下的兩個呼叫 ：

```
interest(bob_score, 0.01); // 錯把Bob的成績做為引數去計算利息 -- 是重修能賺利息的意思嗎？
isPass(amy_balance);       // 錯把Amy的存款餘額做為引數去判斷是否及格 -- 是有錢才會通過的意思嗎？
```

由於功能導向存在著上述的缺點，因此才有了後續程式設計思維的演進 — 從重視主角(功能)的思維，慢慢轉變為以 “ 對象 ” (配角)為核心。

鬆散的相關資料項目

當我們把上述討論的例子再擴大一些後，其實還有更多的問題將會浮現，例如一個銀行帳戶處理的系統，其實不太可能只考慮客戶的帳戶餘額和利息的計算而已，通常可能還包含有客戶的姓名、身份證字號、帳號等資訊以及包含存款、提款、轉帳、終止帳戶等功能；若暫時不討論功能的部份，僅就資料項目來看，我們至少要為Amy這位客戶新增以下的變數宣告：

```
string amy_name;
string amy_ID;
string amy_accountNo;
int    amy_balance;
```

同理，我們也必須為Bob這位同學新增一些變數：

```
string bob_name;
string bob_SID;
int    bob_score;
```

像這種針對單一“ 對象 ”(例如客戶、學生)宣告多個變數的做法，除了過於麻煩以外還有鬆散的問題；因為我們必須要宣告好多個變數，而且同一個“ 對象 ”的多個變數彼此間也不具有相關性。若是未來應用到更多個客戶、更多個學生時，事情就更為為麻煩了，因為我們就必須要為更多的客戶與學生宣告更多的變數：

```
string amy_name;
string amy_ID;
string amy_accountNo;
int    amy_balance;
string betty_name;
string betty_ID;
string betty_accountNo;
int    betty_balance;
string catherine_name;
string catherine_ID;
string catherine_accountNo;
int    catherine_balance;
…略
string bob_name;
string bob_SID;
int    bob_score;
string robert_name;
string robert_SID;
int    robert_score;
string tiffany_name;
string tiffany_SID;
int    tiffany_score;
…略
```

當然這個問題可以使用陣列來稍微改善，例如：

```
string customer_name[SIZE];
string ID[SIZE];
string accountNo[SIZE];
int    balance[SIZE];

string student_name[SIZE];
string SID[SIZE];
int    score[SIZE];
```

但這中將“ 對象 ”(也例子中的客戶與學生)的多個相關資料宣告為陣列的方法，儘管少了宣告大量變數的問題，但仍然沒有解決鬆散的問題 — 因為同一個“ 對象 ”的多個陣列彼此間還是不具備相關性！你怎麼知道customer_name[]、ID[]、accountNo[]與balance[]陣列都是屬於客戶的相關資料呢？你又怎麼知道student_name[]、SID[]與score[]陣列都是屬於學生相關的資料呢？

> 我當然知道啊！我們不是一直在討論這兩個例子嗎？

> 嗯，我的意思是，如果不是我們前述的討論，你能夠從程式碼本身看出來嗎？例如當你看到以下這些混在一起的宣告時：

```
string student_name[SIZE];
string ID[SIZE];
string SID[SIZE];
int    score[SIZE];
string accountNo[SIZE];
int    balance[SIZE];
```

> 你怎麼知道這些陣列所代表的不是學生的名字、身份證字號、學號、分數、領獎助學金的銀行帳號、學餐預付卡的餘額呢？

15.1.2 結構體

在過去，這些在程式碼裡鬆散的變數或陣列是否具有相關性，都必須透過我們另外給定的附加說明才能得知，無法直接從程式碼中加以理解。不過其實這個“ 鬆散 ”的問題其實並不難處理，只要採用結構體就可以解決了。具體做法是針對所要處理的“ 對象 ”使用結構體的方式，將其相關的資料項目集中起來宣告為一個“ 結構體 ”，然後再將我們所要處理的“ 對象 ”宣告為這個結構體的變數。例如銀行的客戶與學生就是我們所討論的“ 對象 ”，而他們的姓名、身份證字號、帳號、學號、成績等資訊就是所謂的相關資料項目。請參考以下的Customer與Student結構體宣告：

```
struct Customer
{
   string name;
   string ID;
   string accountNo;
   int    balance;
};
```

```
struct Student
{
   string name;
   string SID;
   int    score;
};
```

有了Customer與Student這兩個結構體後，就可以將Amy與Bob這兩個我們所要處理的“ 對象 ”宣告為這兩個型態的變數：

```
Customer amy;
Studnet  bob;
```

嗯，不錯，這樣就搞定了吧～ 使用結構體將同一個“ 對象 ”的相關資料項目收納在一起，就把“ 鬆散 ”的問題解決了！現在amy和bob是分別是Customer和Student結構體的變數，所有和amy與bob他們相關的資料項目都屬於他們自己：

```
// 通通都是amy的(.)
```

```
amy.name;
amy.ID;
amy.accountNo;
amy.balance;

// 通通都是bob的(.)
bob.name;
bob.SID;
bob.score
```

透過使用結構體，我們就可以從程式碼看出哪些資料項目是相關；甚至當你錯誤地使用它們時，現在也會有編譯器幫我們把關，例如如果你想要使用銀行客戶Amy的“ 成績 ”，或是學生Bob的“ 帳號 ”時，你將會看到以下的錯誤訊息：

```
 error: no member named 'score' in 'Customer'
    amy.score=100;
    ~~~ ^
error: no member named 'balance' in 'Student'
    bob.balance=200000;
    ~~~ ^
```

很好，這樣的確就不再“ 鬆散 ”了。此處所使用的結構體其實是C++語言承襲C語言而來的，因此上述的討論雖然是C++語言的程式碼，但其實也可以適用於C語言；因此筆者將其稱為「結構體1.0」，代表C語言和C++語言共通支援的結構體語法與功用，並將後續C++語言擴展加入新功能的結構體稱為2.0，我們將在下一小節接續說明。

雖然“ 鬆散 ”的問題被結構體1.0解決了，但是呼叫函式卻給了錯誤引數(語法正確但語意錯誤)的問題仍然還沒解決。請再看一次下面的例子：

```
interest(bob.score, 0.01); // 錯把Bob的成績做為引數去計算利息 -- 是重修能賺利息的意思嗎？
isPass(amy.balance);       // 錯把Amy的存款餘額做為引數去判斷是否及格 -- 是有錢才會通過的意思嗎？
```

是的，儘管我們利用結構體能夠將具有相關性的資料項目集中起來，讓amy的歸amy□bob的歸bob□但是我們仍然沒能阻止餵給函式錯誤引數的問題！同樣的問題仍然還是會發生。

為了解決這個問題，現在就輪到物件導向登場了！

15.1.3 從結構體到類別

物件導向的具體做法之一，就是針對所要處理的“ 對象 ”使用“ 類似 ”結構體的方式，將其相關的資料項目集中起來宣告為一個“ 長得很像結構體 ”的類別，然後再將我們所要處理的“ 對象 ”宣告為這兩個類別的變數…

> 等… 等一下，怎麼聽起來還是很像前面剛講過的結構體？(謎之聲：結構體好吃嗎？)
>
> > 嗯… 我必須承認，是的，這段是Copy過來改的….

> 因為類別可以說是從結構體演化過來的！(謎之聲：所以類別更好吃嗎？)
>
> 可以讓我先繼續講下去嗎？(是的，我覺得類別比結構體好吃！)

做為C語言的後繼者，C++語言除了新增物件導向的特性之外，它也承襲了絕大部份C語言既有的語法與功能，但為了讓它們與時俱進，C++也修改了部份C語言的語法、將它們升級成為更好的自己！

> 你又岔題了～～～
>
> 沒有，完全沒有，這段很重要，你要有耐心，你才能升級成為更好的自己！

C++把承襲自C語言的結構體1.0，增加功能擴充為結構體2.0，然後又把結構體2.0複製貼上成為了類別，最後又“稍微”修改了一下類別，讓它們倆非常相似，但卻又在細節上有些許差異。所以從結果來看，C語言的結構體是1.0和C++語言的結構體2.0是不相同的，兩者間具有比較大的差異；但C++語言的結構體2.0與類別基本上是差不多的，只有在細節上存在差異。

由於結構體2.0和所謂的類別基本上是相同的，所以我們可以把前面討論時所宣告的結構體，改宣告為類別 — 也就是將struct改為class：

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

```
class Student
{
public:
   string name;
   string SID;
   int    score;
};
```

上面這段程式碼的內容是所謂的類別定義，儘管我們還未開始說明其語法，但讀者們應該可以發現它與前面的結構體定義基本上是相同的，除了把struct換成class以外，就只有在大括號裡的開頭處新增了一行：public: — 關於其作用以及結構體2.0與類別間的差異，我們將在下一章加以說明。讀者們可以暫時先將此處所看到的“class”當做是“struct”，如此一來，Customer與Student就成了我們“自定的資料型態”，後續就可以將Amy與Bob這兩個我們所要處理的“對象”宣告為這兩個型態的變數：

```
Customer amy;
Studnet  bob;
```

此時我們所要處理的Amy與Bob這兩個“對象”，就變成了在程式中的amy與bob變數 — 它們是分別依據Customer類別與Student類別所宣告的變數 — 以物件導向的術語來說，它們就是Customer類別與Student類別的物件。

> (灑花)物件終於登場了
>
> 也該登場了吧。

接下來，就是物件導向和功能為導向程式設計最大的差異之處：我們不是針對計算Amy的銀行帳戶餘額利息，以及判斷Bob的修課成績是否及格等功能，將其設計為函式並使用相關的資料內容做為引數進行呼叫；而是針對已經定義的Customer與Student類別去思考，它們各自應該要提供哪些功能給其物件(也就是依其型態所宣告的變數)使用？然後將這些功能定義在類別裡，專門供其物件呼叫使用。例如Customer類別的“ 物件 ”應該有計算銀行帳戶餘額利息的需求，而Student類別的“ 物件 ”則應該有判斷是否及格的功能。現在，讓我們分別將這些功能加入到相關的類別裡面：

```
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);
   }
};
```

這些與特定類別的“ 物件 ”相關的功能，最終還是被設計為“ 函式 ”了，不過和以前不一樣，這些函式現在分別是Customer與Student類別的函式，所以它們將“ 專供 ”自己使用 — 也就是只有該類別的“ 物件 ”(也就是宣告為該類別的“ 變數 ”)才能夠加以呼叫：

```
Customer amy;
Studnet  bob;
...
cout << amy.interest(0.01) << endl;
if(bob.isPass())
   cout << "Bob Pass" << endl;
```

如果你試圖讓amy呼叫isPass()或是讓bob呼叫interest()，現在就會由編譯器去阻止你，以下的錯誤訊息將會映在你眼底：

```
error: no member named 'interest' in 'Student'
    bob.interest();
    ~~~ ^
error: no member named 'isPass' in 'Customer'
    amy.isPass();
    ~~~ ^
```

至此，我們可以發現C++語言的類別(與結構體2.0)，可以為“ 物件 ”定義相關的資料項目以及專供它使用的函式，解決了以往功能導向程式設計所可能帶來的相關問題。

15.2 物件導向思維

到目前為止，我們已經從功能導向的優缺點一路談到結構體1.0、結構體2.0及類別；在範例中的客戶與學生，也從原本稱呼為“ 對象 ”轉變為“ 物件 ”。但更重要的是，我們已經為讀者說明了C++語言是如何使用類別(Class)是如何解決功能導向程式設計的缺點，但是筆者要強調的是儘管我們在最陰暗處看到了希望的曙光，但這道光(**ひかり**)對我們的意義並非如此而已「(**ひかり**也是華燈初上的主要劇情場景之一。)」，它不只照亮了原本陰暗無光的功能導向世界，它更為我們帶來了通往美麗新世界的光明大道！這個美麗的新世界，就是物件導向的世界！換句話說，儘管類別救贖了功能導向世界的人們，但人類終極的希望卻是在光的盡頭、彼岸的那一端 — 一個充滿物件的美麗新世界！

說得更誇張一些：

> “Space, the final frontier. These are the continuing voyages of the starship Enterprise. Her on-going mission: To explore strange new worlds. To seek out new life forms and new civilizations. To boldly go where no man has gone before.” — James Tiberius Kirk, Captain of the USS Enterprise.
>
> (過期的)翻譯吐司 –>“ 物件導向，最後的邊疆。這是初學者持續的程式設計學習旅程。你正在進行的任務，是去探索這神奇的物件新世界，找尋新的程式設計思維方式和技能，勇踏前人未至之境。 ”

> 好吧，我承認，我是星艦迷。以下恢復正常…

在開始學習物件導向的美麗新世界前，要提醒讀者注意的是，上一節的目的其實並不只是要說明類別可以解決功能導向程式設計方法的不足而已，它其實是要告訴讀者們過去所習慣和熟悉的程式設計方法儘管還是可以解決問題、滿足使用者的需求，但隨著程式的複雜性與困難性的提升，過去的方法已經遇到了一些問題與瓶頸；而物件導向的存在不只是為了解決以往的問題，更重要的是它代表了一種全新的觀念與思維方式。對於程式設計師來說，思維的方式將會決定一切：決定我們如何看待應用程式、如何看待程式所代表的物件導向世界、如何看待物件與物件、物件與使用者間的互動等。要學習物件導向程式設計，就是要學習如何用物件導向的思維方法來構思、設計與實作程式。

就像人們在做任何事情時，腦海裡或多或少都有一些習慣性的思維方式，而這些思維的方法就會影響人們的決策與行為。但不要以為思維是自由、不受限的!? 其實思維會受限於我們所使用的工具與環境。試想在鑽木取火時代的原始人類，當他們好不容易狩獵到野豬(或恐龍… 嗯時代好像兜不上)回到部落時，腦海裡浮現的可能只有升火來做原味炭烤，不然就是給它放著風吹日曬成肉乾；再怎麼摩登的原始人，也無法想像有冰箱可以保存食物、有瓦斯爐、烤箱、微波爐、電鍋、大同電鍋、蒸籠等各種器具，還有油、鹽、醬、醋、糖以及各種現代的食材可以料理的時代。所以在原始人的腦海裡永遠不會有“ 把-10度C低溫保存的豬肋排從冰箱拿出來微波解凍後，用烤箱200度高溫烘焙30分鐘 ”的料裡方法吧！

物件導向其實就是一種思維的方式，讀者們除了要能夠掌握、理解以物件為核心的思考方式外，還必須學習如何使用支援物件導向的程式語言，來將概念轉化為實作的程式碼。

所以在本章接下來的部份，將繼續為讀者說明物件導向的概念與思維的方式，待大家有了基本的理解後，才能再後續的章節裡介紹C++語言物件導向相關的語法與程式碼範例。記住，思維方式與做為工具的程式語言是相輔相成的，正確的觀念可以讓我們更容易學會與掌握工具的使用；累積足夠的工具使用經驗，也可以深化觀念成為在腦海中根深蒂固無法抹去的一顆“ 種子 ”。要知道：

> “The seed that we planted in this man's mind may change everything.” — Inception(2010).

(過期的)翻譯吐司 – >植入程式設計師腦海中的想法，足以改變一切。

上一節所介紹的“ 功能導向 ”其實就是一種以“ 功能 ”為主的思考方式，只不過我們並沒有學習過這種思考的方式，而是在學習程式語言與撰寫程式的過程中無形地培養出來的。對程式設計師(特別是初學者)來說，我們的思考很大幅度地必須受限於程式語言的特性，所以慣用結構性程式語言的人(例如本書的初學讀者，在本書第十四章以前所接觸的內容大多是承襲自C語言，可算是屬於結構化程式語言的部份)，很自然地就會朝向使用循序、選擇與重複等控制結構來構思程式的功能該如何實現。對於已經不知不覺在腦海裡植入的以“ 功能 ”為主的讀者而言，要改變既有的思考方式為以物件為主，其實是非常困難的；就好比要教會原始人類在現代化的廚房料理三餐一樣，我們必須先多花點時間帶領他們去適應現代化廚房的用品、器具、調味料以及各種現代的食材，待他們掌握、理解了不同器具的功效與作用、不同調味品與食材的味道後，現代化的美食料理才會出現在原始人類的餐桌上。

不過很重要的一點是，請別忘了就算原始人類不能理解現代化廚房的奧密，他們還是能夠在現代化的廚房裡鑽木取火吃原味炭烤！有些讀者可能會遭遇到無法使用物件導向的思考方式來使用C++語言，因而陷入了用C++語言寫C語言程式的窘境。所以千萬要好好學習物件導向的觀念，若是遇到不理解的地方可以透過實際撰寫程式來交互印證。

15.3 抽象化

當我們說物件導向的思考是以“ 物件 ”為核心，而不是以“ 功能 ”為核心的時候，並不代表使用物件導向所開發的應用程式就沒有了功能！事實上，應用程式當然必須提供功能去滿足它的使用者，否則就失去了存在的意義。使用物件導向技術所開發的應用程式，當然還是要提供“ 功能 ”給使用者，只不過其實現的方法和過去不同而已。下面這段敘述簡略說明了如何開發物件導向的應用程式，以及它是如何執行以完成使用者所需的功能：

> 物件導向的應用程式，就是以物件為思考的核心，所設計建構的一個抽象的、由物件組成的物件導向世界；當應用程式被執行時，就像是啟動了這個物件導向世界的運轉，透過在執行過程中物件屬性的改變、行為的執行以及物件與物件、物件與使用者的互動等，來實現使用者所需要的各項功能。

上面這段敘述的第一個重點是：物件導向應用程式的開發就是要設計與建構一個抽象的、由物件組成的物件導向的世界。

> (疑惑)但是，“ 物件 ”到底是什麼呢？
>
> 物件(Object)就是“ 東西 ”！
>
> (不解)東西又是什麼？
>
> 什麼東西都是東西！
>
> (完全不明白)什麼是什麼東西的東西？
>
> “ 東西 ”是一種廣泛的代稱，一台車子是一個東西、一本書是一個東西、一件衣服也是東西、一個人也可以算是東西… 不管你搞得清楚還是搞不清楚的東西都是東西！
>
> 而“ 物件 ”就是“ 東西 ”的文雅版同義詞，或者用台語來講東西就是“ 物件(mi̍h-kiānn)“□所以什麼 ”東西 ”都是“ 物件 ”！

從物件導向的觀點來看，大到整個宇宙、小到一顆塵埃，萬事萬物都是物件！不過，你不需要“ 全部 ”都放進你的物件導向世界裡，而是只要挑選與應用程式相關的物件放入世界裡 — 此處物件的相關性通常取

決於應用程式的目的。這種挑選相關物件的過程，就叫做「抽象化」！

抽象化(Abstration)是物件導向四大特性之一[2)]，指得是將複雜的事物依據特定目的簡化為精簡版本的過程。在物件導向應用程式的開發過程裡，抽象化的工作範圍涵蓋很廣，包含在初始構思階段時的兩項重要工作:決定要將哪些物件放入到物件導向的世界裡？以及每個物件的內容為何？

抽象化之一:決定要將哪些物件放入物件導向世界

我們可將在物件導向世界裡的每個物件可視為在真實世界裡有形或無形的人、事、時、地、物的抽象對映。具體來說：

- 有形的(Tangible)物件是指具有實體的、看得到、模得到的，例如學生、教師、銀行客戶、員工、商品、書籍與房子等；
- 無形的(Intangible)物件則是指虛擬的、看不到、模不到的，例如系所、班級、課程、學期、銀行帳戶與家庭等。

通常在設計應用程式時，我們會先將真實世界裡與應用程式相關的物件逐一加以考慮，經檢視其與應用程式目的的相關性後，最終只需要部份物件即可。

舉例來說，當我們為一個成績處理應用程式設計其物件導向世界時，首先考慮的就是在真實世界裡與應用程式相關的物件，例如學校、系所、班級、課程、老師與學生等都是與成績處理相關的物件，但當我們再進一步考慮到該應用程式的目的為儲存、列印學生的成績資訊，並針對及格與否進行示警，所以在此物件導向世界裡只需要保留學生的物件，而不必包含學校、系所、班級、課程與老師等物件；可是當應用程式的目的包含輸出每位老師所開授課程的不及格學生名單時，那麼包含學生、老師、課程等物件都應該要存在於物件導向世界裡。

抽象化之二:決定物件的內容

當一個在真實世界中的有形或無形物件被加入到物件導向的世界後，我們還要進一步決定它的內容 — 將物件抽象化表達為屬性、行為與關係：

- 屬性(Attribute)：與物件相關的資料項目或狀態，在C++裡是以變數的方式存在，例如學生物件具有學號、姓名、性別、學籍狀態、就讀班級、成績、聯絡電話、住址、Email信箱等屬性。
- 行為(Behavior)：物件可以進行的工作、運算或資料處理等，在C++裡是以函式方式存在，例如學生物件具有查詢是否及格、加選課程、退選課程等行為。
- 關係(Relationship)：物件與物件間也可以存在特定的關係，例如一個學生物件是歸屬於特定的系所與班級物件、一個班級物件可以擁有多個學生物件、一個課程物件可以有多個選修的學生物件、一個學生物件可以選修多們課程物件等。在C++裡可以在物件裡宣告變數、陣列或是使用特定的資料結構(例如vector等)加以表示。

因為在真實的世界裡，一個物件可能具備許多的屬性、行為與關係，但我們不必全部保留在物件裡，而是再一次的依據與應用程式的目的相關性進行篩選，只保留部份即可。例如在真實的世界裡，一個學生具有包含姓名、學號、身份證字號、系所、班級、成績、性別、地址、電話、血型、身高、體重、生日、偏好的食物、葷食或素食、宗教信仰與疫苗注射記錄(其實還可以繼續寫下去…)等屬性，以及具有查詢成績、參加社團活動、設備借用、申請請假、上課睡覺、下課睡覺、回家整晚熬夜打電玩等許多行為；但是若考慮到成績處理應用程式的相關性，最終可能只需要保留姓名、學號與成績等屬興，以及查詢成績的行為即可。要記住，物件所應該擁有的屬性與行為是依據應用系統的要求而決定的，在不同的應用系統裡可能存在相同或類似的物件，但其屬性和行為可能完全不同。舉例來說，同樣是做為學生的物件，在以下不同應用目的裡，它們的屬性與行為將會有所不同：

	成績處理	健康履歷	圖書館借還書	請假	選課
屬性	學號 姓名 平常成績 期中考成績 期末考成績	學號 姓名 性別 健保卡卡號 身高 體重 血型 傷病記錄	學號 姓名 已借閱書籍 逾期書籍 罰款	學號 姓名 病假數 公假數 曠課數 是否扣考	學號 姓名 系所 班級 學分數上限 已選課程
行為	查詢作業成績() 查詢期中考成績() 查詢期末考成績() 查詢學期總成績()	新增傷病記錄() 查詢傷病記錄() 預約駐校醫師門診()	預約圖書() 取消預約() 查詢已借圖書() 續借圖書()	請假申請() 取消申請() 修改申請() 查詢已核準申請() 查詢未核準申請()	預選登記() 查詢預選結果() 選課登記() 查詢選課順序() 查詢選課人數() 列印已選課清單()

除了物件本身的屬性與行為外，我們還要思考物件與物件間可能存在的各種關係，例如在成績處理方面，學生與學生間具有選修同一門課程的同學關係，以及學生與課程間具有一對多的選修關係；在健康履歷方面，則可能存在的校醫與學生的多對多醫病關係；選課方面，則要考慮不同系所、不同年級學生，與特定課程間的限制性關係，例如大一同學必選修課程、非資訊學院學生必修通識程式設計課程等。這些關係也會影響到程式各項功能的實作，通常都必須在構思階段就應該加以釐清。

事實上，抽象化所涵蓋的工作還不只限於篩選物件以及物件的屬性、行為與關係 — 這些只是在初始的構思階段的工作，未來我們還會繼續看到需要進行抽象化的地方，屆時我們會再加以說明。當我們完成對物件導向世界的構思後，下一步就是要想辦法將這個抽象的物件導向世界建構出來，才能在後續啟動這個世界的運轉(也就是執行應用程式)，以實現應用程式所需要的各式功能。而為了要建構出這個充滿物件的世界，前一節所討論的類別就是關鍵 — 因為我們必須依據類別來產生程式所需要的物件。

因此，在接下來的章節裡，我們將說明如何定義類別?如何使用類別來產生物件？物件到底是什麼東西？如何使用物件來完成程式所需的各式功能?

---

[1] 此處的Function就是功能的意思，並不是數學的函數、也不是程式設計的函式。
[2] 除抽象化外，剩下三個分別是封裝、繼承與多型，我們將在後續章節中陸續加以介紹