

0

讀者須知

$$e^{i\pi} + 1$$

— Leonhard Euler

本章包含大量資訊，旨在讓你大致了解本書內容。請粗略瀏覽並閱讀你感興趣的地方。在編寫任何程式碼之前，請閱讀「PPP 支援」 (§0.4)。教師會發現大部分內容都很有用。如果你是初學者，請不用試著理解所有的東西。對於小程式的編寫和執行感到得心應手後，你可能會想要回頭重讀本章。

- 0.1 本書結構
整體策略；基本練習、進階練習等；本書之後的學習計畫
- 0.2 教學理念
學生須知；教師須知
- 0.3 ISO 標準的 C++
可移植性 (Portability)；保證；C++ 簡史
- 0.4 PPP 支援
Web 資源
- 0.5 作者簡介
- 0.6 參考書目

本書採單色印刷，書中範例若提及執行結果中有顏色顯示，印刷上將無法直接呈現。請讀者以實際執行程式時所呈現的顏色為準。

書中範例程式與練習題旨在強化學習與實作能力，並未提供程式碼與習題解答。

2.1 輸入

「Hello, World!」程式單純只是寫入螢幕。它產生輸出，不會讀取任何內容，也不會從使用者那裡取得輸入。那太無聊了。真正的程式往往會根據我們提供的輸入產生結果，而不是每次執行都做完全相同的事情。

CC 要讀取某個東西，我們需要讀入的地方；也就是說，我們需要在電腦記憶體（memory）中的某個地方放置我們所讀取的內容。我們稱這種「地方（place）」為一個物件。物件（*object*）是記憶體中的一個區域，其型別（*type*）指定了可以在其中放入何種資訊。具名的物件（*named object*）稱為變數（*variable*）。舉例來說，字元字串（character strings）被放入 **string** 變數，而整數（integers）被放入 **int** 變數。你可以把一個物件看成一個「盒子（box）」，在裡面可以放置具有該物件型別的一個值（*value*）：

	int:
age:	42

這表示名為 **age** 的 **int** 型別物件，其中包含整數值 **42**。使用一個字串變數，我們可以從輸入讀取一個字串，然後再像這樣將其寫出：

```
// 讀寫一個名字 (first name)
#include "PPP.h"

int main()
{
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name;           // first_name 是型別為 string 的一個變數
    cin >> first_name;           // 把字元讀入 first_name
    cout << "Hello, " << first_name << "!\n";
}
```

#include 和 **main()** 在第 1 章中已為大家所熟悉。由於我們所有的程式都需要 **#include** 或等效的直接使用 **import**，因此我們在介紹時不將之呈現出來，以免分散注意力。同樣地，我們有時會介紹一些只有放在 **main()** 或其他函式中才能執行的程式碼，比如下面這樣的程式碼：

```
cout << "Please enter your first name (followed by 'enter'):\n";
```

我們假設你能找出辦法將這些程式碼放入完整的程式中以進行測試。

`main()` 的第一行只是寫出一條訊息 (message)，鼓勵使用者輸入名字。這樣的訊息通常稱為提示 (*prompt*)，因為它提示使用者採取某種行動。接下來的幾行定義了 `string` 型別的變數 `first_name`，將源自鍵盤的輸入讀入該變數，並寫出問候語。讓我們依次檢視這三行：

```
string first_name;    // first_name 是型別為 string 的一個變數
```

這將為字元字串的存放預留一個記憶體區域，並將其命名為 `first_name`：

```
string:
first_name: 
```

在程式中引入新名稱並為變數預留記憶體的述句稱為定義 (*definition*)。

下一行從輸入 (鍵盤) 讀取那些字元到該變數中：

```
cin >> first_name;    // 把字元讀入 first_name
```

`cin` 這個名稱指的是標準程式庫中定義的標準輸入串流 (standard input stream，讀作「see-in」，表示「character input」)。`>>` 運算子 (「取自」) 的第二個運算元指定了那個輸入要往哪去。所以，如果我們鍵入某個名字，比如 `Nicholas`，然後接著一個換行 (newline)，字串 "Nicholas" 就會成為 `first_name` 的值：

```
string:
first_name: 
```

那個換行是引起機器注意的必要條件。在輸入一個換行 (按下 `Enter` 鍵) 之前，電腦單純只會收集字元。這種「延遲」使你有機會改變心意，刪除一些字元並用其他字元代替，然後再按下 `Enter` 鍵。那個換行不會成為儲存在記憶體中的字串的一部分。 **AA**

將輸入字串放入 `first_name` 後，我們就可以使用它了：

```
cout << "Hello, " << first_name << "!\n";
```

這將在螢幕上印出 `Hello`，後面接著 `Nicholas` (`first_name` 的值)，然後是 `!` 和一個換行字元 (`'\n'`)：

```
Hello, Nicholas!
```

如果我們是熱愛重複和打字的人，可以改為寫出三個單獨的述句：

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

然而，我們對打字並不熱衷。更重要的是，我們非常不喜歡無謂的重複（因為重複提供了出錯的機會），因此我們將那三個輸出運算合併為單一個述句。

請注意，在 "Hello, " 中，我們用引號圍住那些字元，但對於 `first_name` 並沒有那樣做。需要字面值字串（`literal string`）時，我們會使用引號。不使用引號時，我們用名稱來指涉某個東西的值。請考慮：

```
cout << "first_name" << " is " << first_name;
```

在此，`"first_name"` 給出十個字元 `first_name`，而單純的 `first_name` 則給出變數 `first_name` 的值，在本例中就是 `Nicholas`。因此，我們會得到

```
first_name is Nicholas
```

2.2 變數

CC 基本上，如果不在記憶體中儲存資料，我們就無法使用電腦做任何有意義的事情，就像我們上面的例子中處理輸入字串一樣。我們儲存資料的「地方」稱為物件。要存取物件，我們需要名稱（*name*）。具名的物件稱為變數（*variable*），它有特定的型別（*type*，如 `int` 或 `string`），型別決定了哪些資料可以放入物件中（例如，`123` 可以放入一個 `int` 中，而 `"Hello, World!\n"` 可以放入一個 `string` 中），以及可以套用哪些運算（例如，我們可以使用 `*` 運算子對 `int` 進行乘法運算，並使用 `<=` 運算子比較 `string`）。我們放入變數的資料項目稱為值（*values*）。定義變數的述句（毫不意外地）被稱為定義（*definition*），而定義可以（通常也應該）提供初始值（*initial value*）。請考慮：

```
string name = "Annemarie";
int number_of_steps = 39;
```

`{=}` 後面的值稱為初始器（*initializer*）。

你可以像這樣視覺化這些變數：

	int:		string:
number_of_steps:	39	name:	Annemarie

你不能把錯誤型別的值放入變數中：

```
string name2 = 39;           // 錯誤：39 不是 string
int number_of_steps = "Annemarie"; // 錯誤："Annemarie" 不是 int
```

編譯器會記住每個變數的型別，並確保你依據變數定義中指定的型別使用它。

C++ 提供相當多的型別。你可以在 Web 上（如 cppreference.com）找到完整的清單。不過，你只需使用其中的五種型別就能編寫出非常出色的程式：

```
int number_of_steps = 39;      // 代表整數的 int
double flying_time = 3.5;     // 代表浮點數的 double
char decimal_point = '.';     // 代表個別字元的 char
string name = "Annemarie";    // 代表字元字串的 string
bool tap_on = true;          // 代表邏輯變數的 bool
```

`double` 這個名稱的由來是有歷史淵源的：`double` 是「double-precision floating point（雙精度浮點數）」的簡稱。浮點數是電腦對實數（real number）這一數學概念的近似表示法。

請注意，每種型別都有自己獨特的字面值（literals）形式：

```
39           // int：一個整數
3.5          // double：一個浮點數
'.'          // char：圈在單引號中的一個單獨的字元
"Annemarie"  // string：用雙引號界定的一個字元序列
true         // bool：要不是 true，就是 false
```

也就是說，一個數字序列（a sequence of digits，如 1234、2 或 976）表示一個整數；單引號中的單一字元（如 '1'、'@' 或 'x'）表示一個字元；帶小數點的一個數字序列（如 1.234、0.12 或 .98）表示一個浮點數，雙引號中的一個字元序列（如 "1234"、"Howdy!" 或 "Annemarie"）則表示一個字串。

2.3 輸入與型別

輸入運算 `>>`（「取自」）對型別很敏感，也就是說，它會根據所讀入的變數之 **CC** 型別進行讀取。例如：

```
int main()      // 讀取名字和年齡
{
    cout << "Please enter your first name and age\n";
    string first_name = "???";    // string 變數 ("???" 代表「不知道名字」)
    int age = -1;                 // 整數變數 (-1 代表「不知道年齡」)
    cin >> first_name >> age;     // 讀入一個字串，後面接著一個整數
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

因此，如果輸入 `Carlos 22`，`>>` 運算子會將 `Carlos` 讀入 `first_name`，將 `22` 讀入 `age`，並產生以下輸出：

```
Hello, Carlos (age 22)
```

為什麼它不會把 `Carlos 22`（全都）讀入 `first_name`？因為依照慣例，`string` 的讀取以所謂的空白（*whitespace*，即空格、換行和 `tab` 字元）結束。否則，`>>` 預設會忽略空白。舉例來說，你可以在要讀取的數字前加上任意多的空格（`spaces`），`>>` 單純只會跳過它們，並讀取數字。

正如我們可以在單一輸出述句中寫出多個值一樣，我們也可以在單一輸入述句中讀取多個值。注意 `<<` 和 `>>` 一樣對型別敏感，所以我們可以輸出 `int` 變數 `age` 以及 `string` 變數 `first_name` 和字串字面值 `"Hello, "、" (age "` 和 `")\n"`。

如果你輸入 `22 Carlos`，你會看到一些在你深入思考之前可能會感到驚訝的內容。（錯誤的）輸入 `22 Carlos` 將輸出

```
Hello, 22 (age 0)
```

`22` 將被讀入 `first_name`，畢竟 `22` 是一個字元序列，而且以空白結束。另一方面，`Carlos` 不是整數，所以不會被讀取。輸出結果將是 `22` 和 `age` 的初始值 `0`。為什麼會這樣？因為你沒有成功讀入任何一個值，所以理想上它會保留其初始值，但事實並非如此。在 §4.6.3 和 §9.4 中，我們會看到如何正確測試輸入錯誤和處理不良輸入。至於現在，我們只需注意必須時時提防不良輸入。

AA 使用 `>>` 讀取的一個 `string`（預設情況下）以空白結束；也就是說，它讀取的是單一個詞（`word`）。但有時我們想讀取不只一個詞。當然，有很多方法可以做到這一點。舉例來說，我們可以像這樣讀取由兩個詞構成的一個名稱：

```
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // 讀取兩個字串
    cout << "Hello, " << first << " " << second << '\n';
}
```

我們只需使用兩次 `>>`，每個名稱使用一次。當我們要將名稱寫至輸出時，必須在它們之間插入一個空格。

請注意，作為輸入目標的兩個 `string`（`first` 和 `second`）沒有初始器。預設情況下，一個 `string` 會初始化為空字串，也就是 `""`。

小試身手

執行這個「名字與年齡」範例。然後對其進行修改，寫出以月數為單位的年齡：讀取以年為單位的輸入值，然後乘以 12（使用 * 運算子）。將年齡讀到一個 double，讓孩子們能為自己是五歲半而非只有五歲而感到自豪。

2.4 運算和運算子

除了規定變數中可以儲存哪些值以外，變數的型別還決定了我們可以對其進行什麼運算以及那些運算的含義。例如：

```
int age = -1;
cin >> age;                                // >> 將一個整數讀入 age

string name;
cin >> name;                                // >> 將一個字串讀入 name

int a2 = age+2;                               // + 整數相加
string n2 = name + " Jr. ";                   // + 串接 (concatenates) 字串

int a3 = age-2;                               // - 整數相減
string n3 = name - " Jr. ";                   // 錯誤：- 對字串而言沒有定義
```

這裡所說的「錯誤」是指編譯器會拒絕試圖進行字串相減的程式。編譯器清楚地知道每個變數可以套用哪些運算，因此可以避免許多錯誤。但是，編譯器並不知道在哪些值上進行的哪些運算對你而言是合理的，所以它很樂意接受產出結果在你看來很荒謬的合法運算。例如：

```
age = -100;
```

你或許很清楚不可能有負數年齡（為什麼不能呢？），但沒有人告訴編譯器這件事，所以編譯器會為該定義產生程式碼。

這個表為一些常見且有用的型別列出了實用的運算子：

運算	bool	char	int	double	string
指定	=	=	=	=	=
加法			+	+	
串接					+
減法			-	-	
乘法			*	*	

運算	bool	char	int	double	string
除法			/	/	
取餘數（模數）			%		
遞增 1			++	++	
遞減 1			--	--	
遞增 n			+= n	+= n	
新增到尾端					+=
遞減 n			-- n	-- n	
先乘再指定			*=	*=	
先除再指定			/=	/=	
取餘數再指定			%=		
從 s 讀入到 x	s >> x	s >> x	s >> x	s >> x	s >> x
將 x 寫入到 s	s << x	s << x	s << x	s << x	s << x
相等	==	==	==	==	==
不相等	!=	!=	!=	!=	!=
大於	>	>	>	>	>
大於或等於	>=	>=	>=	>=	>=
小於	<	<	<	<	<
小於或等於	<=	<=	<=	<=	<=

空白的地方表示某個運算不能直接用於某個型別（儘管可能有間接使用該運算的方式；請參閱 §2.9）。

我們將繼續解釋這些運算和更多的運算。這裡的要點在於，有很多實用的運算子，而且對於類似的型別，它們的意義往往是相同的。

讓我們舉一個涉及到浮點數的例子：

```
int main()      // 實際使用運算子的簡單程式
{
    cout << "Please enter a floating-point value: ";
    double n = 0;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
```

```
<< '\n';
}
```

顯然，一般算術運算（arithmetic operations）的符號和含義就跟我們在小學時學到的一樣。唯一的例外是，「相等（equal）」的符號是 `==`，而不單純只是 `=`。當然，並不是我們想對浮點數進行的所有運算，如取平方根（square root），都可以透過運算子取用。許多運算都是以具名函式（named functions）來表示。在本例中，我們使用標準程式庫中的 `sqrt()` 函式求 `n` 的平方根：`sqrt(n)`。這種符號在數學中並不陌生。這一路上我們都會使用函式，並會在 §3.5 和 §7.4 中詳細討論它們。

小試身手

執行這個小程式。然後，修改它以讀取一個 `int` 而非一個 `double`。同時，「實際使用」一些其他運算，例如模數（modulo）運算子 `%`。請注意，對於 `int` 來說，`/` 是整數除法，而 `%` 是取餘數（模數），因此 `5/2` 是 2（而不是 2.5 或 3），`5%2` 是 1。整數 `*`、`/` 和 `%` 的定義保證了對於兩個正值的 `int`，如 `a` 和 `b`，我們會有 `a/b * b + a%b == a`。

字串的運算子較少，但有大量的具名運算（請參閱 PPP2.Ch23）。不過，它們所擁有的運算子都可以按常規使用。例如：

```
int main()      // 讀取第一個和第二個名稱
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // 讀取兩個字串

    string name = first + ' ' + second; // 串接字串
    cout << "Hello, " << name << '\n';

}
```

對於字串，`+` 代表串接（concatenation）；也就是說，當 `s1` 和 `s2` 都是字串時，`s1+s2` 會是一個字串，其中來自 `s1` 的字元後面接著來自 `s2` 的字元。舉例來說，如果 `s1` 的值是 "Hello"，而 `s2` 的值是 "World"，那麼 `s1+s2` 的值就是 "HelloWorld"。`string` 的比較（comparison）特別有用：

```
int main()      // 讀取並比較名稱
{
    cout << "Please enter two names\n";
    string first;
```

```

string second;
cin >> first >> second;    // 讀取兩個字串

if (first == second)
    cout << "that's the same name twice\n";
if (first < second)
    cout << first << " is alphabetically before " << second << '\n';
if (first > second)
    cout << first << " is alphabetically after " << second << '\n';
}

```

在此，我們使用一個 `if` 述句（將在 §3.4.1.1 中詳細說明）根據條件選擇運算。

2.5 指定和初始化

CC 從許多方面來說，最有趣的運算子是指定（assignment，或稱「賦值」），用 `=` 表示。它賦予變數新的值。例如：

<code>int a = 3;</code>	<code>// a 一開始的值為 3</code>	a: 3
<code>a = 4;</code>	<code>// a 得到 4 這個值（變成 4）</code>	a: 4
<code>int b = a;</code>	<code>// b 一開始拷貝 a 的值（也就是 4）</code>	a: 4 b: 4
<code>b = a+5;</code>	<code>// b 得到值 a+5（也就是 9）</code>	a: 4 b: 9
<code>a = a+7;</code>	<code>// a 得到值 a+7（也就是 11）</code>	a: 11 b: 9

XX 最後一個指定值得注意。首先，它清楚表明 `=` 並不代表等於（equals）——顯然，`a` 並不等於 `a+7`。它意味著指定（assignment），即在變數中放入新值。`a=a+7` 所做的事如下：

- [1] 首先，取得 `a` 的值，即整數 4。
- [2] 接著，把 7 加到那個 4，產出整數 11。
- [3] 最後，將那個 11 放入 `a`。

我們還可以用字串來闡明指定的運作方式：

<code>string a = "alpha";</code>	<code>// a 一開始有 "alpha" 這個值</code>	a: alpha
<code>a = "beta";</code>	<code>// a 得到值 "beta" (變成 "beta")</code>	a: alpha
<code>string b = a;</code>	<code>// b 一開始拷貝 a 的值 (也就是 "beta")</code>	a: beta
		b: beta
<code>b = a+"gamma";</code>	<code>// b 得到值 a+"gamma" (也就是 "betagamma")</code>	a: beta
		b: betagamma
<code>a = a+"delta";</code>	<code>// a 得到值 a+"delta" (也就是 "betadelta")</code>	a: betadelta
		b: betagamma

我們使用「一開始有 (starts out with)」和「得到 (gets)」來區分兩種相似但邏輯上不同的運算： **CC**

- 初始化 (*Initialization*)：賦予變數初始值。
- 指定 (*Assignment*)：賦予變數新的值。

從邏輯上來說，指定和初始化是不同的。原則上，初始化的動作總是會發現變數是空的。另一方面，指定（原則上）必須先清除變數中的舊值，然後再放入新值。你可以把變數想像成一種小盒子，而值則是放入其中的具體事物，比如一枚硬幣。初始化前，盒子是空的，但初始化後，盒子裡一直都放有一枚硬幣，因此，要放入新的硬幣，你（即指定運算子）必須先移除舊的硬幣（「銷毀舊的值」）。電腦記憶體中實際的情況並不完全是如此，但這也不失為想像其運作方式的一種好辦法。

2.5.1 範例：偵測重複的字詞

當我們要將新的值放入物件時，就需要進行指定。仔細想想，很明顯，多次執行任務時，指定是最有用的。當我們想用不同的值再次做某事時，就需要指定。讓我們來看看一個小程序，它會偵測字詞序列中相鄰的重複字詞。這種程式碼是大多數文法檢查器 (grammar checkers) 的一部分：

```
int main()
{
    string previous;           // 前一個字詞；初始化為 ""
    string current;           // 目前字詞
    while (cin>>current) {    // 讀取一連串字詞
```

```

        if (previous == current)    // 檢查該字詞是否與上一個相同
            cout << "repeated word: " << current << '\n';
        previous = current;
    }
}

```

這個程式並不是特別有用，因為它並沒有告訴我們重複出現的字詞在文字中出現的位置，但就現在而言，這樣就夠了。我們將從以下這行開始，逐行檢視這個程式

```
string current;    // 目前字詞
```

預設情況下，一個 `string` 會被初始化為空字串（empty string），因此我們不必明確地對其進行初始化。我們使用下列程式碼來將一個字詞讀入 `current`：

```
while (cin>>current)
```

AA 這種構造（construct）稱為 `while` 述句，它本身就很有趣，我們將在 §3.4.2.1 中進一步研究它。這個 `while` 述句指出，只要輸入運算 `cin>>current` 成功，`(cin>>current)` 後面的述句就會重複，而只要標準輸入中還有字元可以讀取，`cin>>current` 就會成功。請記住，對於 `string`，`>>` 讀取的是以空白分隔的字詞。給程式一個輸入結束字元（end-of-input character，通常稱為 *end of file*，檔案結尾），就可以終止這個迴圈。在 Windows 機器上，那會是 `Ctrl+Z`（同時按下 `Control` 和 `Z`），然後再按下 `Enter`（`return`）鍵。在 Linux 機器上，則是 `Ctrl+D`（同時按下 `Control` 和 `D`）。

所以，我們要做的就是把一個字詞讀到 `current` 中，然後將其與前一個字詞（儲存在 `previous` 中）進行比較。如果它們相同，我們就這麼說：

```

if (previous == current) // 檢查該字詞是否與上一個相同
    cout << "repeated word: " << current << '\n';

```

然後，我們必須準備好為下一個字詞再次這樣做。為此，我們要將 `current` 拷貝到 `previous` 中：

```
previous = current;
```

只要能夠啟動流程，這就可以處理所有情況。第一個字詞沒有前一個字詞可以比較，我們該怎麼辦呢？`previous` 的定義解決了這個問題：

```
string previous;    // 前一個字詞；初始化為 ""
```

空字串不是字詞。因此，在第一次執行 `while` 述句時，這個測試

```
if (previous == current)
```

會失敗（正如我們想要的那樣。）

了解程式流程的一種方法是「扮演電腦」，也就是說，按照程式逐行執行其指定的動作。只需在紙上畫出方框，並將它們的值寫入其中。依據程式的規定更改所儲存的值。 **AA**

小試身手

用一張紙自己執行這個程式。輸入 `The cat cat jumped`。即使是經驗豐富的程式設計師，也會用這種技巧來視覺化一小段程式碼的動作，以了解它們表面上並不那麼明顯的行為。

小試身手

執行「重複字詞偵測程式」。用 `She she laughed "he he he!" because what he did did not look very very good good` 這個句子測試它。有多少個重複的字詞？為什麼？這裡所用的字詞（*word*）之定義是什麼？重複字詞（*repeated word*）的定義是什麼？（例如，`She she` 是重複的字詞嗎？）

2.5.2 複合指定（composite assignment）運算子

遞增（*incrementing*）一個變數（即向變數加 1）在程式中是如此的常見，所以 C++ 為其提供了一種特殊的語法。舉例來說：

```
++counter
```

這意味著

```
counter = counter + 1
```

根據變數目前的值改變其值的常見方式還有很多。例如，我們可能想把它加上 7、減去 9 或乘以 2。C++ 也直接支援這些運算。舉例來說：

```
a += 7;    // 代表 a = a+7
b -= 9;    // 代表 b = b-9
c *= 2;    // 代表 c = c*2
```

一般來說，對於任何二元運算子（*binary operator*）`oper`，`a oper= b` 表示 `a = a oper b`。首先，這條規則為我們提供運算子 `+=`、`-=`、`*=`、`/=` 與 `%=`。這提供令人愉悅的簡潔符號，直接反映了我們的概念。舉例來說，在許多應用領域，`*=` 和 `/=` 被稱為「縮放（*scaling*）」。

2.5.3 範例：找出重複字詞

請考慮前面偵測相鄰重複字詞的例子。我們可以透過給出重複字詞在序列中的位置來改善它。這個想法的簡單變體就是單純計數到目前為止有幾個字詞，然後輸出重複字詞的編號：

```
int main()
{
    int number_of_words = 0;
    string previous;           // 前一個字詞；初始化為 ""
    string current;
    while (cin>>current) {
        ++number_of_words;     // 遞增字數
        if (previous == current)
            cout << "word number " << number_of_words << " repeated: "
                 << current << '\n';
        previous = current;
    }
}
```

我們的字詞計數器（word counter）從 0 開始，每看到一個字詞，計數器就遞增一次：

```
++number_of_words;
```

這樣，第一個字詞就變成了 1 號，下一個字詞變成了 2 號，以此類推。這樣做也可以達成相同的目的：

```
number_of_words += 1;
```

或甚至是

```
number_of_words = number_of_words+1;
```

但 `++number_of_words` 更簡短，而且直接表達了遞增的概念。

AA 注意到這個程式與 §2.5.1 中的程式有多麼相似。顯然，我們只是把 §2.5.1 中的程式稍加修改，以達到我們的新目的。這是一種非常常見的技巧：當我們需要解決某個問題時，我們會尋找類似的問題，並在適當修改後使用我們對於那個問題的解決方案。除非萬不得已，否則不要從頭開始。使用某個程式的舊版本作為修改的基礎，往往能節省大量時間，而且我們還能從原程式的許多努力中獲益。

10.1 為什麼要做圖形處理？

為什麼我們要花四章來討論圖形（graphics），一章來討論 GUI（graphical user interfaces，圖形使用者介面）呢？畢竟，這是一本關於程式設計的書，而不是一本關於圖形的書。有很多有趣的軟體主題我們都沒有討論，而關於圖形主題我們最多只能觸及表面。那麼，「為什麼要做圖形處理？」。基本上，圖形是可以讓我們探索軟體設計、程式設計和程式語言設施等幾個重要領域的主題：

- 圖形很有用。程式設計遠不止於圖形處理，軟體也遠不限於透過 GUI 操作的程式碼。然而，在許多領域，良好的圖形是不可或缺，或者說非常重要的。舉例來說，如果不具備繪製資料圖表的能力，我們就別想研究科學計算、資料分析或任何定量學科（quantitative subject）。第 13 章介紹繪製資料圖表的簡單（但通用的）設施。此外，也可以想想瀏覽器、遊戲、動畫、科學視覺化、電話和控制顯示器。
- 圖形很有趣。在電腦領域中，很少有領域能讓一段程式碼的效果一目了然，而且在最終沒有 bugs 的情況下，還能讓人賞心悅目。即使沒有用處，我們也會忍不住去把玩圖形！
- 圖形提供許多有趣的程式碼供我們閱讀。學習程式設計的一部分工作就是閱讀大量程式碼，以了解好的程式碼是什麼樣子。同樣地，要成為一名優秀的英語作家，也需要閱讀大量的書籍、文章和高品質的報紙。由於我們在螢幕上看到的內容與我們在程式中編寫的內容直接對應，因此簡單的圖形程式碼比大多數具有類似複雜性的程式碼更具可讀性。本章將證明，經過幾分鐘的介紹，你就能讀懂圖形程式碼；第 11 章將示範如何再花幾個小時就能寫出圖形程式碼。
- 圖形是設計範例的豐沛來源。設計和實作好的圖形和 GUI 程式庫其實是很難的。圖形是設計決策和設計技巧的具體和實務範例非常豐富的來源。設計類別、設計函式、將軟體分層（抽象層）以及建構程式庫的一些最有用的技巧，都可以用相對少量的圖形和 GUI 程式碼來闡明。
- 圖形可以很好地介紹一般所說的物件導向程式設計（*object-oriented programming*）、以及支援物件導向程式設計的語言功能。儘管有反面的傳言存在，物件導向程式設計並不是為了做圖形處理而發明的（請參閱 PPP2. §22.2.4），但它很快就被套用到那之上，圖形提供物件導向設計的一些最容易理解和最具體的例子。

- 有些關鍵的圖形概念並不容易理解。因此，它們值得傳授，而不是讓你自己主動（並耐心地）去尋找資訊。如果我們不展示圖形和 GUI 是如何完成的，你可能會認為它們是「魔法」，從而違背了本書的基本宗旨之一。

10.2 一種顯示器模型

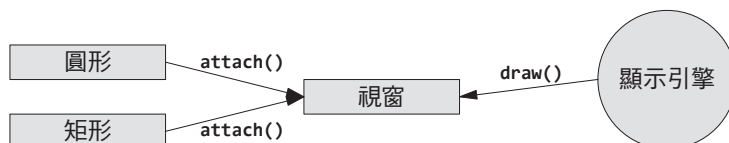
`iostream` 程式庫主要用於讀取和寫入字元串流，就像讀寫數值串列或書籍中的文字一樣。唯一直接支援圖形位置概念的是換行（`newline`）和 `tab` 字元。你可以在一維（`one-dimensional`）字元串流中嵌入顏色和 2D（`two-dimensional`，二維）位置等概念。這就是 Troff、TeX、Word、Markup、HTML 和 XML（及其相關圖形套件）等佈局（排版、「標示」）語言所做的工作。例如：

```
<hr>
<h2>
Organization
</h2>
This list is organized in three parts:
<ul>
    <li><b>Proposals</b>, numbered EPddd, ...</li>
    <li><b>Issues</b>, numbered EIddd, ...</li>
    <li><b>Suggestions</b>, numbered ESddd, ...</li>
</ul>
<p>We try to ...
<p>
```

這是一段 HTML，其中指定了一個標頭（`<h2> ... </h2>`）、包含串列項目（`<li> ... </li>`）的一個串列（`<ul> ... </ul>`）和一個段落（`<p>`）。我們省略了大部分實際文字，因為在此它們無關緊要。重點是，你可以用純文字表達佈局（`layout`）概念，但所寫的字元與螢幕上顯示的內容之間的關聯是間接的，由解讀這些「標記（`markup`）」命令的程式所控制。這種技巧從根本上說很簡單，而且非常有用（你讀到的幾乎所有內容都是用這種技巧產生的），但也有其侷限性。

在本章和接下來的四章中，我們將提出另一種選擇：一種直接針對電腦螢幕的圖形和 GUI 概念。其基礎概念本質上就是圖形的（而且是 2D 的，適用於電腦螢幕的矩形區域），如座標、線條、矩形和圓。從程式設計的角度來看，其目的是將記憶體中的物件與螢幕上的影像（`images`）直接對應起來。

CC 基本模型如下：我們用圖形系統提供的基本物件（如線條）組成物件。我們將這些圖形物件「接附（attach）」到代表實體螢幕的視窗物件（window object）上。然後，我們可以將程式視為顯示器（display）本身、作為一種「顯示引擎（display engine）」、作為「我們的圖形程式庫（graphics library）」、作為「GUI 程式庫」，甚至（幽默地）視為「坐在螢幕後面的小地精（gnome）」，它將我們接附到視窗的物件繪製（draw）到螢幕上：



「顯示引擎」在螢幕上繪製線條、在螢幕上放置文字字串、為螢幕區域著色等。為了簡單起見，我們將使用「我們的 GUI 程式庫」或甚至「系統（system）」來代表顯示引擎，儘管我們的 GUI 程式庫所做的遠不止繪製物件。就像我們的程式碼讓 GUI 程式庫為我們完成大部分工作一樣，GUI 程式庫也將其大部分工作委派給了作業系統（operating system）。

10.3 第一個範例

我們的任務就是定義類別，從中我們可以製作出想要在螢幕上看到的物件。舉例來說，我們可能希望將一個圖繪製為一系列相連的線條。這裡有個小程序，展示了這個概念非常簡單的一個版本：

```

#include "Simple_window.h"    // 取用我們的視窗程式庫
#include "Graph.h"           // 取用我們的圖形程式庫設施

int main()
{
    using namespace Graph_lib;    // 我們的繪圖設施在 Graph_lib 中

    Application app;              // 啟動一個 graphics/GUI 應用程式

    Point tl {900,500};          // 成為視窗的左上角

    Simple_window win {tl,600,400,"Canvas"}; // 製作一個簡單的視窗

    Polygon poly;                // 製作一個形狀（一個多邊形）
    poly.add(Point{300,200});    // 新增一個點
    poly.add(Point{350,100});    // 新增另一個點
  
```

```

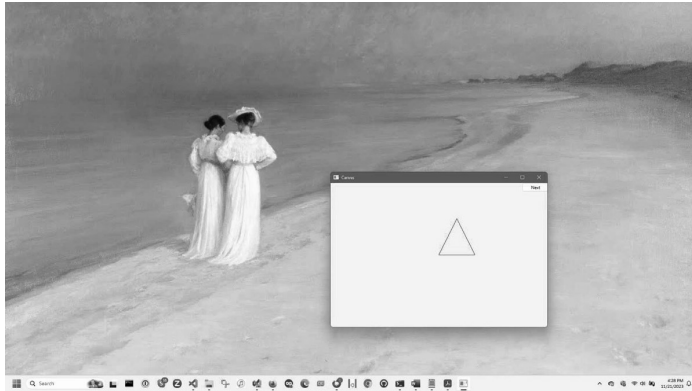
poly.add(Point{400,200});           // 新增第三個點
poly.set_color(Color::red);         // 調整 poly 的特性

win.attach (poly);                  // 將 poly 連接到視窗

win.wait_for_button();              // 把控制權交給顯示引擎
}

```

當我們執行這個程式時，螢幕的畫面看起來會像這樣：



在我們視窗的背景中，可以看到一部筆記型電腦的螢幕（為了本次展示已事先整理過）。若有人對無關緊要的細節感到好奇，我可以告訴你，我的背景圖像是丹麥畫家 Peder Severin Krøyer 的一幅名畫。那兩位女士是 Anna Ancher 和 Marie Krøyer，也都是著名畫家。如果仔細觀察，你會發現我們執行的是 Microsoft C++ 編譯器，但我們也可以使用其他編譯器（如 GCC 或 Clang）。讓我們逐行檢視程式看看它們做了什麼事。

首先，我們 `#include` 圖形介面程式庫：

```

#include "Simple_window.h"           // 取用我們的視窗程式庫
#include "Graph.h"                   // 取用我們的圖形程式庫設施

```

為什麼不使用 `Graph_lib` 模組（§7.7.1）？其中的一個原因是，在撰寫本文之時，並非所有的實作都能使用模組來完成這項相對複雜的任務。舉例來說，我們用來實作圖形程式庫的 Qt 系統使用標頭檔匯出其機能（§7.7.2）。另一個原因是，「現存」的 C++ 程式碼有太多都使用標頭檔（§7.7.2），以致於我們需要在某處展示實際的例子。