C++程式設計入門

1-開發環境

1.1 安裝 Code::Blocks

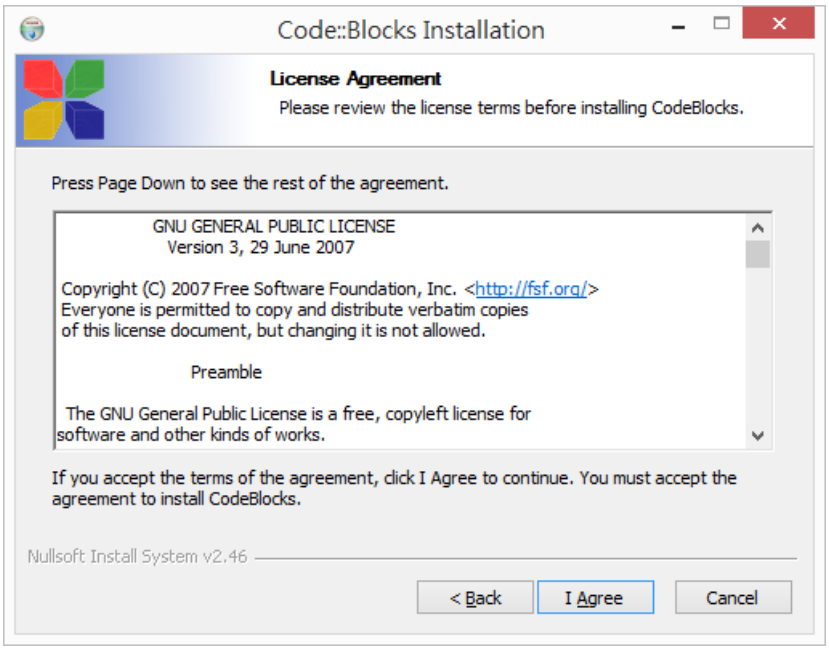
下載

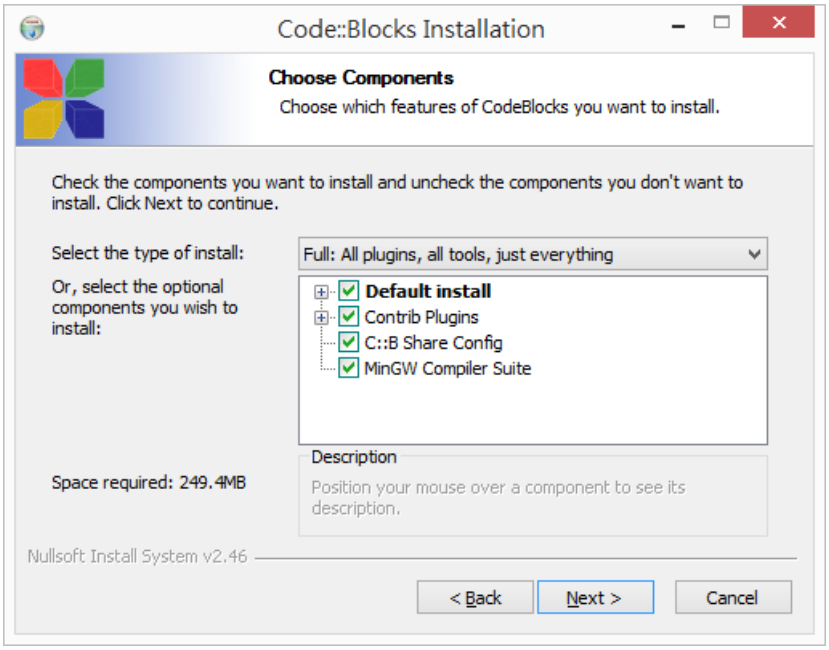
至 https://www.codeblocks.org/ ,依序點選 [Downloads] →[Download the binary release]。

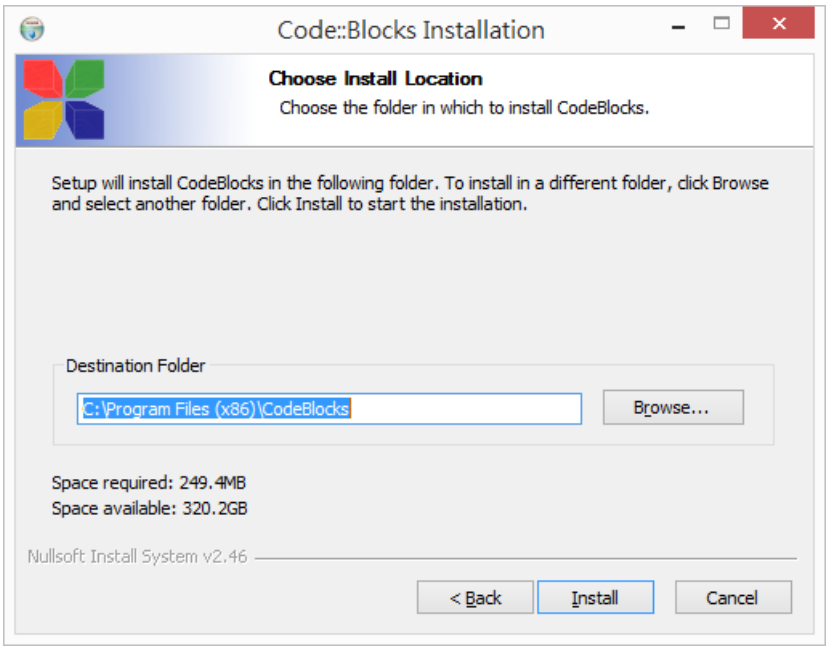
下載 [codeblocks-16.01mingw-setup.exe] (16.01 是版本,請找當時最新版的)

安裝

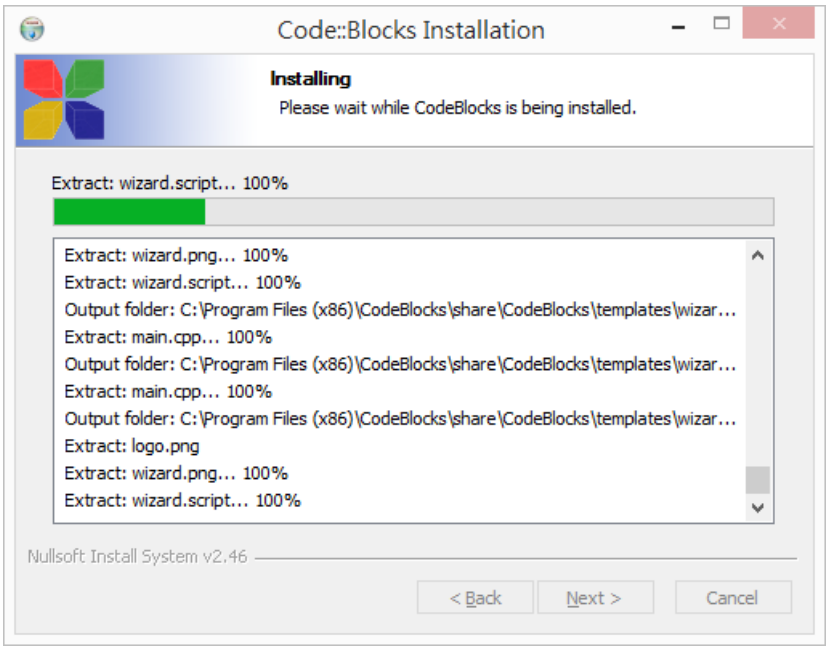
基本上都接受預設值,按 [Next] →[I Agree] →[Next]→[Install] 即可。


Code::Blocks Installation
License Agreement
Please review the license terms before installing CodeBlocks.
Press Page Down to see the rest of the agreement.
GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007
Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
Preamble
The GNU General Public License is a free, copyleft license for
software and other kinds of works.
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install CodeBlocks.
Nullsoft Install System v2.46
< Back
I Agree
Cancel

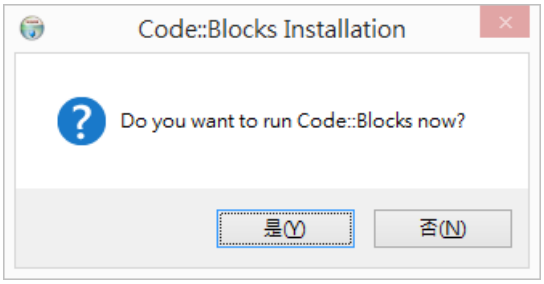

Code::Blocks Installation
Choose Components
Choose which features of CodeBlocks you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select the type of install:
Full: All plugins, all tools, just everything
Or, select the optional components you wish to install:
Default install
Contrib Plugins
C::B Share Config
MinGW Compiler Suite
Description
Position your mouse over a component to see its description.
Space required: 249.4MB
Nullsoft Install System v2.46
< Back
Next >
Cancel


Code::Blocks Installation
Choose Install Location
Choose the folder in which to install CodeBlocks.
Setup will install CodeBlocks in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.
Destination Folder
C:\Program Files (x86)\CodeBlocks
Browse...
Space required: 249.4MB
Space available: 320.2GB
Nullsoft Install System v2.46
< Back
Install
Cancel

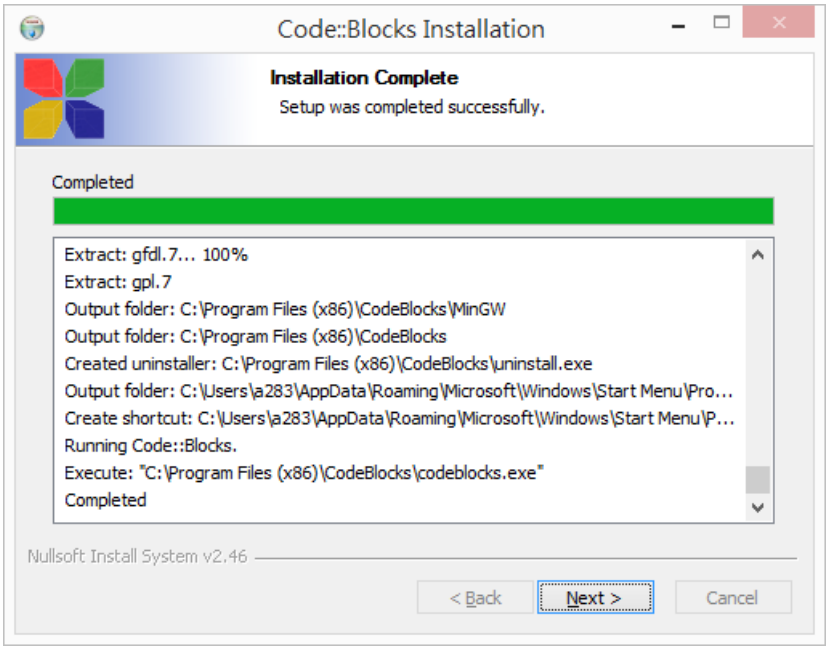
等待安裝結束。

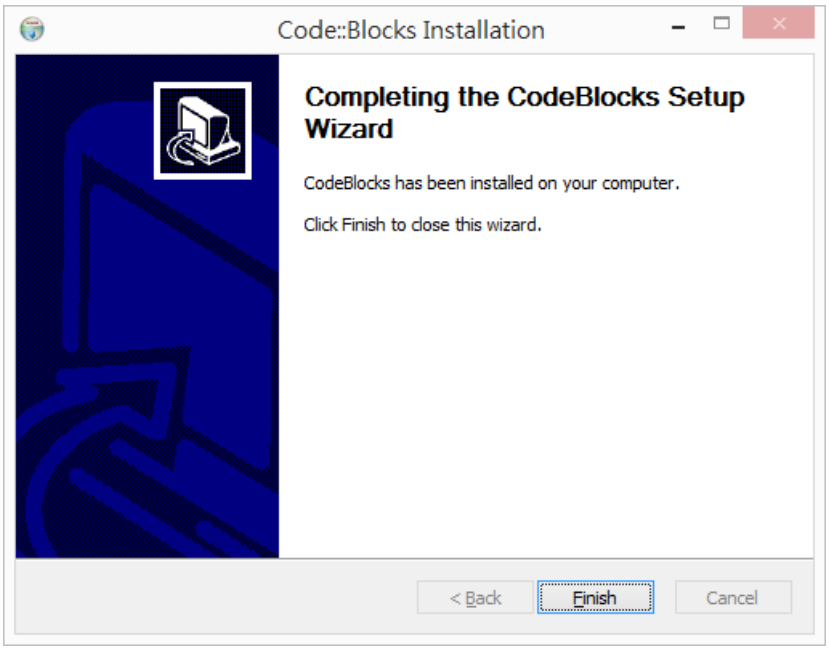


選 [是] 啟動 Code::Blocks。



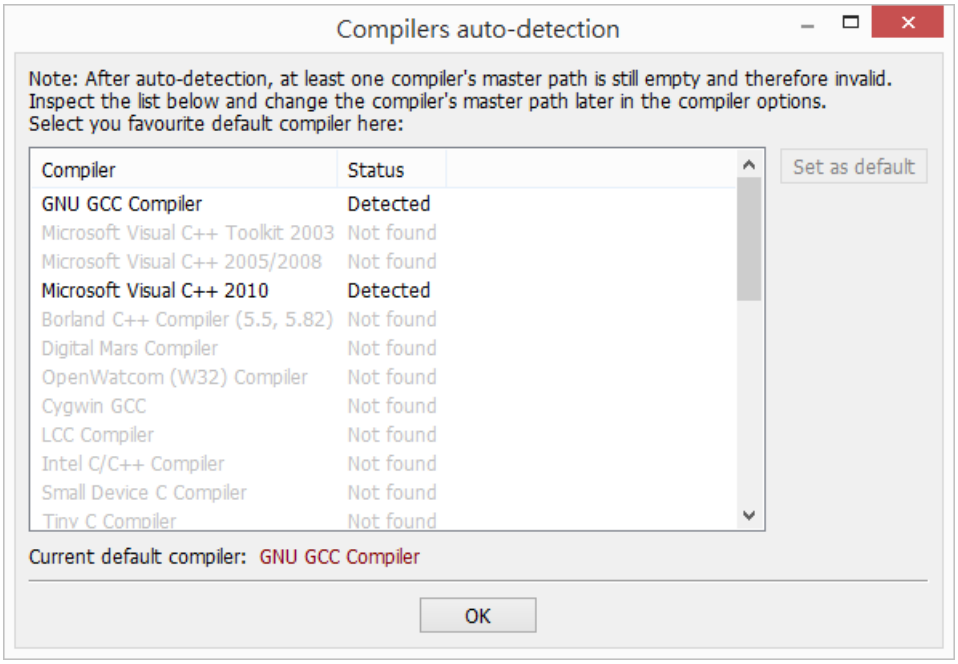
把安裝程式結束 [Next]→[Finish]。





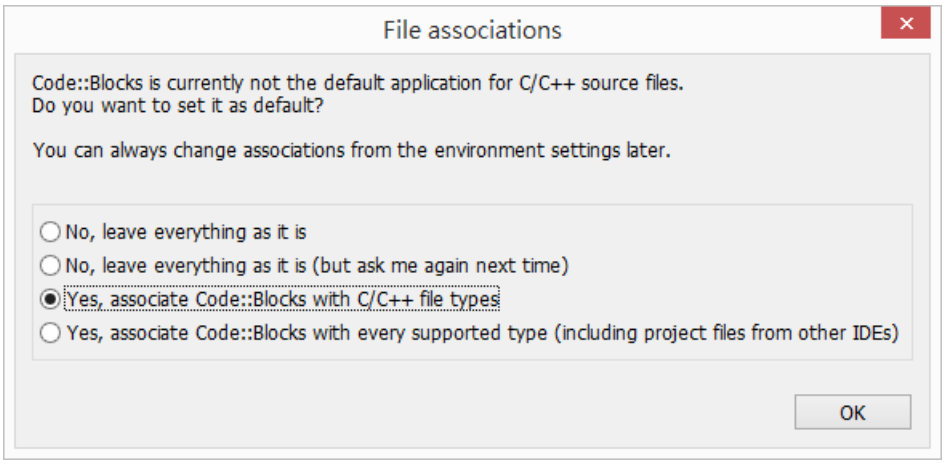
啟動

首次啟動時，會自動尋找電腦上有安裝的編譯器。大部分同學應該都只會看到 GNU GCC Compiler。選擇它，按下 [Set as default]，再按下 [OK] 即可。



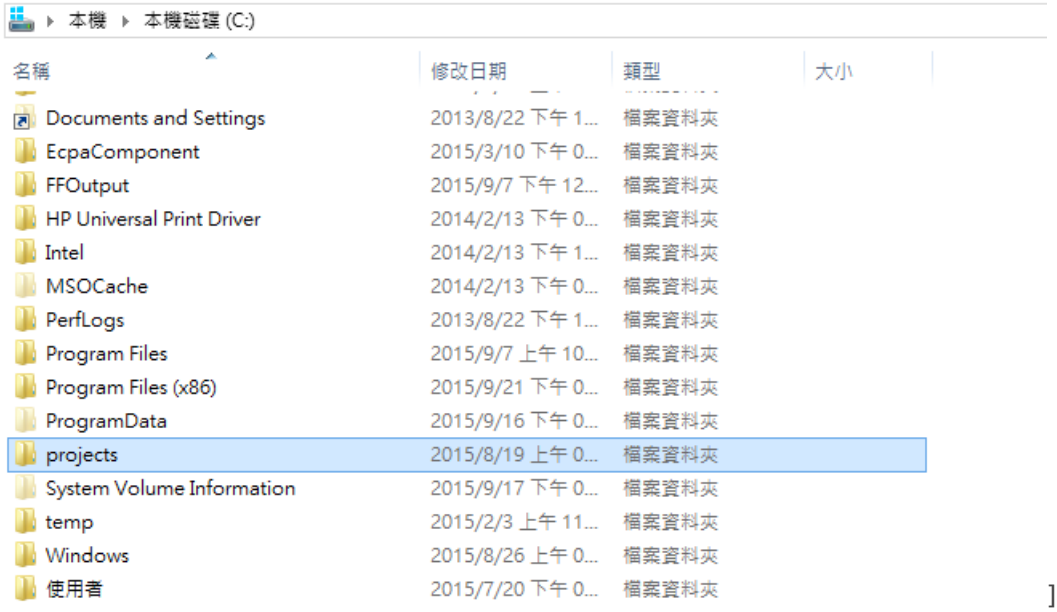
在檔案關聯部分，如果你這台電腦沒有安裝其他 C/C++ 編輯工具的話，可以接受預設值 "Yes, associate Code::Blocks with C/C++ file types"

如果有的話(例如: 父母、兄姐有在這台電腦上寫 C/C++程式)，請先選第一個 "No, leave everything as it is"，以免覆蓋掉檔案的關聯設定。



建立專案資料夾

在磁碟中建立一個專案資料夾，建議為 C:\projects 或 D:\projects。以後我們寫的專案就會放在這個資料夾中。



[ ]
(https://nlmoodle.ddns.net/uploads/images/gallery/2023-05/cb12.png

用 C++ 開發軟體時，我們所謂的 **「專案(project)」** 是什麼意思呢？

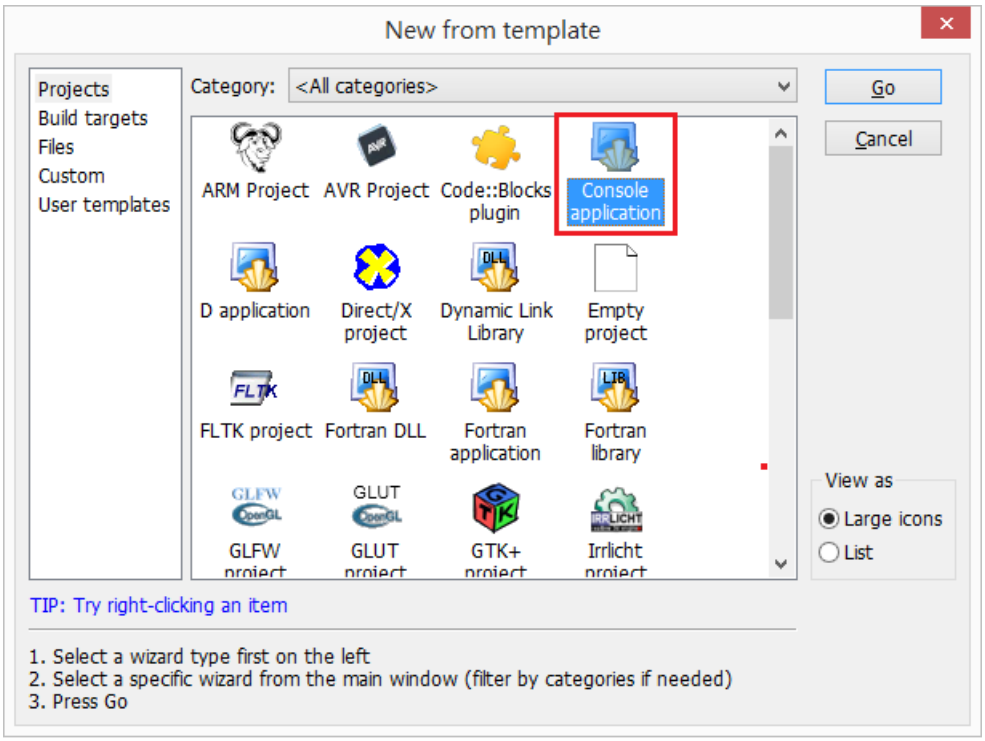
舉例來說，今天我們要開發一個「貪食蛇」遊戲，這個遊戲會有許多的程式碼檔案、圖片檔案、音樂檔、音效檔……等等。

為了好管理，我們會把這一堆檔案依一定規範放在一個資料夾中，這整個資料夾就是我們的專案。所以每次我們要寫一個新程式時，就會建立一個新的專案(資料夾)。

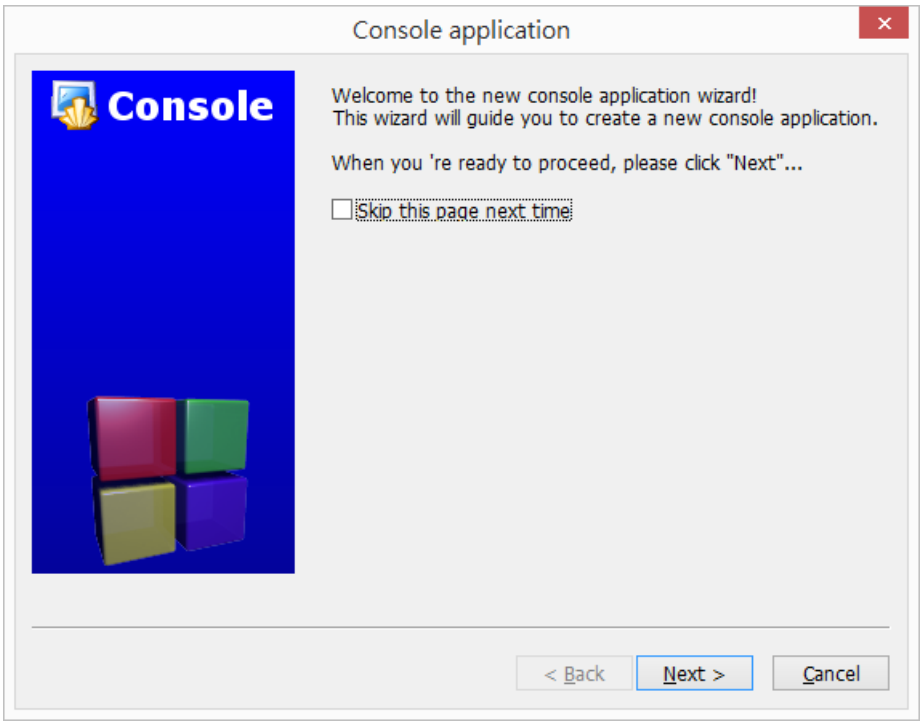
建立專案

點選 [Create new project]，或是 [File] → [New] → [Projects...]

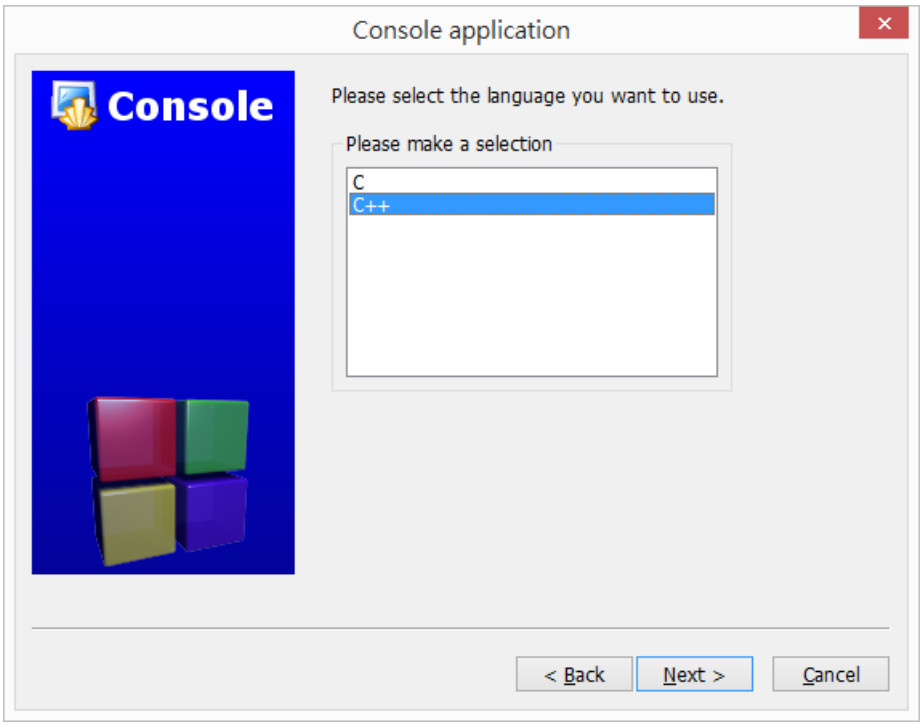
專案樣板(template)選擇 Console application。



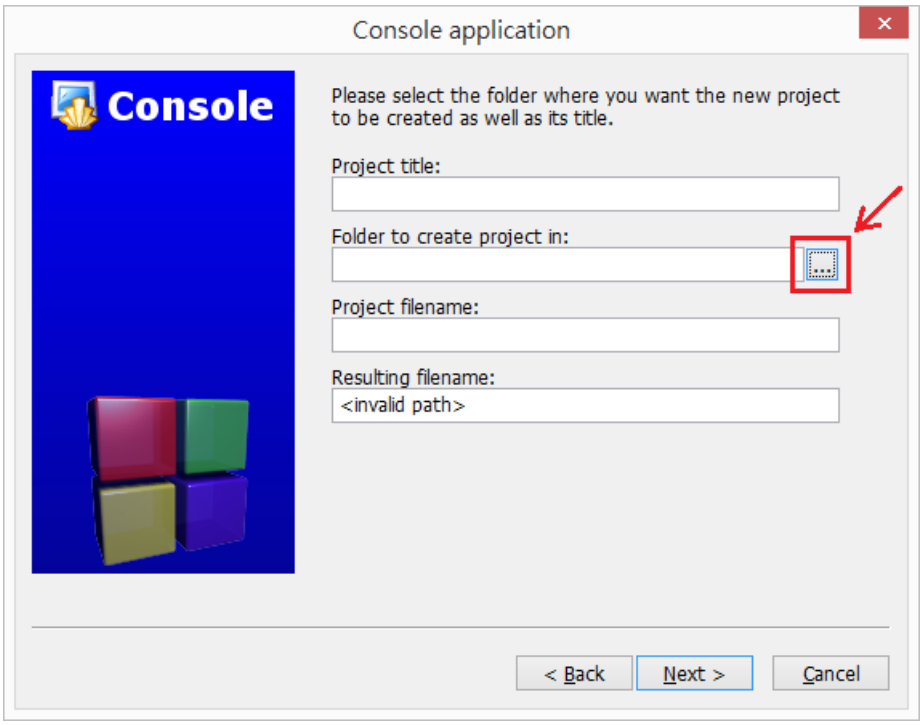
這頁只是說明，按 [Next] 即可。



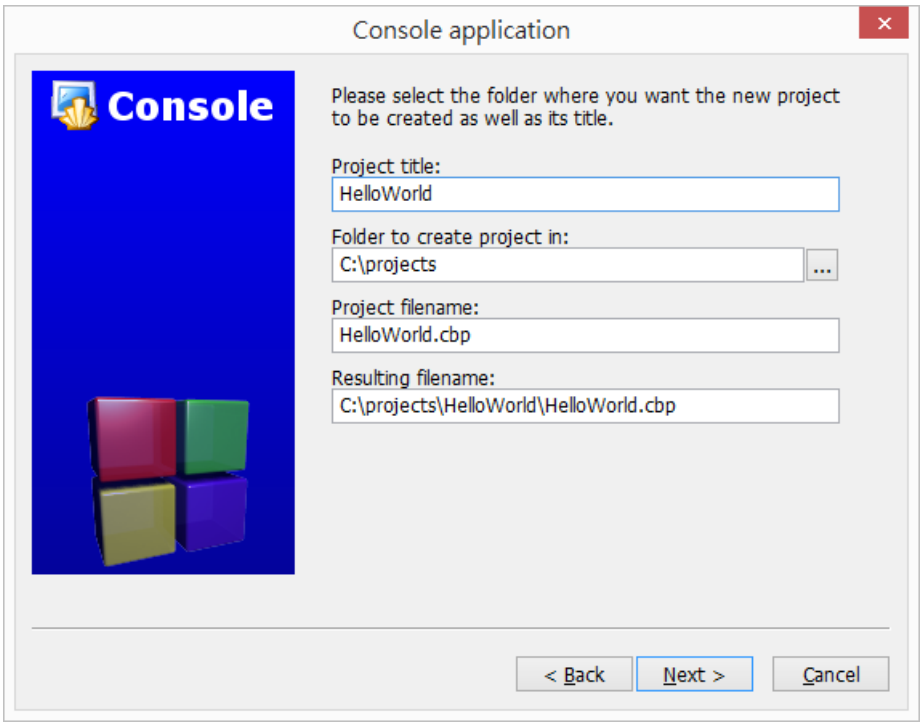
程式語言部分，選擇 [C++]。



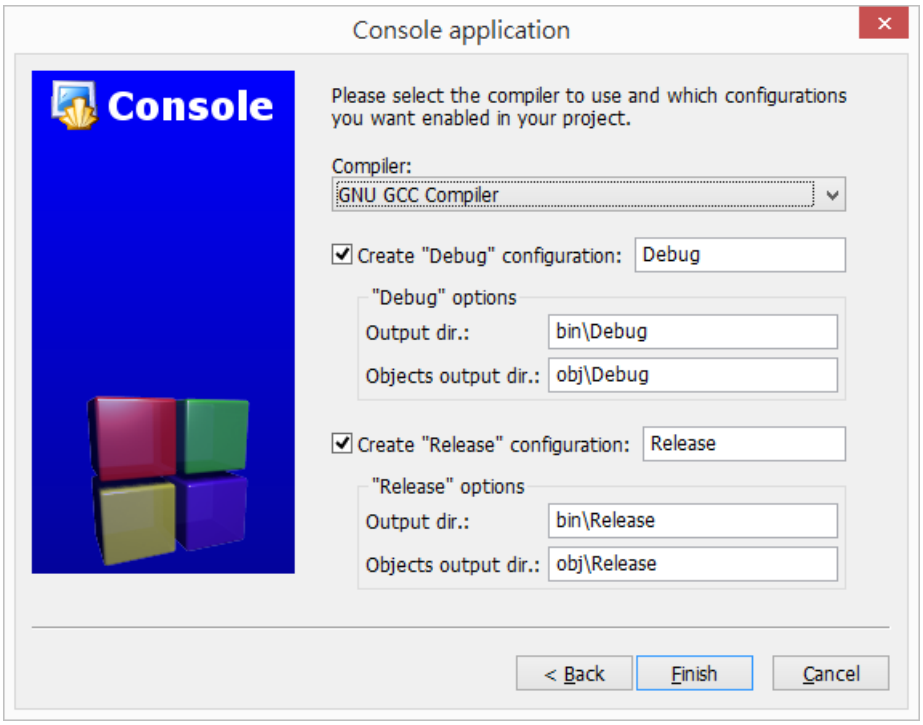
瀏覽到剛剛建立的資料夾。



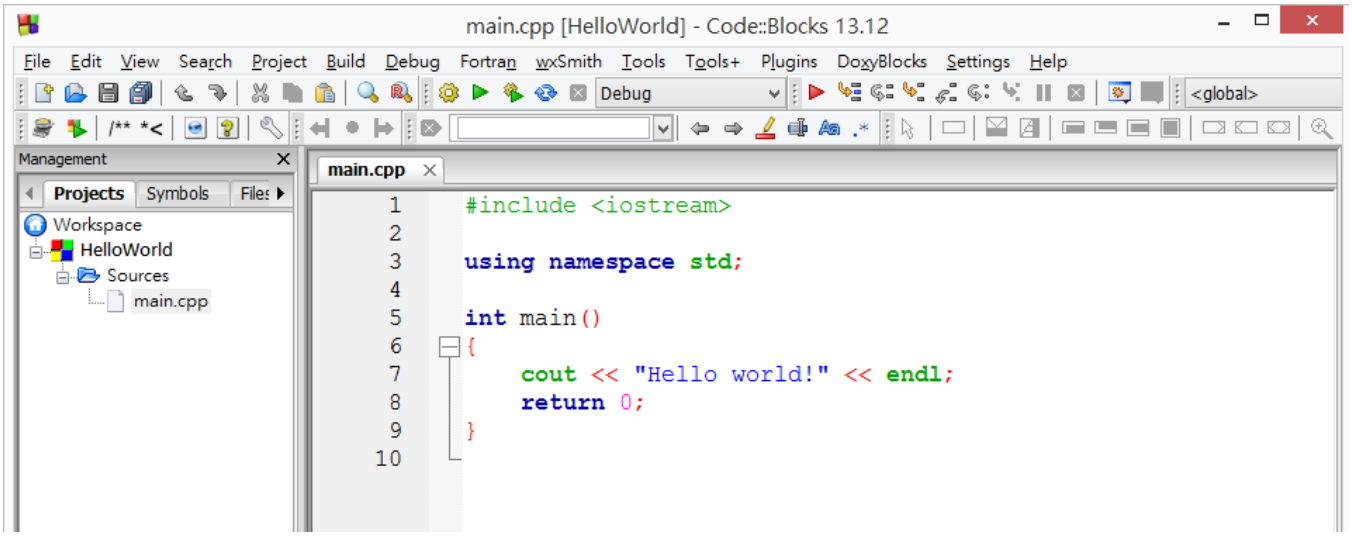
在 Project title 欄位輸入專案名稱 “HelloWorld”。其他欄位的值會自動填好。按下 [Next] 即可。



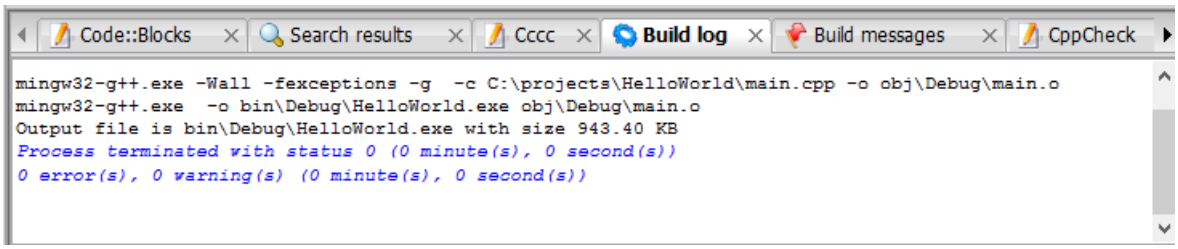
接受預設值，完成新增專案 [Finished]。



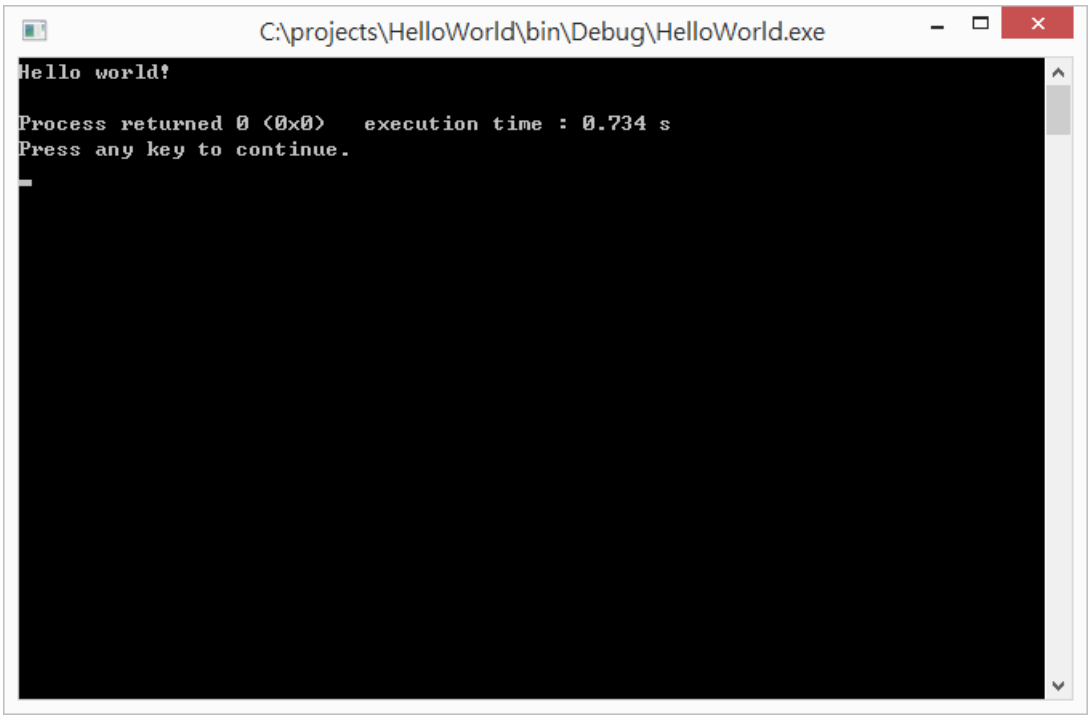
將左側專案視窗中的 HelloWorld、Sources 展開，然後在 main.cpp 上點兩下。這個程式檔會在右側開啟。



按一下中間 **黃色的齒輪按鈕** 或是 [Build] →[Build]。順利的話會在下方的 [Build log] 裡看到以下的訊息。0 error(s), 0 warning(s) 表示沒有錯誤，也沒有警告。



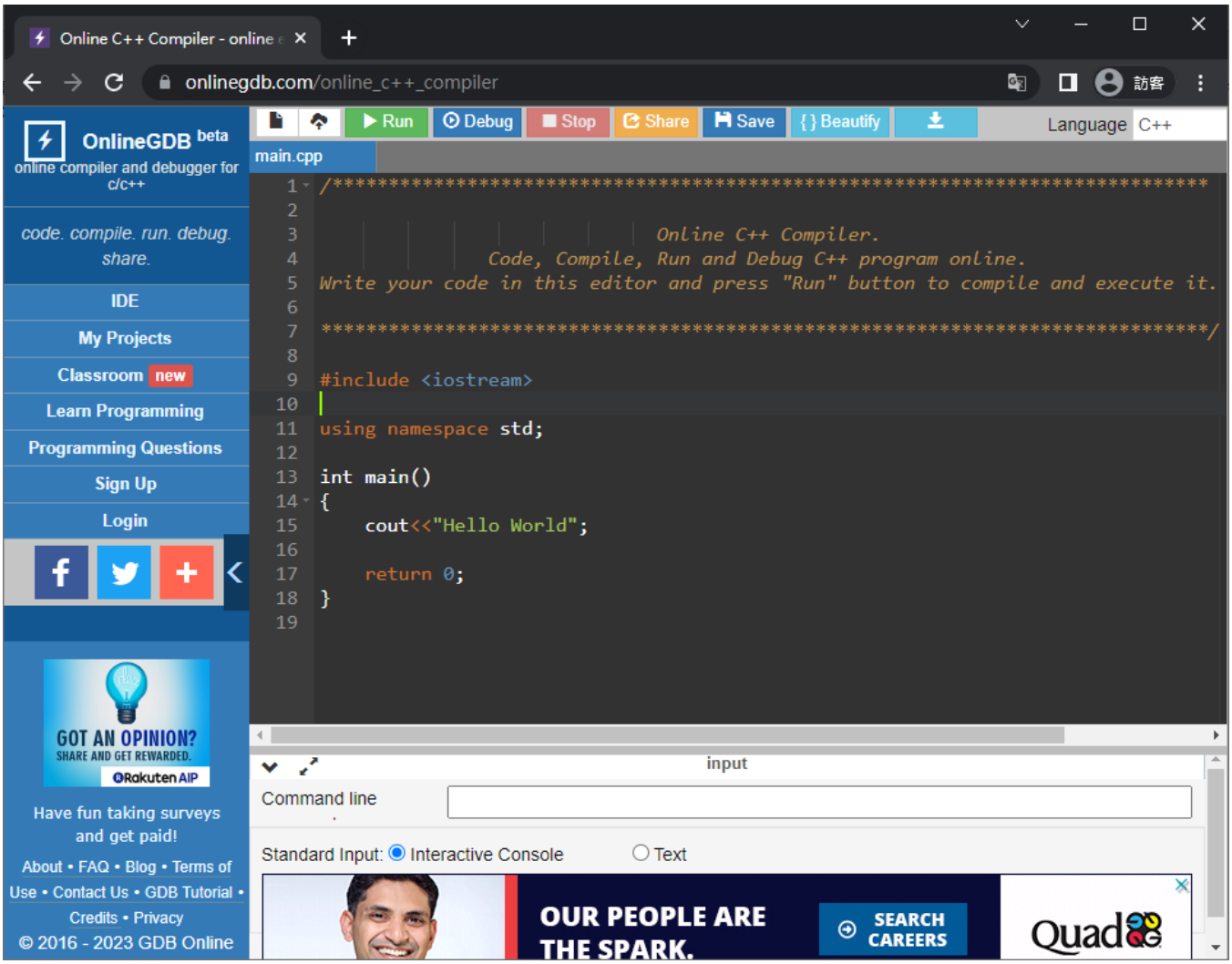
按下中間三角形的 **綠色播放按鈕** ，可以看到執行結果如下。


C:\projects\HelloWorld\bin\Debug\HelloWorld.exe
Hello world!
Process returned 0 (0x0)   execution time : 0.734 s
Press any key to continue.

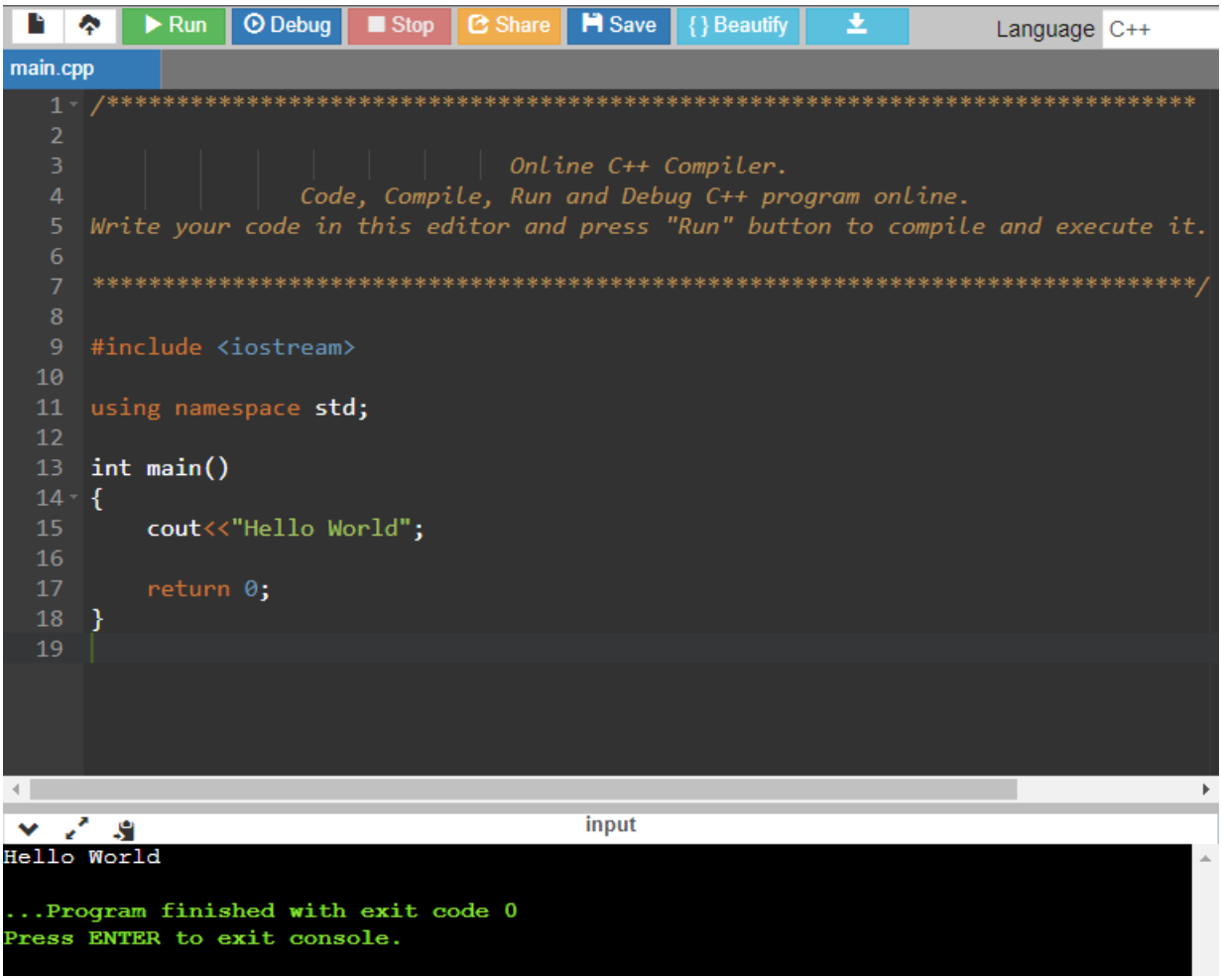
按任一鍵，結束執行視窗。

1.2 可在線上撰寫程式的 OnlineGDB

如果你只是在練解題，只會寫些小程式，OnlineGDB 是一個不錯的選擇，你不用安裝開發環境，只要連上 OnlineGDB 網站，就可以直接使用。



按下上面綠色的 [Run]，執行結果就會顯示在下面。



當手邊沒有電腦，只有手機和平板時，這種線上開發平台就很方便。

2-變數與輸入、輸出

2.1 輸出

使用 cout 輸出資料

在 Code::Blocks 裡建立一個專案後，它會自動產生這樣一個程式架構。

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

main function

其中的 `main()` 稱為主函數(main function)，它是程式的起點。程式在啟動後，會由 main 裡的第一行開始依序執行下去。

`return 0;` 表示帶著回傳值 0，返回呼叫 main() 的地方，以這個例子來說就是返回作業系統。

```
int main()
{
    // 要做的事情寫在這裡面
    return 0;
}
```

剩下的那行就是真正在做的事情，程式執行後會在畫面上看到。

```
Hello world!
```

單行註解：雙斜線 // 開始到該行結尾都屬於「註解」，用來對程式碼做說明。這是給人類看的，對執行完全沒影響。

標準輸出

`cout` 是用來將資料輸出到標準輸出(standard output / stdout)，而一般的標準輸出指的是螢幕。

我們可以把它想像成，資料沿著 `<<` 的方向流到螢幕那邊。

至於 `endl` 則是 end of line，即「換行」的意思。如果把程式修改一下。

```
cout << "Hello" << endl << "world!";
```

那麼輸出就會變這樣。

```
Hello
world!
```

提醒：程式碼內容有更動後，都要[save]->[build]，才能[run]。 因為 [build] 會根據你目前的程式碼做出新的執行檔。若沒這麼做，你執行的還是之前舊的執行檔。

字串 和 運算式

比較一下這兩行程式碼有什麼不同。

```
cout << "2+3" << endl;
cout << 2+3 << endl;
```

看起來都是 2+3，執行結果卻大不相同。

```
2+3
5
```

第一行輸出的是 `"2+3"`，前後有雙引號。雙引號框起來的內容都會被視為文字，整個被當作字串(string)來處理，所以你寫什麼樣子，它就輸出什麼樣子。

第二行輸出的是 `2+3`，前後**沒有**雙引號。在這種情況下，cin 會等待 `2+3` 這個運算式的結果被計算出來，再將其結果輸出。因此我們會看到第二行的輸出結果是 5。

練習

請推測以下程式執行後的輸出結果為何？

自己寫下結果後，再實際輸入這個程式，觀察執行結果。

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    cout << 1 + 2 << endl;
    cout << 1 + 2 * 3 - 4 << endl;
    cout << (1 + 2) * (3 - 4) << endl;
    cout << "(1 + 2) * 5 / 3 = " << (1 + 2) * 5 / 3 << endl;

    return 0;
}
```

2.2 變數與輸入

像 `1`, `24`, `3.14` 這樣的數，我們稱為 **字面常數**(literal constant)，它的值是固定不變的。

另外像我們在數學代數中用到的 `x`, `y`, `z` 等，則稱為 **變數** ，它的值可以改變。

在電腦程式中，變數是很重要的。它可以用來儲存輸入的資料，計算中的數值，表示某個狀態等。

使用 cin 輸入資料

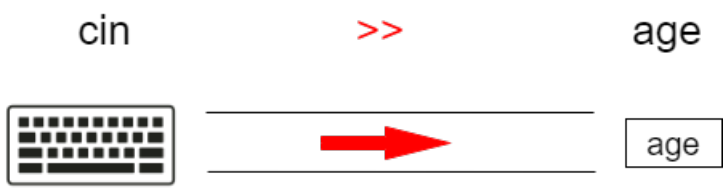
下面這段程式在執行之後，會先詢問你的年齡，在你輸入年齡並按下 [Enter] 後，輸出 `"You are xx years old."`。這個 `xx` 會是你輸入的值。

```
int age;

cout << "How old are you?";
cin >> age;
cout << "You are " << age << " years old." << endl;
```

標準輸入

在程式中出現的 `cin` 是用來由 **標準輸入(standard input)** 將資料讀入電腦，我們可以把它想像成和 `cout` 相反的流向。

一般來說標準輸入指的是鍵盤的輸入，而輸入的值必須被存放到電腦裡，供後續運算和使用。



變數

我們可以把變數想像成是一塊有名字的記憶體。但是它除了有名字之外還有 **型別** ，一個型別為整數(integer)的變數，裡面只能放整數；型別為字串(string)的變數，裡面只能放字串。

宣告

在上面程式碼的第一行 `int age;` ，是在 **宣告(declare)** 這個變數。每一個變數在使用前都必須先宣告，明確指出變數的型別和名字。

一個典型的變數宣告，長這個樣子。

型別 變數名;

例如：

int age;

如果不宣告就使用變數，會發生什麼事呢？

```
#include <iostream>

using namespace std;

int main()
{
    cout << "How old are you?";
    cin >> age;
```

```
    cout << "You are " << age << " years old." << endl;

    return 0;
}
```

在上面這個程式的第 8 行，我們想使用 age 這個變數。但是往前看卻沒有看到這個變數的宣告。

這個程式在編譯時，編譯器(compiler)會發出如下的錯誤訊息。

```
main.cpp: In function ‘int main()’:
main.cpp:8:12: error: ‘age’ was not declared in this scope
    8 |     cin >> age;
      |            ^~~
```

編譯器的錯誤訊息都是英文的，但是同學們一定要學會看錯誤訊息，看懂錯誤訊息可以讓你很快抓到重點，把錯誤修正。

在這段錯誤息裡

- 第一行是告訴我們，它抓到錯誤的位置在 `main.cpp` 這個檔案裡的 `int main()` 函數裡。
- 第二行可以看到精確的位置，第 8 列(row)，第 12 行(column)。所以到第 8 列第 12 個字元的位置，你就可以看到這個錯誤訊息描述的 'age'。
- 繼續往下看 `error` 表示這是個「錯誤」，不修正它程式就無法成功編譯執行。
- 之後則是錯誤的描述 `‘age’ was not declared in this scope`，它說「這個 'age' 沒有在這個範圍內宣告」(它有在下面把位置標記給你看)

> 稍後我們還會看到編譯器給出 `warring` 也就是「警告」的狀況，這是編譯器發現某處可能有問題，但程式依然可以完成編譯並執行。

看懂錯誤訊息後，下一步就是修正它。我們在前面補上宣告即可。

```
#include <iostream>

using namespace std;

int main()
{
    int age; // <-- 我們在這裡補上宣告
    cout << "How old are you?";
    cin >> age;
    cout << "You are " << age << " years old." << endl;

    return 0;
}
```

變數的資料型別和名字

前面提到了兩個重點

- 變數在使用前要 **宣告(declare)**
- 宣告時要明定其 **「型別」** 和 **「名字」**

變數名命規則

我們可以自行命名每一個變數，但是必須遵守以下規則：

1. 變數名稱的第一個字元必須是底線 `_` 或英文字母 `A~Z, a~z`
2. 除了第一個字元外，變數名稱只能由底線 `_` 、英文字母 `A~Z, a~z` 和數字 `0~9` 組成。

- 合法的變數名稱例子，如：`Age`, `age`, `length`, `_name`, `id1246`
- 不合法的變數名稱例子，如：`369city` (開頭不能是數字), `my#name` (變數名稱不能用 #)

注意！ C++裡的變數名稱是有區分大小寫的，也就是 `Age` 和 `age`，會被視為 2 個不同的變數。

常用的基本資料型別

名稱	關鍵字	大小	範圍	備註
字元	char	1 Byte	$0 \sim 255$	• ASCII • 如：'a', '@'
整數	int	4 Byte	$-2^{31} \sim 2^{31}-1$	如：12, -65
無號整數	unsigned int	4 Byte	$0 \sim 2^{32}-1$	如：23, 656372
單精確浮點數	float	4 Byte		• IEEE 754 • 如：3.14, 5.0
雙精確浮點數	double	8 Byte		• IEEE 754 • 如：3.14, 5.0
字串	string			如："Peter"

資料型別牽涉到資料如何被儲存到記憶體裡，以及記憶體中的資料要如何被解讀。

避免誤用或使用不合適的型別

在下面這個用半徑計算圓周長的程式裡，我們宣告了一個名為 pi 的整數(int)型別變數，但是卻把一個浮點數 3.14 放入這個變數裡。

```
    int r;
    cout << "請輸入半徑 r:";
    cin >> r;

    int pi;
    pi=3.14;
    cout << "半徑為 " << r << " 的圓，其周長為 " << 2*pi*r << endl;
```

編譯時沒有出現任何錯誤訊息，執行結果如下。

```
請輸入半徑 r:1
半徑為 1 的圓，其周長為 6
```

如果把第 5 行的 pi 宣告成浮點數

```
    double pi;
```

執行結果則為

```
請輸入半徑 r:1
半徑為 1 的圓，其周長為 6.28
```

修改前因為 int 無法儲存小數的值，所以在 `pi=3.14;` 這裡發生了一些狀況。

3.14 本來是個浮點數，因為被指定(assign)到 int 型別的變數 pi，所以隱式轉型(implicit casting)為 int，喪失精確度變成了 3，過程中沒有出現任何錯誤訊息。如果我們在寫一個需要精確到小數以下 2 位的程式時，誤用 int 型別的變數來儲存數值，那麼結果可能會造成重大的損失。

如果是連喪失精確度轉型都做不到的狀況呢？

```
#include <iostream>

using namespace std;

int main()
{
    int name;

    name = "Peter";  // <--我們在這裡把字串放入整數型別的變數裡
    cout << "Hello, " << name << endl;

    return 0;
```

```
}
```

會出現錯誤訊息。

```
main.cpp: In function ‘int main()’:
main.cpp:9:12: error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
    9 |     name = "Peter";
      |            ^~~~~~~
      |            |
      |            const char*
```

重點在這句 `invalid conversion from ‘const char*’ to ‘int’`

不可以把 `const char*` 轉型為 `int`，因為「字串」沒辦法轉換成「整數」。

連續讀取多個值

cout 可以像這樣串接，一次輸出多組資料。

```
cout << "I am " << 16 << " years old.";
```

cin 也可以串接，讀取多個輸入值。

```
string name;
int age;

cin >> name >> age;
cout << name << " is " << age << " years old.";
```

原則上判斷輸入的斷句是在 `[空白]` 或 `[Enter]`

程式執行後，你可以輸入 `Peter` `[空格]` `16` `[Enter]`，或是輸入 `Peter` `[Enter]` `16` `[Enter]`

第一個輸入的值("Peter")會被放入變數 name 裡，第二個輸入的值(16)會被放入變數 age 裡。

```
Peter is 16 years old.
```

2.3 運算子與運算優先順序

在掌握了基本輸入、輸出之後，我們已經具備「將資料讀進電腦」，「將處理後資料送回外界」的能力，接下來重點就是中間的「處理」，也就是運算的部分。

首先我們要認識兩個名詞：

- **運算元(operands)**
- **運算子(operator)**

以 2+3 為例，`2` 和 `3` 都是運算元，`+` 是運算子。

我們可以把運算子想成是「運算符號」，運算元則是「運算的對象」。

指定(assign)運算子

在 C++ 中，`=` 不是 **等於(equal)** 而是 **指定(assign)**

`a = 3` 是「把 3 **指定** 給 a 這個變數」，而非表示「a 和 3 的值是相等的」

執行完這行後，變數 a 的值就會變成 3。

```
a = 3;
cout << a; // 3
```

運算子 `=` 會將其右側的運算結果，指定到左側的儲存空間。

例如 `v = 3+5` 是把 3+5 的運算結果指定給變數 v。

也就是說如果 `=` 的右側不是單純的值而是運算式，要先完成運算後，再將運算結果指定給其左側的變數。 所以，執行完這行後，變數 v 的值就會變成 8。

```
v = 3+5;
cout << v; // 8
```

想想看，以下這段程式執行後的輸出為何？

```
int a;
a = 3;
a = a + 2;

cout << a << endl;
```

其中第 2 行是把 3 指定給 a，所以執行後 a 的值為 3。

第 3 行因為 `=` 的右側是運算式 `a + 2`，因此要先完成這個運算，目前 a 的值是 3，所以 `a+2` 的運算結果是 5。

接著可以想像第3行變成 `a = 5`，所以整行執行後，a 的值為 5。

千萬不要用數學符號的角度把 `=` 當成「等於」去看待 `a = a + 2`，這行敘述，否則你會看不懂它。

練習

以下這段程式執行後，a 和 b 的值各為何？

```
int a = 3;
int b = 5;

a = a + b;
b = a - b;
a = a - b;
```

```
cout << "a = " << a << endl;
cout << "b = " << b << endl;
```

算術(arithmetic)運算子

「乘、除」 運算子的優先權高於「加、減」運算子，也就是在運算時會 **先乘除後加減**。

例如：`1+2*3-4;` 的運算結果是 `3`。

和數學運算式一樣，可以加上括號指定優先運算的部分。

例如：`(1+2)*(3-4);` 的運算結果是 `-3`。

有個需要留意的地方是，不同於我們在數學課中用小括號、中括號、大括號來一層層的指定優先運算的層次。C++裡只有小括號，不管是幾層都是用小括號來表示。中括號、大括號這兩個符號是用在其他地方。

數學課裡的 $(1+2)\times[(3-4)+2]$

在 C++ 裡是 `(1+2)*((3-4)+2)`

整數除法

除法運算子 `/` 有個需要注意的地方，如果它的左、右側運算元都是整數型別，那麼其運算結果也會是整數。

`cout << 5/2;` 的輸出不會是 `2.5`，而是`2`.

但 `cout << 5.0/2;`、`cout << 5/2.0;`、`cout << 5.0/2.0;` 的輸出，都是 `2.5`。

取餘數

另一個只能用在整數型別的 **模數(modulo)** 運算子 `%`，是用來計算兩整數相除後的餘數。

```
cout << 8%3 << endl; // 2
cout << 6%2 << endl; // 0
cout << 3%7 << endl; // 3
```

在許多演算法中，取餘數是很重要的運算，所以這是個很重要的運算子。

關係(Relational)運算子

下一個單元開始，我們的程式碼將不再只是單純按順序一行一行執行下去，它開始可以按照條件選擇接下來要執行哪一條分支路線。

例如：如果「a大於0」那麼.....否則.....。

那個條件在當下有兩種可能：

- 若成立，表示其為 **真(true)**
- 若不成立，表示其為 **偽(false)**

關係運算子的運算結果是 **布林值(Boolean)**。不同於整數型別的值有多種可能的值 `... -2, -1, 0, 1, 2, ...`，布林型別的值只有兩種 `true`, `false`。

以下是六個關係運算子的作用。

名稱	運算子	範例	範例運算結果
等於	`==`	`2==3`	false
不等於	`!=`	`2!=3`	true
大於	`>`	`2>3`	false
小於	`<`	`2<3`	true

名稱	運算子	範例	範例運算結果
大於等於	`>=`	`2>=3`	false
小於等於	`<=`	`2<=3`	true

邏輯(logical)運算子

前面提到的關係運算子，可以讓我們判斷一個條件是否成立，例如：判斷成績是否及格。

```
score>=60
```

但有時狀況比較複雜一點，例如：要產生補考名單時，我們要確認成績有沒有落在這個區間 $40\leq score<69$。也就是有2個條件要同時成立。

這時候我們可以用 **And 邏輯運算子** `&&`。

```
score>=40 && score<60
```

當 `score>=40` 為 true 而且 `score<60` 為 true，則整條式子的運算結果為 true。

Or 邏輯運算子 `||` 則是只要其中一個運算是 true，則整條式子的運算結果就是 true。例如：只要國文或英文成績大於等於80分，就發給獎學金。

```
chinese>=80 || english>=80
```

Not 邏輯運算子 `!` 可以把邏輯狀態反轉，就是把 true 變成 false，把 false 變成 true。

例如：fileIsOpen 若為 true，表示檔案已開啟。程式在讀取檔案內容前，要確定檔案已開啟。我們想在檔案未開啟時顯示錯誤訊息，只要在這個狀態為 true 時即可。

```
!fileIsOpen
```

因為當 fileIsOpen 為 false 時，把它用 Not 反轉後就是 true。

位元(bitwise)運算子

`&`, `|`, `^`, `~`, `>>`, `<<` 這幾個運算子可以在位元層級做運算，也就是對每一個 bit 做運算。我們會在之後對二進位系統有一定了解後再來討論。

其他的運算子

C++ 還有許多運算子，可以參考這張網上的 [運算子和優先順序圖表]。

*2.4 C 語言的 printf( ) 格式化輸出函數

為什麼需要「格式化」輸出？

想像一下，如果你的程式計算出圓周率是 3.1415926，但你只想在螢幕上顯示 3.14；或者你希望輸出的成績單欄位能夠像表格一樣文字靠左對齊，數值靠右對齊。

這些都無法單純地將變數丟出來就辦到，我們需要「告訴」printf 應該用什麼「格式」來呈現資料，這就是格式化輸出的精髓。

標頭檔

要使用 printf，請務必在程式開頭引用標頭檔：

```
#include <cstdio>
```

`cstdio` 是指 C語言的標準輸入輸出(c standard input output)。

printf 語法與核心：「格式化字串」

printf 的語法結構如下：

```
printf("格式化字串", 變數1, 變數2, ...);
```

它的靈魂就在於第一個參數——**格式化字串**。這個字串由兩種內容組成：

1. 一般文字：會被原封不動地輸出到螢幕上。
2. 格式指定符 (Format Specifier)：以 % 符號開頭，作為一個「佔位符」，它會被後面依序對應的變數值給取代。

這是最基本也最常用的幾種，我們先從它們開始。

指定符	對應資料類型	說明
Text	Text	Text
%d	int(整數)	以十進位形式輸出整數。
%f	float, double (浮點數)	輸出浮點數 (小數)。
%c	char	(字元) 輸出單一字元。
%s	字串 (char 陣列)	輸出一整個字串。
%%	無	若你想顯示一個 % 符號，需使用 %%。

範例 1：基本應用

```
#include <cstdio>

int main() {
    int student_id = 101;
    float score = 85.5;
    char level = 'B';
    printf("學生學號：%d, 分數：%.1f, 等級：%c\n", student_id, score, level);
    // %.1f 的意思是浮點數只顯示到小數點後第一位
    return 0;
}
```

執行結果：

```
學生學號：101, 分數：85.5, 等級：B
```

格式化字串的各種變化與應用

接下來是今天的重頭戲。我們可以對 `%` 加上一些「修飾」，來精準控制輸出的樣式。

1. 控制輸出寬度 (Field Width)

我們可以指定一個數字來表示該欄位最少要佔用的寬度。如果實際內容比指定的寬度窄，預設會在**左邊**用空白補滿（也就是**靠右對齊**）。

語法： `%[寬度]d`、`%[寬度]f` ...

範例 2：讓數字整齊排列

```
#include <cstdio>

int main() {
    int num1 = 123;
    int num2 = 45;
    int num3 = 6789;

    printf("原始輸出：\n");
    printf("%d\n", num1);
    printf("%d\n", num2);
    printf("%d\n", num3);

    printf("\n指定寬度為 5 輸出 (靠右對齊)：\n");
    printf("%5d\n", num1); // 在 123 左邊補 2 個空白
    printf("%5d\n", num2); // 在 45 左邊補 3 個空白
    printf("%5d\n", num3); // 寬度不足 5，但數字不會被切斷，會完整顯示
    return 0;
}
```

執行結果：

```
原始輸出：
123
45
6789


指定寬度為 5 輸出 (靠右對齊)：
  123
   45
 6789
```

2. 控制對齊方式 (Alignment)

如果我們想靠左對齊，只要在寬度數字前加上一個負號 `-` 即可。

語法： `%-[寬度]d`、`%-[寬度]s` ...

範例 3：文字的靠左與靠右

```
#include <cstdio>

int main() {
    char product1[] = "Apple";
    char product2[] = "Banana";

    printf("--- 商品清單 (寬度 10) ---\n");
    printf("靠右對齊: |%10s|\n", product1);
    printf("靠左對齊: |%-10s|\n", product2);
```

```
    printf("-------------------------\n");
    return 0;
}
```

執行結果：

```
--- 商品清單 (寬度 10) ---
靠右對齊: |     Apple|
靠左對齊: |Banana    |
-------------------------
```

對於浮點數 (%f)，我們可以用 .數字 來指定要顯示到小數點後幾位。注意：系統會自動進行四捨五入。

語法：`%.[位數]f`

範例 4：計算圓面積並控制精度

```
#include <cstdio>

int main() {
    double pi = 3.1415926535;

    printf("原始數值：%f\n", pi);
    printf("顯示到小數點後 2 位：%.2f\n", pi); // 輸出 3.14
    printf("顯示到小數點後 4 位：%.4f\n", pi); // 輸出 3.1416 (注意看，有四捨五入！)
    printf("不顯示小數：%.0f\n", pi);          // 輸出 3
    return 0;
}
```

執行結果：

```
原始數值：3.141593
顯示到小數點後 2 位：3.14
顯示到小數點後 4 位：3.1416
不顯示小數：3
```

我們也可以將寬度和精度結合起來，創造出更完美的排版。

語法：`%[總寬度].[小數位數]f`

範例 5：顯示商品價格

```
#include <cstdio>

int main() {
    float price1 = 5.99;
    float price2 = 123.5;

    // 總寬度為 8，小數點後 2 位
    printf("價格清單 (總寬度 8)：\n");
    printf("|%8.2f|\n", price1);
    printf("|%8.2f|\n", price2);
    return 0;
}
```

執行結果：

```
價格清單 (總寬度 8)：
|    5.99|
|  123.50|
```

解說：

總共佔 8 個字元寬，並且小數點後面固定顯示 2 位，不足的會補 0。

3. 補零 (Zero Padding)

如果希望在靠右對齊時，不是補空白而是補 `0`，只要在寬度數字前加上 `0` 即可。這常用於學號、時間等場合。

語法： `%0[寬度]d`

範例 6：顯示學號

```
#include <cstdio>

int main() {
    int id1 = 7;
    int id2 = 123;
    printf("三年一班 座號列表：\n");
    printf("補空白：%3d\n", id1);
    printf("補零：%03d\n", id1); // 寬度 3，不足處補 0
    printf("補零：%03d\n", id2);
    return 0;
}
```

執行結果：

```
三年一班 座號列表：
補空白：  7
補零：007
補零：123
```

總結與速查表

以下表格提供同學們需要時查詢使用：

分類	語法	說明與範例
基礎類型	%d	輸出十進位整數 (int)。 `printf("%d", 100);` → `100`
	%f	輸出浮點數 (float, double)。 `printf("%f", 12.34);` → `12.340000`
	%c	輸出單一字元 (char)。 `printf("%c", 'A');` → `A`
	%s	輸出字串 (char 陣列)。 `printf("%s", "Hi");` → `Hi`
	%%	輸出 % 符號本身。 `printf("100%%");` → `100%`
寬度控制	%[n]d	輸出寬度至少為 n 的整數，靠右對齊。 `printf("%4d",12);` → `  12`
	%- [n]d	輸出寬度至少為 n 的整數，靠左對齊。 `printf("%-4d",12);` → `12  `
精度控制	%.[n]f	輸出浮點數，顯示到小數點後 n 位。 `printf("%.2f", 3.14159);` → `3.14`
組合使用	%[w].[p]f	總寬度為 w，小數點精度為 p。 `printf("%6.2f", 3.14159);` → `  3.14`
特殊旗標	%0[n]d	輸出寬度為 n 的整數，不足處在左邊補 0。 `printf("%04d", 55);` → `0055`

這些符號放在字串中會有特殊功能。

序列	名稱	功能
\n	換行符	將游標移至下一行的開頭。
\t	定位符	(Tab) 將游標移至下一個定位點，常用於對齊欄位。
\	反斜線	顯示一個 \ 符號。
"	雙引號	在字串中顯示一個 " 符號。

練習題作業

請同學動手實作，加深對 printf 格式化的理解。

練習題 1：我的個人檔案

- **目標：** 練習 %s, %d, %.1f 的基本使用。
- **說明：** 請宣告變數來儲存姓名、年齡、身高(公尺)，並依照以下格式輸出。

輸出範例：

```
姓名：陳月光
年齡：17 歲
身高：1.7 公尺
```

練習題 2：商品價目表

- **目標：** 練習使用寬度、對齊與小數點精度，製作對齊的表格。
- **說明：** 有三樣商品及其價格如下：
  - "Milk": 65.5 元
  - "Bread": 42 元
  - "Juice": 51.25 元

請使用 printf 格式化功能，輸出如下對齊的價目表。商品名稱欄位寬度為 10 且靠左對齊，價格欄位總寬度為 8 且顯示到小數點後 2 位。

輸出範例：

```
+------------+----------+
| Item       | Price    |
+------------+----------+
| Milk       |    65.50 |
| Bread      |    42.00 |
| Juice      |    51.25 |
+------------+----------+
```

練習題 3：數位時鐘

- 目標：練習使用補零 0 的技巧。
- 說明：請宣告三個整數變數 h, m, s 分別代表時、分、秒，並賦值 (例如 h=8, m=5, s=30)。請使用 printf 輸出 HH:MM:SS 的格式，也就是不足兩位數時要補零。

輸出範例：

```
目前時間為：08:05:30
```

*2.5 C 語言的 scanf( ) 格式化輸入函數

我們已經學會如何用 printf 讓程式輸出精美的訊息。但一個真正有用的程式，不僅要會「說」，更要會「聽」。它需要接收使用者的指令、數據，才能進行下一步的處理。

scanf (scan formatted) 就是這座溝通的橋樑。它會暫停程式的執行，靜靜地等待使用者從鍵盤輸入資料，然後依照我們指定的「格式」去解析這些輸入，並將它們存放到對應的變數中。

同樣地，要使用 scanf，請務必在程式開頭引用標頭檔：

```
#include <cstdio>
```

2.5.1- scanf 的核心語法與「&」的秘密

scanf 的語法看起來和 printf 有點像，但有一個關鍵且絕對不能忘記的區別。

```
scanf("格式化字串", &變數1, &變數2, ...);
```

- **格式化字串：** 由一或多個「格式指定符」組成，用來告訴 scanf 使用者將會輸入什麼類型的資料。例如，"%d" 代表使用者會輸入一個整數。
- **& (取址運算子)：** 這是 scanf 最重要、也最容易出錯的地方！

讓我們用一個生活化的比喻來理解：

想像一下，變數是一個「置物櫃」，裡面可以存放資料。

- 當你使用 `printf("%d", score);` 時，你是告訴 printf：「打開 score 這個櫃子，把裡面的東西（值）拿出來秀給大家看。」
- 當你使用 `scanf("%d", &score);` 時，你是告訴 scanf：「我給你 score 這個櫃子的地址（&score），請你把使用者輸入的東西，親自送到這個地址的櫃子裡放好。」

scanf 需要的是「存放資料的地址」，而不是「變數裡現有的值」。所以，除了字串陣列（我們稍後會提）以外，`幾乎所有變數在使用 scanf 時都必須在前面加上 & 符號！`

2.5.2- scanf 的各種應用與常見陷阱

常用的格式指定符：

指定符	對應資料類型	說明
%d	int	讀取一個十進位整數。
%f	float	讀取一個浮點數。
%lf	double	注意！ 讀取 double 類型時要用 %lf (long float)，這是新手常犯的錯誤。
%c	char	讀取一個單一字元。
%s	字串 (char 陣列)	讀取一個字串（但遇到空白、Tab或換行時會停止）。

範例 1：讀取學生的基本資料

```
#include <cstdio>

int main() {
    int age;
    double height;

    printf("請輸入你的年齡：");
```

```
    scanf("%d", &age);

    printf("請輸入你的身高 (公尺)：");
    scanf("%lf", &height); // 讀取 double，使用 %lf

    printf("好的，你 %d 歲，身高 %.2f 公尺。\n", age, height);
    return 0;
}
```

執行過程：

```
請輸入你的年齡：17 (使用者輸入後按 Enter)
請輸入你的身高 (公尺)：1.75 (使用者輸入後按 Enter)
好的，你 17 歲，身高 1.75 公尺。
```

你可以在格式化字串中放置多個指定符，scanf 會要求使用者一次輸入多個值，並用空白、Tab 或換行鍵來分隔它們。

範例 2：輸入座標

```
#include <cstdio>

int main() {
    int x, y;
    printf("請輸入一個二維座標 (例如: 15 30)：");
    scanf("%d %d", &x, &y);
    printf("你輸入的座標點為 (%d, %d)\n", x, y);
    return 0;
}
```

執行過程：

```
請輸入一個二維座標 (例如: 15 30)：15 30
你輸入的座標點為 (15, 30)
```

這是 scanf 最經典的陷阱！當你讀取完一個數字後，緊接著要讀取一個字元時，常常會出問題。

範例 3 (錯誤示範)：

```
#include <cstdio>

int main() {
    int choice;
    char confirm;
    printf("請選擇項目 (1-3): ");
    scanf("%d", &choice);
    printf("你確定嗎? (Y/N): ");
    scanf("%c", &confirm); // 這行會出問題
    printf("你的選擇是 %d, 確認字元是 '%c'\n", choice, confirm);
    return 0;
}
```

執行過程：

```
請選擇項目 (1-3): 2 (使用者輸入 2 後按 Enter)
你確定嗎? (Y/N): 你的選擇是 2, 確認字元是 '
'
```

問題分析：

當你輸入 2 並按下 Enter 鍵時，你其實輸入了兩個字元：`'2'` 和 `'\n'` (換行符號)。

`scanf("%d", ...)` 只讀走了 `'2'`，那個 `'\n'` 還留在輸入緩衝區中。輪到 `scanf("%c", ...)` 時，它立刻把還留著的 `'\n'` 讀走了，導致程式根本不等待你輸入 Y 或 N。

解決方案：

在 `%c` 前面加一個空格，`" %c"`。這個空格會告訴 scanf：「請忽略前面所有空白類的字元（包含空格、Tab、換行符），然後再讀取下一個真正的字元。」

範例 3 (正確寫法)：

```
#include <cstdio>
int main() {
    int choice;
    char confirm;
    printf("請選擇項目 (1-3): ");
    scanf("%d", &choice);
    printf("你確定嗎? (Y/N): ");
    scanf(" %c", &confirm); // 在 %c 前加一個空格
    printf("你的選擇是 %d, 確認字元是 '%c'\n", choice, confirm);
    return 0;
}
```

%s 雖然方便，但`它遇到任何空白字元（空格、Tab、換行）就會停止讀取`。

範例 4：讀取姓名

```
#include <cstdio>

int main() {
    char name[30];
    printf("請輸入你的英文全名 (例如: Peter Pan): ");
    scanf("%s", name); // 注意：字串陣列 name 本身就是位址，所以不用加 &
    printf("你好, %s！\n", name);
    return 0;
}
```

執行過程：

```
請輸入你的英文全名 (例如: Peter Pan): Peter Pan
你好, Peter！
```

問題分析：

scanf 只讀到了 `"Peter"` 就因為遇到空格而停止了，`"Pan"` 則被留在了後面。

這也是為什麼在 C++ 中，當需要讀取一整行含有空格的文字時，我們未來會學習更適合的 cin.getline() 或 fgets() 函式。

2.5.3- 總結與速查表

scanf 是程式接收輸入的基礎，雖然有些小陷阱，但只要小心使用，它依然非常強大。

指定符	對應資料類型	關鍵提醒
%d	int	變數前記得加 &。例如 `scanf("%d", &num);`
%f	float	變數前記得加 &。例如 `scanf("%f", &price);`
%lf	double	務必使用 %lf，而非 %f。變數前記得加 &。例如 `scanf("%lf", &pi);`
%c	char	變數前記得加 &。若前面有其他輸入，建議用 `" %c"` 來清除換行符。
%s	char[]	變數前不用加 &。只能讀取不含空白的字串。

2.5.4- 練習題作業

練習題 1：華氏溫度轉換

- **目標：** 練習讀取浮點數 (double) 並進行計算。
- **說明：** 請使用者輸入攝氏溫度，程式將其轉換為華氏溫度後輸出。

- **公式：** 華氏 = 攝氏 * 9 / 5 + 32
- 執行範例：

```
請輸入攝氏溫度：25.0
轉換後的華氏溫度為：77.0
```

練習題 2：簡易對話程式

- **目標：** 綜合練習 %d, %s。
- **說明：** 撰寫一個程式，詢問使用者的名字和學號，然後向他打招呼。
- **執行範例：**

```
你好！請問你叫什麼名字？ (請輸入英文名)
David
你的學號是幾號？
10123
你好, David (學號: 10123)，歡迎使用本系統！
```

練習題 3：解決字元輸入問題

- **目標：** 練習處理 %c 的換行符陷阱。
- **說明：** 撰寫一個程式，讓使用者輸入一個整數代表分數，然後再輸入一個字元代表評等 (A/B/C/D)。最後將兩者一起輸出。請務必確保程式能正確等待使用者輸入評等。
- **執行範例：**

```
請輸入你的分數：88
請輸入你的評等 (A-F)：A
你的成績是 88 分，評等為 A。
```

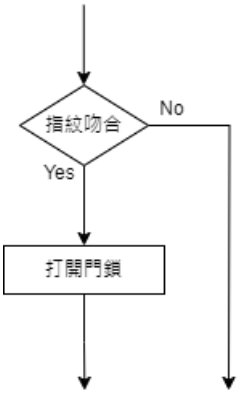
03-選擇結構

3.1 if ... else ...

每一行程式碼都會執行到？

我們寫的每一行程式碼都有用嗎？當然有用。那每一行都會被執行嗎？這要看情況。

之前我們寫的程式，會從 `main()` 函數的第一行開始一行一行依序執行下去，直到程式結束。但在真實世界運作的程式是要有彈性的，例如：指紋鎖必須要在使用者指紋與內部設定吻合時，才會開鎖，否則什麼都不做。也就是這樣一個結構：



接下來我們我們以「計算絕對值」為例，來看看這種結構。

if 敘述

練習：絕對值

讀入使用者輸入的整數 a，計算並輸出其絕對值 $|a|$。

絕對值表示數線上原點到該數值的距離，所以若 $a\ge 0$ 則 a 的值就是其絕對值，否則將 a 的值乘上 -1 才是其絕對值。

```
#include <iostream>

using namespace std;

int main()
{
    int a;

    cin >> a;

    if(a<0)
    {
        a = -1*a;
    }

    cout << "|a|=" << a << endl;

    return 0;
}
```

在這裡我們使用到 if 敘述，它的語法如下：

```
if( 條件判斷式 )
{
    條件成立時要做的事
}
```

if 後面的小括號裡是個 **條件判斷式**，它的運算結果必需是 **布林值(boolean)**。

如果條件判斷式的運算結果為 **true** 則執行接下來那組大括號內的程式碼，否則就略過那整個大括號的內容。

關鍵往往在於你是否能找到一個合宜的條件判斷式，來抓到你要的狀態。

練習：判斷奇數(odd number)

讀入使用者輸入的整數 a，若其為奇數，輸出 "奇數" 否則什麼都不做。

我們要怎麼判定 a 是奇數呢？只要把它除以 2，看餘數是不是 1 就知道了。

```
int a;
cin >> a;

if(a%2==1)
{
    cout << "奇數" << endl;
}
```

if ... else ...

如果我們在條件判斷式成立時要做「工作A」，不成立時要做「工作B」，該怎麼辦呢？

這種 2 條分支的狀況，可以使用 **if ... else ...** 敘述，語法如下。

```
if( 條件判斷式 )
{
    條件成立時要做的事
}
else{
    條件不成立時要做的事
}
```

可以把它想成是「**如果** (...) **就做** {...} **否則做** {...}」

練習：判斷奇、偶數

讀入使用者輸入的整數 a，若其為奇數，輸出 "奇數" 否則輸出 "偶數"。

```
int a;
cin >> a;

if(a%2==1)
{
    cout << "奇數" << endl;
}
else
{
    cout << "偶數" << endl;
```

```
}
```

if ... else if ... else

狀況再複雜一點，如果分支多於 2 條呢？例如：判斷 a 是「正數」、「負數」還是「0」

這時我們需要加入 **else if**，來做到 「**如果** (...) **就做** {...} **否則如果** (...) **就做** {...} **否則做** ...」

```
if( 條件判斷式1 )
{
    條件1成立時要做的事
}
else if( 條件判斷式2 )
{
    條件2成立時要做的事
}
else
{
    前面條件都不成立時要做的事
}
```

練習：正、負數與零的判斷

讀入使用者輸入的整數 a ，輸出其為 "正數"、"負數" 或 "零"。

```
int a;

cin >> a;

if(a>0)
{
    cout << "正數" << endl;
}
else if(a<0)
{
    cout << "負數" << endl;
}
else
{
    cout << "零" << endl;
}
```

分支大於 3 條時，只要重覆多個 **else if** 即可。

練習：分數轉換為等第

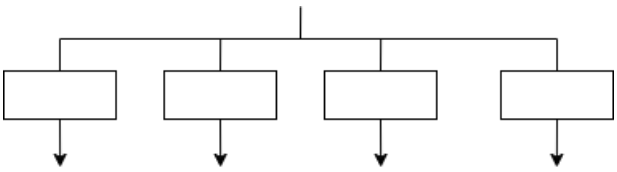
讀取使用者輸入的成績，輸出其相應的等第，轉換參考表如下。

分數	等第
90 ~	A
80 ~ 89	B
70 ~ 79	C
60 ~ 69	D
~ 59	E

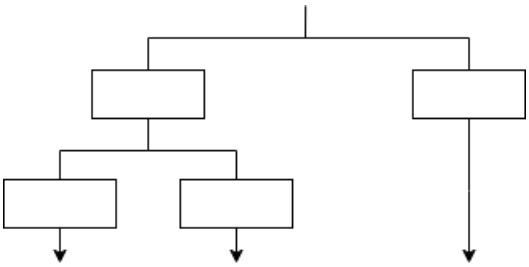
```
int score;

cin >> score;

if(score>=90)
{
    cout << "A" << endl;
}
else if(score>=80)
{
    cout << "B" << endl;
}
else if(score>=70)
{
    cout << "C" << endl;
}
else if(score>=60)
{
    cout << "D" << endl;
}
else
{
    cout << "E" << endl;
}
```

巢狀(nested)/多層 結構

目前我們遇到的選擇結構是像這樣 **單層多分支** 。



但是也有像這樣 **多層多分支** 的選擇結構。



練習：是否需服兵役

由使用者輸入性別、年齡，只有男生且年齡大於等於 20 歲才需要服兵役。

只有 2 個檢查條件都成立，才會被判定需當兵。

```
string gender;
cout << "性別(男, 女):";
cin >> gender;

int age;
cout << "年齡:";
cin >> age;

if(gender=="男")
{
    cout << "你是男生，";
    if(age>=20)
    {
```

```
        cout << "需要當兵" << endl;
    }
   else
    {
        cout << "但是年紀太小，還不用當兵" << endl;
    }
}
else
{
    cout << "你是女生，不用當兵" << endl;
}
```

在這裡我們用到了如下的巢狀結構。

```
if( 條件判斷式1 )
{
    條件 1 成立時要做的事
    if( 條件判斷式2 )
    {
        條件 1, 2 都成立時要做的事
    }
    else
    {
        條件 1 成立、2 不成立時要做的事
    }
}
else
{
    條件 1 不成立時要做的事
}
```

一組大括號包起來的部分，我們稱為一個程式區塊(block)。每個區塊內都可以再塞進其他的區塊。

縮排(indent)

如前所述我們可以有各式各樣很多層次的程式碼。當層次一多起來，程式碼就會開始亂，連自己的的東西都不容易看懂。

所以在撰寫 C++ 程式時，我們都會遵守一個規範，在出現大括號時，裡面的程式碼就會縮一層(4個空白字元或是一個[tab])，這樣可以讓程式碼的層次一目了然。

雖然不縮排，程式也能執行，但是你的伙伴會看不懂你在寫什麼，也不會想看你的程式。

```
string genger;
cout << "性別(男, 女):";
cin >> gender;

int age;
cout << "年齡:";
cin >> age;

if(gender=="男")
{
    cout << "你是男生，";
    if(age>=20)
    {
        cout << "需要當兵" << endl;
    }
    else
    {
        cout << "但是年紀太小，還不用當兵" << endl;
    }
}
else
{
    cout << "你是女生，不用當兵" << endl;
}
```

3.2 關於 if 敘述大括號的使用

內容只有一行時可以省略大括號

`if...else if...else` 的大括號內如果只有一行時，可以省略大括號。

所以前面範例練習的內容可以寫成這樣。

練習：絕對值

```
#include <iostream>

using namespace std;

int main()
{
    int a;

    cin >> a;

    if(a<0)
        a = -1*a;

    cout << "|a|=" << a << endl;

    return 0;
}
```

練習：判斷奇、偶數

```
int a;
cin >> a;

if(a%2==1)
    cout << "奇數" << endl;
else
    cout << "偶數" << endl;
```

但並不建議同學們這樣做，因為這樣有時會難以閱讀而造成意外的錯誤，不如老老實實的都加上大括號。

例如：下面這段程式碼就很難清楚理解，事實上是有錯誤的。

練習：是否需服兵役

```
string gender;
cout << "性別(男, 女):";
cin >> gender;

int age;
cout << "年齡:";
cin >> age;

if(gender=="男")
    cout << "你是男生，";
    if(age>=20)
        cout << "需要當兵" << endl;
   else
        cout << "但是年紀太小，還不用當兵" << endl;
else
    cout << "你是女生，不用當兵" << endl;
```

兩種常見的大括號使用風格

風格一：左右括號都單獨佔一行

```
int a;
cin >> a;

if(a%2==1)
{
    cout << "奇數" << endl;
}
else
{
    cout << "偶數" << endl;
}
```

風格二：左括號放在行末

```
int a;
cin >> a;

if(a%2==1) {
    cout << "奇數" << endl;
}
else {
    cout << "偶數" << endl;
}
```

這兩種撰寫風格都很常見，同學們可以自行選擇。唯一的提醒就是 **務必要正確的縮排**。

3.3 複合條件判斷式

搭配使用邏輯運算子

底下是一個典型的帳密驗證程式片斷。

```
string id, password;

cin >> id;
cin >> password;

if(id=="admin")
{
    if(password=="123456")
    {
        cout << "登入成功" << endl;
    }
    else
    {
        cout << "登入失敗" << endl;
    }
}
else
{
    cout << "登入失敗" << endl;
}
```

因為「帳號正確」、「密碼正確」兩者皆需成立，所以使用了二層 `if...else` 敘述，看起來很累贅。

And 邏輯運算子 &&

「帳號正確」**而且**「密碼正確」可以這樣表示。

```
string id, password;

cin >> id;
cin >> password;

if(id=="admin" && password=="123456")
{
    cout << "登入成功" << endl;
}
else
{
    cout << "登入失敗" << endl;
}
```

邏輯運算子可以對 **布林值(boolean)** 進行運算。

我們把 `id=="admin"` 視為**條件A**，`password=="123456` 視為**條件B**

條件A 的運算結果有兩種可能 true, false。條件B 也是一樣。所以 A and B 有四種可能狀況，表列如下。

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

這種表叫做 **「真值表(truth table)」** ，可以表示某個邏輯運算的各種狀態和運算結果，我們也很常用 0 表示

false，用 1 表示true。

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

由這張 And 的真值表可以看出只有在兩個條件都是 true 的情況下 A and B 的邏輯運算結練才會是 true，其他狀況都是 false。

Or 邏輯運算子 ||

Or 的真值表如下，只要 A, B 其中之一為 true，運算結果就會是 true。

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

如果獲得獎學金的資格是「國文成績85(含)以上」或「英文成績80(含)以上」，則我們可以用 Or 運算子這樣判斷個案是否可以領取獎學金。

```
if(chi>=85 || eng>=80)
{
    cout << "符合領取獎學金資格" << endl;
}
```

Nor 邏輯運算子 !

And 和 Or 運算子左右各有一個運算元，所以我們稱之為 **二元運算子**。

Not 只有右邊一個運算元，所以它是 **一元運算子**。Not 的運算結果是把它後面那個運算元的真值反轉，也就是 true 變成 false，false 變成 true。

A	!A
0	1
1	0

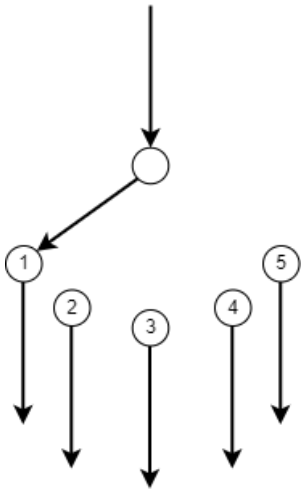
我們可以這樣來過濾 n「不是 4 的倍數」

```
if(!(n%4==0))
{
    cout << n << "不是 4 的倍數" << endl;
}
```

3.4 switch ... case

另一種「多分支」選擇結構

當你遇到像這樣的多分支選擇結構時，可以用 `if...else if...else` 來解決。



例如：一個像這樣的選單功能

```
#include <iostream>

using namespace std;

int main()
{
    cout << "(1) 提款" << endl;
    cout << "(2) 存款" << endl;
    cout << "(3) 查詢帳上餘額" << endl;
    cout << "(0) 結束" << endl;
    cout << "請選擇服務項目(0-3):";

    int i;
    cin >> i;

    if(i==1) {
        cout << "進行提款作業中..." << endl;
        cout << "提款作業完成!" << endl;
    }
    else if(i==2) {
        cout << "進行存款作業中..." << endl;
        cout << "存款作業完成!" << endl;
    }
    else if(i==3) {
        cout << "進行查詢作業中..." << endl;
        cout << "查詢作業完成!" << endl;
    }
    else if(i==0) {
        cout << "結束服務" << endl;
    }
    else {
        cout << "無此項目" << endl;
    }

    return 0;
}
```

除此之外，在 C++ 裡還有另一種 `switch ... case ...` 敘述，非常適合這種選單型的多分支結構。

它的基本語法如下：

```
switch (運算式)
{
    case 常數運算式1:
        ...
        break;
    case 常數運算式2:
        ...
        break;
    ...
    default:
        ...
}
```

- **switch** 後面括號中的運算式，其運算結果必需是 *整數* 或是 *字元*。
- 程式執行到 **switch** 時，會把小括號內運算式的運算結果拿來依序比對 **case** 後面的常數運算式(可能是 *整數* 或是 *字元*)。如果發現符合就跳到那個 **case** 的下一行開始執行，直到遇到 **break** 離開 switch 程式區塊。
- 如果 switch 的程式區塊中有 **default** 關鍵字，則當所有的 case 都不相符時，將由 default 處開始執行。

前面的選單程式，可以改寫成這樣。

```
#include <iostream>

using namespace std;

int main()
{
    cout << "(1) 提款" << endl;
    cout << "(2) 存款" << endl;
    cout << "(3) 查詢帳上餘額" << endl;
    cout << "(0) 結束" << endl;
    cout << "請選擇服務項目(0-3):";

    int i;
    cin >> i;

    // 以下用 switch ... case ... 改寫
    switch(i)
    {
        case 1:
            cout << "進行提款作業中..." << endl;
            cout << "提款作業完成!" << endl;
            break;
        case 2:
            cout << "進行存款作業中..." << endl;
            cout << "存款作業完成!" << endl;
            break;
        case 3:
            cout << "進行查詢作業中..." << endl;
            cout << "查詢作業完成!" << endl;
            break;
        case 0:
            cout << "結束服務" << endl;
            break;
        default:
            cout << "無此項目" << endl;
    }

    return 0;
}
```

break 很重要

使用 `switch ... case ...` 時最常發生的錯誤就是忘記加上 **break**。

case x: 只是一個標籤，程式跳到符合的標籤後開始向下執行，若沒遇到 **break** 會一直向下執行下去，即使遇到另一個 case 也是一樣。

```
    int num = 1;

    // 這不是我們想要的
    switch(num) {
        case 1:
            cout << "this number is 1." << endl;
        case 2:
            cout << "this number is 2." << endl;  // num 為 1 時這行也會執行到
    }
```

程式輸出如下:

```
this number is 1.
this number is 2.
```

```
    int num = 1;
    // 這才是我們想要的
    switch(num) {
        case 1:
            cout << "this number is 1." << endl;
            break;
        case 2:
            cout << "this number is 2." << endl;
            break;
    }
```

程式輸出如下:

```
this number is 1.
```

Fall-through - 利用沒加 break 的副作用

有時候我們會故意不加 **break** 利用它會一直執行下去的副作用來達到特別的目的。

練習：各月份所屬的季節

讀取使用者輸入的一個整數 $m, (1 \leq m \leq 12)$，輸出其所屬的季節。

- 2, 3, 4 月：春
- 5, 6, 7 月：夏
- 8, 9, 10 月：秋
- 11, 12, 1 月：冬

這樣寫看起來很累贅。

```
    int m;
    cin >> m

    switch(month) {
        case 2:
            cout << "Spring" << endl;
            break;
        case 3:
            cout << "Spring" << endl;
            break;
        case 4:
            cout << "Spring" << endl;
            break;
        case 5:
            cout << "Summer" << endl;
            break;
        case 6:
```

```
            cout << "Summer" << endl;
            break;
        case 7:
            cout << "Summer" << endl;
            break;
        case 8:
            cout << "Fall" << endl;
            break;
        case 9:
            cout << "Fall" << endl;
            break;
        case 10:
            cout << "Fall" << endl;
            break;
        default:
            cout << "Winter" << endl;
    }
```

這樣寫就好多了。

```
    int m;
    cin >> m

    switch(month) {
        case 2:
        case 3:
        case 4:
            cout << "Spring" << endl;
            break;
        case 5:
        case 6:
        case 7:
            cout << "Summer" << endl;
            break;
        case 8:
        case 9:
        case 10:
            cout << "Fall" << endl;
            break;
        default:
            cout << "Winter" << endl;
    }
```

3.5 三元運算子？：

3.5.1 三元運算子？：

在 C++ 中，三元運算子（Ternary Operator）是唯一一個需要三個運算元的運算子。它的符號是？和：。

這個運算子主要用來取代簡單的 if-else 判斷式，讓程式碼在一行內就能完成條件判斷與賦值，非常方便。

語法

三元運算子的基本語法結構如下：

```
條件式 ? 運算式1 : 運算式2;
```

條件式 (Condition): 這是一個會回傳 true (真) 或 false (假) 的布林表達式。

- **? :** 這是三元運算子的核心符號。
- **運算式1 (Expression1):** 如果「條件式」的結果為 true，則執行這個運算式，並將其結果作為整個三元運算式的最終結果。
- **運算式2 (Expression2):** 如果「條件式」的結果為 false，則執行這個運算式，並將其結果作為整個三元運算式的最終結果。

3.5.2 實例練習

讓我們來看幾個例子，比較一下使用 if-else 和使用三元運算子的差別。

範例：判斷奇偶數

假設我們要讓使用者輸入一個整數，然後判斷它是奇數還是偶數。

傳統的 if-else 寫法：

```
    int number;
    cout << "請輸入一個整數: ";
    cin >> number;

    string result;
    if (number % 2 == 0) {
        result = "偶數";
    } else {
        result = "奇數";
    }

    cout << "這個數字是 " << result << endl;
```

使用三元運算子的寫法：

```
    int number;
    cout << "請輸入一個整數: ";
    cin >> number;

    // 一行就搞定！
    string result = (number % 2 == 0) ? "偶數" : "奇數";

    cout << "這個數字是 " << result << endl;
```

解說：

在三元運算子的版本中，(number % 2 == 0) 是我們的條件式。

- 如果 number 除以 2 的餘數為 0 (條件為 true)，則回傳？後面的字串 "偶數"。

- 如果餘數不為 0 (條件為 false)，則回傳：後面的字串 "奇數"。
- 最後，回傳的字串會被直接賦值給 result 變數。是不是簡潔很多呢？

你還可以進一步像這樣直接使用他的運算結果

```
    int number;
    cout << "請輸入一個整數: ";
    cin >> number;

    // 注意：要用 ( ) 包覆整個運算式
    cout << "這個數字是 " << ((number % 2 == 0) ? "偶數" : "奇數") << endl;
```

範例：絕對值計算

```
    int a;
    cout << "請輸入一個整數 a: ";
    cin >> a;

    cout << "|a|= " << ((a < 0) ? -a : a) << endl;
```

3.5.2 使用時機與注意事項

- **優點：** 程式碼簡潔。
- **缺點：** 不適合處理複雜的邏輯。如果 `if` 或 `else` 區塊中需要執行多行程式碼，就不應該使用三元運算子，否則會讓程式碼變得難以閱讀和維護。
- **原則：** 當 `if-else` 只是為了根據一個簡單條件來賦予變數不同的值時，就是使用三元運算子的最佳時機。

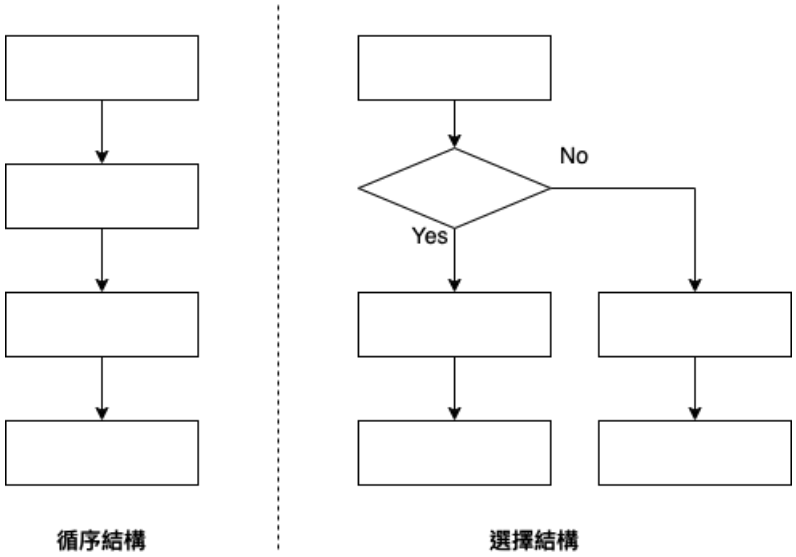
三元運算子 `? :` 是一個非常實用的語法糖 (Syntactic Sugar)，能讓你用更精簡的方式寫出條件判斷。熟練使用它可以提升程式碼的美觀與效率，但切記不要濫用，以免降低複雜邏輯的可讀性。

04-重覆結構

4.1 while 迴圈

程式執行流程結構

目前為止我們學過了兩種程式執行的流程結構，(1)循序結構；(2)選擇結構。



接下來我們要學的是 **重覆結構**，也就是可以重覆執行一段程式。

while

練習：輸出一行，共 5 個 '*'

這個很簡單，只要一行 cout 就能搞定。

```
cout << "*****" << endl;
```

那如果是這題呢？

練習：輸出一行，共 375 個 '*'

我們不太可能傻傻的在字串裡一邊打字一邊數 375 個吧？我們想要的是重覆 `cout << '*';` 375 次。而且要簡單明瞭，不是複製後貼上 375 次。

在這裡我們引入 `while` 敘述，它可以在指定條件成立時，不斷重覆指定的工作，直到該條件不再成立為止。

while 的基本語法如下：

```
while( 條件判斷式 )
{
    條件成立時要重覆做的事
}
```

就輸出 375 個 '*' 來說，使用 while 可以這麼做。

```
int i=1;

while(i<=375)
{
    cout << '*';
```

```
    i = i+1;  // 這行如果忘記，迴圈永遠不會結束
}
cout << endl;
```

請注意在這個 while 迴圈中，變數 i 擔任的角色。它一開始的初值是 1，每執行一次會遞增 1，當 i 大於 375 之後，就離開迴圈。這是一個「計數器」的角色，我們利用一個變數來記錄這個迴圈繞到第幾圈了。

練習：輸出一行，共 n 個 '*'

如果我們要程式更有彈性一點，由使用者指定要輸出的 '*' 數量 n。

只要把前一個程式的 375 改成變數 n 即可。

```
int n;
cin >> n;

int i=1;

while(i<=n)
{
    cout << '*';
    i = i+1;
}
cout << endl;
```

雖然在前面的例子裡我們說 i 擔任「計數器」的角色，但它本質上就只是個變數，也可以參與到迴圈內的計算、輸出……。

練習：輸出 1~n

在這個例子裡，我們在迴圈的每一圈輸出 i 當下的值。

```
int n;
cin >> n;

int i=1;
while(i<=n)
{
    cout << i << " ";
    i = i+1;
}
cout << endl;
```

```
5
1 2 3 4 5
```

接下來這題我們來看兩種做法。

輸出 1~n 之間的奇數

方法一： 首項為1，公差為2 的等差數列

```
int n;
cin >> n;

int i=1;  // 首項為 1
while(i<=n)
{
    cout << i << " ";
    i = i+2;  // 公差為 2
}
cout << endl;
```

```
10
1 3 5 7 9
```

方法二： 在 1~n 之間，逐一過濾符合條件的才輸出

```
int n;
cin >> n;

int i=1;
while(i<=n)
{
    if(i%2==1)
    {
        cout << i << " ";
    }
    i = i+1;
}
cout << endl;
```

```
10
1 3 5 7 9
```

就上題來說，**方法一** 比較有效率。但有時候除了逐一過濾檢查之外，沒有更好的辦法。

練習：輸出 n 的所有正因數

某個整數的因數，可不會簡單到成等差數列分布。n 的所有正因數是在 1 ~ n 之間每個可以整除 n 的整數。

```
int n;
cin >> n;
cout << n << " 的正因數有：";

int i=1;
while(i<=n)
{
    if(n%i==0)
    {
        cout << i << " ";
    }
    i = i+1;
}
cout << endl;
```

```
16
16 的正因數有：1 2 4 8 16
```

練習：n 有幾個正因數？

這個例子裡，我們要的是正因數的數量，作法為：

1. 將一個用來計數用的變數歸零
2. 每發現一個正因數，就將該計數累加 1
3. 最後輸出該計數值

```
int n;
cin >> n;

int sum = 0; // 一開始要給定初值歸零

int i=1;
while(i<=n)
{
    if(n%i==0)
    {
        sum = sum+1;
    }
    i = i+1;
}
cout << n << " 有 " << sum << " 個正因數" << endl;
```

```
16
16 有 5 個正因數
```

break - 跳出迴圈

有時候我們使用迴圈的目的是要在一個可能範圍中 (1)找出特定目標，(2)確定某條件。所以一但找到或確定了，就可以離開迴圈，不必繼續把後面的圈數跑完。

質數判定

在數學和電腦科學中，判定一個正整數是否為質數是很重要的。如果你去 google 一下，會發現方法有非常多種。在這裡我們使用一個最簡單的概念來實作這個判定 - **「如果一個正整數 n 只有 1 和 n 這兩個因數，則 n 為質數」**。

練習：質數判定(1)

使用前一個「n 有幾個正因數？」程式碼，可以很快完成這個質數判定程式。

正因數數量為 2 的是質數。其他的都不是質數。

```
int n;
cin >> n;

int sum = 0;

int i=1;
while(i<=n)
{
    if(n%i==0)
    {
        sum = sum+1;
    }
    i = i+1;
}

if(sum==2)
{
    cout << n << " 是質數" << endl;
}
else
{
    cout << n << " 不是質數" << endl;
}
```

練習：質數判定(2)

我們也可以這權想：在 2 ~ (n-1) 之間，如果發現任一個 n 的因數，那麼 n 就不是質數，反之 n 為質數。

下面這個例子，我們先假設 n 是質數，記錄在布林型別的變數 is Prime 中(isPrime = true)。

接著嘗試在 2 ~ (n-1) 之間試試看能否找到 n 的因數。若找到了，表示 n 不是質數，把這個事實記錄下來(isPrime = false)。

在把所有可能的因數都看完後，由 isPrime 的值即可判定 n 是否為質數。

```
int n;
cin >> n;

bool isPrime = true; // 先假設 n 是質數 (n is a prime number)

int i=2;
while(i<n)
{
    if(n%i==0)
```

```
    {
        isPrime = false;  // 如果發現任一 n 的因數，修正 isPrime 為 false
    }
    i = i+1;
}

if(isPrime)
{
    cout << n << " 是質數" << endl;
}
else
{
    cout << n << " 不是質數" << endl;
}
```

練習：質數判定(3)

上面這個判定質數演算法是可行的，但是效率上有頗多浪費之處。例如 n=256，明明一開始我們就發現 2 是 n 的因數，當下就可以判定 n 不是質數，但卻還是把剩下的 253 圈(i=3~255)跑完。

在第 11 行之後，我們就可以跳出迴圈了。

在這裡我們可以使用 `break` 敘述，程式執行到 break 時，跳出當下所在的那一層迴圈。

雖然只是加了這一行，卻可以省下大量的時間。

```
int n;
cin >> n;

bool isPrime = true; // 先假設 n 是質數 (n is a prime number)

int i=2;
while(i<n)
{
    if(n%i==0)
    {
        isPrime = false;
        break;  // 跳出迴圈
    }
    i = i+1;
}

if(isPrime)
{
    cout << n << " 是質數" << endl;
}
else
{
    cout << n << " 不是質數" << endl;
}
```

continue - 繼續下一圈

想像一下這個場景，你是一個在櫃枱負責審核資料的員工，客戶按抽到號碼牌的順序來到你面前。對每個客戶，你要審查他給你的 20 張表單是否符合申請需求。

整個流程應該是像這樣。

```
num = 0

while(還沒到下班時間)
{
    num = num+1
    廣播請 num 號到櫃枱
    審查 表單1
    審查 表單2
    審查 表單3
```

```
    ......
    審查 表單20
}
```

如果今天有個客戶，他的表單 3 不符資格，當下你就可以告知他審查結果為「不符資格」，並請下一位客戶過來櫃枱，無需再把他後面的 17 張表格看完。

`continue` 就是這樣一個「下一位」的敘述。程式執行到 continue 時，會略過當下那圈剩下的所有工作，直接回到迴圈的開頭並繼續執行下去。

練習：排除 1 ~ n 間，3 的倍數和尾數為 3 的數。

```
int n;
cin >> n;

int i=0;

while(i<n)
{
    i = i+1;
    if(i%3==0 || i%10==3)
    {
        continue;
    }
    cout << i << " ";
}
cout << endl;
```

```
16
1 2 4 5 7 8 10 11 14 16
```

注意！在 while 迴圈中，continue 只是立刻回到迴圈開頭處，判斷若條件成立便再進入迴圈執行。並不會自己幫你把 i 的值加 1。

所以若把上面的程式改成這樣是不會正確運作的。

```
int n;
cin >> n;

int i=1;

while(i<=n)
{
    if(i%3==0 || i%10==3)
    {
        continue;  // i 沒有遞增，會造成無窮迴圈
    }
    cout << i << " ";
    i = i+1;
}
cout << endl;
```

讀取若干組資料

在競技程式設計比賽時，很常見一種輸入資料不確定有幾組的狀況，例如以下這個例子。

練習：加總計算

輸入說明： 輸入為若干個整數

輸出說明： 請輸出這些整數的總合。

若干個？你根本不知道有幾個數要怎麼做？要讀到第幾個才能輸出？

在競賽中並不是由裁判手動在那裡用鍵盤輸入資料，他們早把要輸入的資料都寫到一個檔案裡，然後再把那個檔案 **餵給**

你的程式。

有時候也會註明，輸入資料以 EOF 做為結束，EOF 即 `End Of File`，就是檔案的結束。所以你要做的就是一直讀到沒有資料可以讀為止。

以下為常見的模版，使用 `while(cin>>a)` 迴圈來讀取不定數量的資料。

```
#include <iostream>

using namespace std;

int main()
{
    int sum = 0;
    int a;

    while(cin>>a)  // 順利讀到資料即為 true，否則為 false
    {
        sum = sum+a;
    }
    cout << sum << endl;

    return 0;
}
```

4.2 do...while 迴圈

猜數字遊戲

有時候事情要先做了，看狀況才知道要不要繼續下去。例如我們小時候玩的猜數字遊戲，A 心裡選定一個 1~100 之間的整數由 B 來猜，每次 B 猜了之後，A 就要回應他 (1)再大一點；(2)再小一點；(3)答對了。直到 B 猜中那個數字為止。目標是在最少的猜測次數中，命中正確答案。

把它寫成程式，大致如下。主要問題在於，B 要先猜一個數字，你才知道他猜的對不對，要不要繼續讓他猜下去。我們按下面程式這樣設計，while 迴圈的第一次條件判斷會遇到問題 -「yourguess 的值還沒確定」，因為 B 根本還沒開始猜。

```
int answer = 32;  // A選定的數字
int yourguess;    // 你猜的數字
int count = 0;    // 記錄猜了幾次

while(answer!=yourguess)  // B 根本就還沒開始猜，yourguess 是多少？
{
    cout << "請猜一個數字(1~100):";
    cin >> yourguess;
    count++;

    if(yourguess<answer)
    {
        cout << "再大一點" << endl;
    }
    else if(yourguess>answer)
    {
        cout << "再小一點" << endl;
    }
}

cout << "答對了！你一共猜了 " << count << "次" << endl;
```

解決的方法大致有兩種。

方法一：先在迴圈外猜一次

```
int answer = 32;  // A選定的數字
int yourguess;    // 你猜的數字

cout << "請猜一個數字(1~100):";
cin >> yourguess;
int count = 1;    // 這裡猜了一次

while(answer!=yourguess)
{
    if(yourguess<answer)
    {
        cout << "再大一點" << endl;
    }
    else if(yourguess>answer)
    {
        cout << "再小一點" << endl;
    }
    else
    {
         cout << "請猜一個數字(1~100):";
         cin >> yourguess;
         count++;
    }
}
```

```
cout << "答對了！你一共猜了 " << count << "次" << endl;
```

這種作法會在外面重覆一段程式碼。

方法二：給定 yourguess 一個保證錯的數值

這個作法可以保證 while 第一圈的條件判斷式一定成立，但是若是規則包含可以使用負數、範圍可自定，那就比較麻煩了。

```
int answer = 32;     // A選定的數字
int yourguess = -1;  // -1 在可能的答案範圍之外
int count = 0;       // 記錄猜了幾次

while(answer!=yourguess)  // 第一圈保證是 false
{
    cout << "請猜一個數字(1~100):";
    cin >> yourguess;
    count++;

    if(yourguess<answer)
    {
        cout << "再大一點" << endl;
    }
    else if(yourguess>answer)
    {
        cout << "再小一點" << endl;
    }
}

cout << "答對了！你一共猜了 " << count << "次" << endl;
```

do ... while

有別於 `while` 是先確定條件判斷式才進去執行一圈，我們還有一種 `do ... while` 敘述，可以在做完一圈工作後，再判斷要不要執行下一圈。

`do ... while` 的基本語法如下：

```
do
{
    條件成立時要重覆做的事
}while( 條件判斷式 );
```

> 注意：do ... while(條件判斷式) 最後面有一個分號

使用 do ... while 就可以完美解決我們問題。

```
int answer = 32;  // A選定的數字
int yourguess;    // 你猜的數字
int count = 0;    // 記錄猜了幾次

do
{
    cout << "請猜一個數字(1~100):";
    cin >> yourguess;
    count++;

    if(yourguess<answer)
    {
```

```
        cout << "再大一點" << endl;
    }
    else if(yourguess>answer)
    {
        cout << "再小一點" << endl;
    }
}while(answer!=yourguess);

cout << "答對了！你一共猜了 " << count << "次" << endl;
```

比較 while 和 do ... while

絕大多數的情況下，只要用一點技巧，while 和 do ... while 可以互相取代。

以下的比較供大家判斷當下使用何者較恰當。

	判斷條件的時機	區塊被執行的次數
while	先檢查條件是否成立再做事	可能一次都不會被執行
do ... while	先做事再檢查條件是否成立	至少執行一次

4.3 遞增、遞減與複合指定運算子

遞增與遞減運算子

我們很常在迴圈裡用到 i = i+1 這樣的遞增敘述。

```
int i=1;

while(i<=10)
(
    cout << i << " ";
    i = i+1;  // 遞增 1
}
cout << endl;
```

```
1 2 3 4 5 6 7 8 9 10
```

這種情況可以使用 **遞增(increment)運算子** `++` 來處理。

```
int i=1;

while(i<10)
(
    cout << i << " ";
    i++;  // 遞增 1
}
cout << endl;
```

```
1 2 3 4 5 6 7 8 9 10
```

`i++` 就相當於 `i=i+1`。

同樣的 `i=i-1;` 可以用 **遞減(decrement)運算子** `--` 來處理。

```
int i=10;

while(i>0)
(
    cout << i << " ";
    i--;  // 遞減 1
}
cout << endl;
```

```
10 9 8 7 6 5 4 3 2 1
```

複合指定運算子

如果是增減 1 之外的數值，如 `i = i+2`，則可以用 **複合指定(compound assignment)運算子**。

```
int i=1;

while(i<10)
(
    cout << i << endl;
    i+=2;  // 遞增 2
}
```

`i+=2` 就相當於 `i=i+2`。

常用的複合指定運算子

運算子	範例	相當於
`+=`	`i += 2`	`i = i+2`
`-=`	`i -= 2`	`i = i-2`
`*=`	`i *= 2`	`i = i*2`
`/=`	`i /= 2`	`i = i/2`
`%=`	`i %= 2`	`i = i%2`

遞增、遞減運算子的評估時機

遞增運算子有兩種使用方式，如果我們要將 `變數i` 遞增 1。

- `i++`
- `++i`

以下兩個程式的執行結果相同。

使用 i++

```
int i=1;

while(i<10)
(
    cout << i << " ";
    i++;  // 遞增 1
}
cout << endl;
```

使用 ++i

```
int i=1;

while(i<10)
(
    cout << i << " ";
    ++i;  // 也是遞增 1
}
cout << endl;
```

那麼 `++` 放在變數的前面和後面有什麼差別呢？主要在於 **先遞增再評估其值** 還是 **先評估其值再遞增**。

看了以下這個實例應該就很清楚了。

```
int i=1;

cout << i++ << endl;  // 1
cout << i << endl;    // 2
cout << ++i << endl;  // 3
cout << i << endl;    // 3
```

執行到第3行時，cout 要輸出 i++ 的值，到底是 **要先輸出i的值，再遞增i** 還是要 **先遞增i，再輸出i的值**？

因為我們把 `++` 寫在 `i` 後面，所以當下是 **先評估 i 的值給 cout，之後再遞增 i**。

而在第5行，因為因為我們把 `++` 寫在 `i` 前面，所以當下是 **先遞增 i，再評估 i 的值給 cout**。

如果牽涉到指定(assign)運算時也是一樣。

```
int i=1;
int a;
```

```
a = i++;
cout << a << endl;  // 1
cout << i << endl;  // 2
a = ++i;
cout << a << endl;  // 3
cout << i << endl;  // 3
```

遞減運算子的運作方式相同就不再贅述。

⚠ 由於遞增遞減運算子使用在複雜的指定敘述中，很容易讓人在閱讀時搞錯評估時機和實際指定過去的值。所以建議只在很單純，絕對不會搞錯的地方使用。否則寧可用 (i+1) 或 (i-1) 這樣明確的寫法。

4.4 for 迴圈

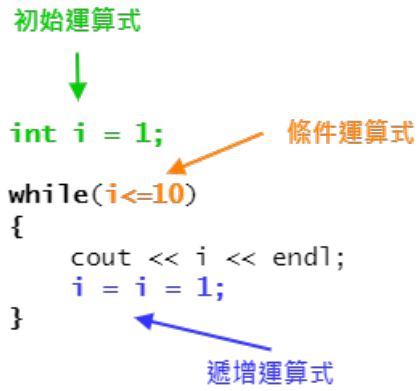
while 和 do...while 迴圈很適合用在「你知道什麼條件下迴圈要繼續或停止」，因為決定是否再繞一圈的就是一個條件判斷式。

但是在你很清楚一共要繞幾圈的情況下，使用接下來介紹的 for 迴圈，會輕鬆很多。

for 迴圈

使用 while 迴圈來繞指定圈數，我們多採用這樣的架構，其中變數 i 擔任計數器，我們會：

1. 指定計數器的初始值
2. 每圈檢查計數器的值是否仍符合條件
3. 每圈遞增計數器的值



for 迴圈可以一次搞定這三者。

for 的基本語法

```
for(初始運算式;條件運算式;遞增運算式)
{
    ......
    要重覆做的事情
    ......
}
```

以輸出 1~10 為例，程式看起來比較簡潔，而且還是很清晰。

練習：輸出 1 ~ 10

```
for(int i=1; i<=10; i=i+1)
{
    cout << i << endl;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

練習：輸出 n 的所有正因數

因為 n 的所有正因數是 1~n 之間的整數，所以我們用一個 for 迴圈來遍歷整個區間做篩選。

```
int n = 16;

cout << n << "的正因數有：";

for(int i=1; i<=n; i++)
{
    if(n%i==0)
    {
        cout << " " << i;
    }
}
cout << endl;
```

```
16的正因數有： 1 2 4 8 16
```

變數的生命週期

輸入以下這段程式後編譯執行，在編譯時期就會發生錯誤。

```
#include <iostream>

using namespace std;

int main()
{
    for(int i=1; i<=5; i++)
    {
        cout << i << endl;
    }

    cout << "now i=" << i << endl;

    return 0;
}
```

```
12:25: error: 'i' was not declared in this scope
    cout << "now i=" << i << endl;
                        ^
```

錯誤訊息表示在 12 行那邊使用到變數 i 但是沒有宣告。但是你往上看會覺得「明明我在第7行，for 迴圈那裡一開始就宣告了啊」。

仔細看一下錯誤訊息第一行末的 - "in this scope"，他是說你沒有在這個 scope 裡宣告 i。這個 scope 是什麼意思呢？

我們來看一下這個程式：

```
#include <iostream>

using namespace std;

int main()
{
    {
        int i=5;
        cout << "1: i=" << i << endl;
        i=i+1;
    }

    cout << "2: i=" << i << endl;

    return 0;
}
```

第 8, 9, 10 行被放在一組大括號裡，整個大括號範圍可以視為一個程式區塊(block)。這個區塊算是一個 **區塊範圍**

(block scope)，宣告在這個區塊裡的變數屬於 **區域變數(local variable)**，該變數的生命週期始於宣告完成，終於離開區塊。

所以在第 8 行開始，到第 11 行結束的範圍內，都可以存取變數 i 的值。但是在第 12 行開始，或第 7 行之前，都看不到也無法存取這個變數 i 的值。

試著編譯並執行這個程式，你會發現第 9, 10 行存取 變數i 都沒有問題，但是第 13 行會發生編譯錯誤，編譯器會抱怨變數 i 沒有在這個 scope 裡宣告。

若是把大括號拿掉，變成這樣。

```
#include <iostream>

using namespace std;

int main()
{
    int i=5;
    cout << "1: i=" << i << endl;
    i=i+1;

    cout << "2: i=" << i << endl;

    return 0;
}
```

現在整個程式只剩下一個 block，即第 6 ~ 14 行。所以變數 i 的生命週期始於第 7 行，終於第 14 行。程式輸出結果如下。

```
1: i=5
2: i=6
```

for 敘述(包含整個大括號範圍)也是一個 block scope，所以如果我們在 for 裡面宣告變數，它的生命週期也只限於該 for 迴圈內。

一般來說若只是單純用於迴圈的計數器，我們會像這樣把它宣告在 for 敘述裡。

```
    for(int i=1; i<=5; i++)  // 宣告在 for 敘述裡面
    {
        ......
    }
```

若是該變數在迴圈結束之後還有用處，我們會把它宣告在 for 迴圈的外面。

```
#include <iostream>

using namespace std;

int main()
{
    int i;  // 宣告在 for 敘述外面

    for(i=1; i<=5; i++)
    {
        cout << i << endl;
    }

    cout << "now i=" << i << endl;

    return 0;
}
```

```
1
2
3
4
```

```
5
now i=6
```

關於 scope 的詳細說明，有興趣的話可以先看一下這份文件 - scope。以後我們會另外開一個主題做更全面的討論。

Online judge 讀取 n 筆測資

在競程的題目中，有一種測資型式是這樣的。

輸入說明：

輸入的第一行有一個整數 t。接下來的 t 行每行有一個正整數 y，代表西元年份。

範例輸入：

```
4
1992
1993
1900
2000
```

這種情形就很適合使用 for 迴圈來讀取測資。

```
int n;
cin >> n;

for(int i=0; i<n; i++)
{
    int year;
    cin >> year;
    // do something
}
```

4.5 巢狀迴圈

多層迴圈

如同 if ... else 可以有多層結構，迴圈也可以有多層結構。多層迴圈是什麼樣子呢？我們以時鐘的時針、分針為例來說明。

分針和時針各是一個迴圈，分針 0~59，時針 0~11。

分針會由 0 分 轉到 59 分，接下來轉到 60 分時，時針會前進一格，分針則歸零重新開始新的一圈。

```
for(int hour=0; hour<12; hour++)  // 外圈是時針
{
    for(int minute=0; minute<60; minute++)  // 內圈是分針
    {
        cout << hour << ":" << minute << endl;
    }
}
```

1. 一開始外圈的 hour 是 0
2. 進入迴圈的主體 (3~6行)
   1. 內圈的 minute 一開始是 0
   2. 進入迴圈的主體 (第5行)
      1. minute 一邊遞增，一邊把第 5 行執行 60 次
   3. 內圈執行完畢
3. hour 遞增 1
4. 再次進入迴圈的主體 (3~6行)
   1. 內圈的 minute 一開始是 0
   2. 進入迴圈的主體 (第5行)
      1. minute 一邊遞增，一邊把第 5 行執行 60 次
   3. 內圈執行完畢
5. ……

程式執行後的輸出如下：

```
0:0
0:1
0:2
0:3
.
.
.
0:59
1:0
1:1
1:2
.
.
.
11:57
11:58
11:59
```

練習：3x6 星號矩陣

```
******
******
******
```

在這個練習中，我們要輸出如上的一個 3x6 星號矩陣

看到「重覆」的部分，我們很直覺的會想用迴圈來簡化程式碼。如果只會單層迴圈，可能這樣處理。

```
for(int i=0; i<3; i++)
{
    cout << "******" << endl;
}
```

迴圈內的 6 個星號，依然是「重覆」的狀態，所以它也可以使用迴圈來輸出。於是我們再加一個內層迴圈，讓它來輸出那 6 顆星號。

```
for(int i=0; i<3; i++)
{
    for(int j=0; j<6; j++)
    {
        cout << "*";
    }
    cout << endl;
}
```

請注意換行的 `cout << endl;` 放在什麼位置。想想看為什麼要放在這裡，而不是放在內層迴圈裡。

在這個例子裡，使用迴圈來處理重覆的工作，同時也讓程式變得有彈性。如果今天我們要輸入任意正整數 m, n 指定的 m x n 星號矩陣，只要將 3, 6 替換成變數 m, n 即可，其他程式碼都無需更動。

練習：m x n 星號矩陣

m = 2
n = 5
*****
*****

```
int m, n;

cout << "m=";
cin >> m;
cout << "n=";
cin >> n;

// 以下修改之前的雙層迴圈程式碼
for(int i=0; i<m; i++)
{
    for(int j=0; j<n; j++)
    {
        cout << "*";
    }
    cout << endl;
}
```

練習：輸出 n 階數字方陣

n=3

111
222
333

n=5

11111
22222
33333
44444
55555

———

有時候 for 敘述首行的變數不是單純只當計數器，也會參與到迴圈內的運算或輸出。所以在設計起迄數值時，我們會花點心思考量。

```
int n;
cin >> n;

for(int i=1; i<=n; i++)  // 一共有 n 列資料要輸出。(為什麼 i 由 1~n，而非如之前用 0~n-1？)
{
    for(int j=0; j<n; j++)  // 每列要輸出 n 個數字
    {
        cout << i;      // 要輸出的數字為當下的 i 值
    }
    cout << endl;
}
```

練習：n 階星號階梯

n=3

```
*
**
***
```

n=5

```
*
**
***
****
*****
```

———

在這個例子裡，外層迴圈的 i 除了幫外層計數，同時也是內層計數的終點值。

```
int n;
cin >> n;

for(int i=1; i<=n; i++)  // 一共有 n 列資料要輸出。(為什麼 i 由 1~n，而非如之前用 0~n-1？)
{
    for(int j=0; j<i; j++)  // 每列要輸出 i 個 *
    {
        cout << "*";
    }
    cout << endl;
}
```

下面這題給大家自己挑戰一下。

練習：n 階數字階梯

n=3

```
1
22
333
```

n=5

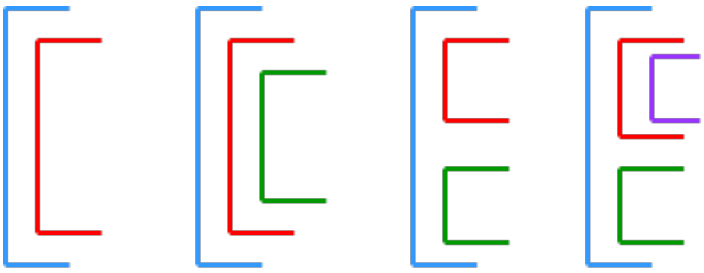
```
1
22
333
4444
55555
```

———

可以有的組合

多層迴圈可以由 while, do...while, for 迴圈任意組成。例如：外圈是 while，內圈是 for.....等等。

至於迴圈的結構也可以有多種變化，例如以下這幾種。



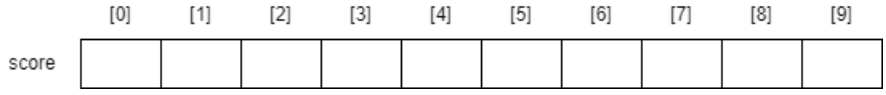
05-陣列

5.1 一維陣列

陣列(Array)的結構

陣列這種資料結構是用來儲存許多相同型別的資料用的。如果我們把變數想像成是一個可以放東西的箱子，那麼陣列就是一堆箱子的集合，而且每個箱子都有一個連續編號的索引值(index)。

例如：我們要儲存 10 個學生的成績(都是整數)，我們可以使用這樣一個內含 10 個元素(element)/項目(item)的陣列。



宣告陣列

在程式中我們可以這樣宣告這個陣列 score。

```
int score[10];
```

其語法為

```
型別 陣列名稱[元素數量];
```

其中陣列名稱的命名規則與一般變數的命名規則相同。

要特別注意的是，陣列的索引值是由 0 開始，所以宣告大小為 10 的陣列 score。可以使用的元素是 `score[0]` ~ `score[9]`。

給定初值

如同變數可以在宣告同時給定初值，陣列也可以。

如果只宣告，但不給定初值，則陣列內各元素的值會無法預測(會是分配到的記憶體當下的值)。

```
int a[5] = {1, 3, 5, 7, 9};

for(int i=0; i<5; i++)
{
    cout << a[i] << " ";
}
```

```
1 3 5 7 9
```

初值不給足

如果陣列有 5 個元素，但是初值只給 2 個，剩下 3 個的值會是什麼？

```
int a[5] = {1, 3};

for(int i=0; i<5; i++)
{
    cout << a[i] << " ";
}
```

```
1 3 0 0 0
```

觀察執行結果，可以發現它們被設為 0。

所以對於整數陣列，我們常用這樣的技巧來宣告並指定其初值皆為 0。

```
int a[5] = {0};

for(int i=0; i<5; i++)
{
    cout << a[i] << " ";
}
```

```
0 0 0 0 0
```

讓編譯器幫你算數量

我們可以在宣告時給初值但不指定陣列大小，編譯器會幫你算好填入。

```
int a[] = {1, 3, 5, 7};  // 相當於 int a[4] = {1, 3, 5, 7};
```

存取陣列中指定元素的值

原則上存取陣列 a 裡索引為 i 的元素值，和一般變數一樣，只要用 a[i] 來表示要存取的元素即可。

練習：讀取學生成績，並接受查詢

讀取使用者輸入的 1~10 號學生成績，並接受以座號查詢其成績。輸入 0 結束程式。

```
int score[10] = {0};

for(int i=0; i<10; i++)
{
    cin >> score[i];
}

while(true)
{
    cout << "輸入座號查詢成績: ";
    int id;
    cin >> id;
    if(id==0)
        break;
    cout << id << " 號的成績為 " << score[id-1] << endl;  // 想一想，為什麼索引值是 id-1，而不是 id？
}
```

```
11 22 33 44 55 66 77 88 99 100
輸入座號查詢成績:3
3 號的成績為 33
輸入座號查詢成績:6
6 號的成績為 66
輸入座號查詢成績:0
```

因為陣列的索引值是由 0 開始編號，和我們一般生活中由 1 開始編號的情境不同。所以也有人會選擇「浪費一個元素」來讓程式寫起來比較直覺。

```
int score[11] = {0};  // 宣告 11 個，索引 0 那個不用

for(int i=1; i<=10; i++)  // i 由 1~10，而不是 0~9
{
    cin >> score[i];
}

while(true)
{
    cout << "輸入座號查詢成績: ";
    int id;
```

```
    cin >> id;
    if(id==0)
        break;
    cout << id << " 號的成績為 " << score[id] << endl;  // id 不用減 1 了
}
```

陣列大小在宣告後無法改變

陣列大小在宣告後無法改變，所以通常我們會宣告「足夠」的大小。例如：在班級成績儲存時，若班級人數不超過 50 人，我們會宣告大小為 50 的陣列。

練習：讀取學生成績，並接受查詢(n 人版)

讀取使用者輸入的 1~n 號學生成績，並接受以座號查詢其成績。輸入 0 結束程式。 班級人數不超過 50 人。

輸入說明：

- 輸入的第一行為正整數 n，表示接下來有 n 個整數，分別代表 1~n 號學生的成績。

```
int score[50] = {0};  // 足夠的大小
int n;
cin >> n;

for(int i=0; i<n; i++)
{
    cin >> score[i];
}

while(true)
{
    cout << "輸入座號查詢成績: ";
    int id;
    cin >> id;
    if(id==0)
        break;
    cout << id << " 號的成績為 " << score[id-1] << endl;
}
```

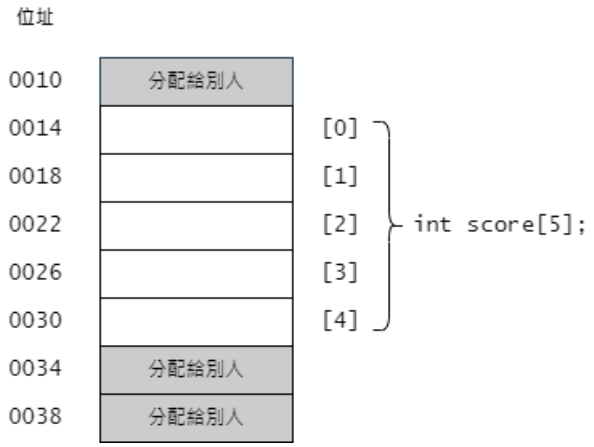
為什麼陣列的大小不能在程式執行中動態改變呢？

這可能跟陣列的特性有關，陣列有以下的特點：

1. 所有元素都是相同型別
2. 所有元素在記憶體中相鄰緊密排列
3. 可以依索引值快速隨機存取(無需循序)任一內部元素

其中 3 是因為 1, 2 才有辦法做到的。

以下面這個陣列為例：



因為每個元素都是 int，也就是佔記憶體的大小都是 4 byte。所以只要有陣列的開頭位址 $ 0014 $，和索引 $ i $，就可以知道 score[i] 在記憶體裡的位址為 $ 0014+i*4 $。

如果我們可以在記憶體中另外找 5 個 int 大小的空間給 score 來讓它的 size 由 5 變成 10。則這兩塊不連續的空間便無法再擁有原來設計的高速隨機存取優勢。

C99 的可變長度陣列(VLA)

上一個練習題，你會不會很想要這樣寫呢？

```
int n;
cin >> n;  // 先知道 n 的值

int score[n] = {0};  // 再宣告大小剛好為 n 的陣列

for(int i=0; i<n; i++)
{
    cin >> score[i];
}

......
```

實際寫下去執行，會發現還真的可以成功，這是為什麼呢？

C 語言的 C99 標準，支援可變長度陣列 (VLA, variable-length array) 。所以我們可以像上面那樣在執行中宣告一個以變數指定大小的陣列(但是之後就固定那個大小)。

由於我們使用的 C++ 編譯器 gcc 使用 extention 支援了 VLA，所以也可以做到。但是這個並不是 C++ 標準裡的東西，也就是並非所有的 C++ 編譯器會支援，你的程式碼可能在別的環境下會編譯失敗。

安全性問題

看一下以下的程式碼，預測他的輸出結果。

```
int numberOfStudent = 6;
int score[6];

for(int i=1; i<=6; i++) {  // 依序輸入 10 20 30 40 50 60
    cin >> score[i];
}
cout << numberOfStudent << endl;
```

使用 Code::Blocks 預設的 32 位元編譯器來編譯執行後，令人意外的，我們在程式裡根本沒有動到 numerOfStudent，但是輸出時卻發現 numerOfStudent 已經由 6 變成 60 了，Why？

因為程式裡宣告了 `int score[6];`，理應只使用 `score[0]`~`score[5]`，但是我們誤操作為 `score[1]`~`score[6]`。而 `score[6]` 推算出來的位址正好是 numberOfStudent 所在的位置。

編譯器不會提出警告，因為這是合法的操作(雖然在這情境下不合理)。程式設計師要自己負責做這種邊界檢查。

這讓 C++ 的程式變得容易有安全弱點，所以近年來有人提議使用會自己做記憶體管理和邊界檢查的程式語言。但是相對的就要付出一定的性能做為代價。

競賽可能遇到的問題

在競賽時為了搶時間求效能，我們常會宣告一個很大的陣列，而不是在那斤斤計較的省記憶體。

下面個程式可以成功編譯，但是執行後就直接 crash，連第一行 cout 都沒執行到。

```
#include <iostream>

using namespace std;

int main()
{
    int dat[200000001];  // 宣告在區域(local)

    cout << "Input a integer:";
    cin >> dat[200000000];
    cout << dat[200000000] << endl;

    return 0;
}
```

結束時返回的狀態碼是 **Process terminated with status -1073741571** (stack over flow)。在比賽時你可能會得到的訊息是 **segmentation fault**。

如果不要把它宣告在 main 函數內，而是宣告存全域區，讓它成為全域變數則可以成功執行。

```
#include <iostream>

using namespace std;

int dat[200000001];  // 宣告在全域區(global)

int main()
{
    cout << "Input a integer:";
    cin >> dat[200000000];
    cout << dat[200000000] << endl;

    return 0;
}
```

差別在哪裡呢？宣告在 local 的話，會用 stack 裡的記憶體來配置給它，而預設的 stack 都不大，可能只有幾 MB。而宣告在 global，則會配置在 data segment 裡，有更大的空間可用。

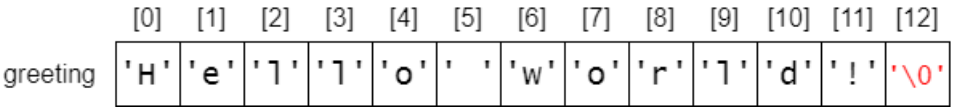
所以在比賽時，如果沒有變數污染的顧慮，宣告在 global 會比較好。

5.2 字串

字串是字元的陣列

字串可以被視為一個字元型別的一維陣列，例如： " Hello world! " 在記憶體中是這樣一個一個字元儲存的。

```
#include <iostream>

using namespace std;

int main()
{
    char greeting[13] = "Hello world!";

    cout << greeting;

    return 0;
}
```



在上圖中最後一個字元 `'\0'` 是什麼呢？這個是所謂的 `null` 字元。

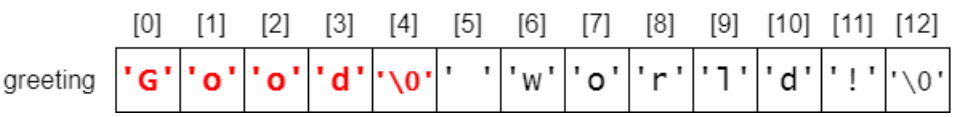
因為每個字串的長度是不一定的，`cout << greeting;` 中，傳給 cout 的其實只是整個字串開頭在記憶體中的位址，cout 會逐個字元輸出，直到遇到 `null` 字元為止。

所以雖然 "Hello world!" 只有 12 個字元長，但是我們準備了長度為 13 的 char 型別陣列來儲存它。

如果我們把程式改成這樣。

```
#include <iostream>

using namespace std;

int main()
{
    char greeting[13] = "Hello world!";

    cout << greeting;  // 輸出: "Hello world!"

    cin >> greeting;  // 輸入: "Good"

    cout << greeting;  // 輸出: "Good"

    return 0;
}
```

在第 11 行輸入 "Good" 之後，greeting 陣列的內容會變成這樣。



輸入的字串被存放在 greeting 裡，而且最後被加上一個 `null` 字元。

如果我們在第 11 行輸入的是 "This_is_a_test_for_a_very_long_string."，想想看會發生什麼事情？

我們輸入的字串會覆蓋掉原來在 greeting 陣列後面的一大堆值。(加底線 _ 是因為 cin 預設讀取到空白或換行字元就會停。)

我們更常用的是 C++ 的 string 型別

如前所述，在不知道別人會輸入多長的資料下，要準備多長的字元陣列才夠？這對系統安全來說是個很重要的問題。

以前在 C 語言裡，我們要想辦法來處理這個問題，而在 C++ 裡現在有一個很方便的字串型別 string 可用。你可以很安全的這樣使用它。(按規矩，需要先 `#include <string>` ，才能使用 string。但有的編譯器只要你 `#include <iostream>` ，就會include string，所以不寫也有可能會過，但寫了一定不會錯)

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string greeting = "Hello world!";

    cout << greeting;  // 輸出: "Hello world!"

    cin >> greeting;  // 輸入: "This_is_a_test_for_a_very_long_string."
                      // 很安全，不會覆蓋到其他資料
    cout << greeting;  // 輸出: "This_is_a_test_for_a_very_long_string."

    return 0;
}
```

string 不是 int, float, double, char ...這種原生資料型別(Primitive Data Types)。他是用 C++ 寫出的一個類別(class)，所以擁有比原生資料型別更多的能力。

取得 string 字串長度

使用 string 的 `length()` 方法(method)，可以取得 string 內儲存字串的長度。

```
    string str;

    str = "abc";
    cout << str.length() << endl;  // 3

    cin >> str;  // 輸入 Memory
    cout << str.length() << endl;  // 6
```

練習：Reverse output - 反向輸出字串

讀取一個不含空白字元的字串，反向輸出它。

範例輸入：

Hello

範例輸出：

olleH

要反向輸出字串，我們需要知道該宇串的長度，才能使用索引值，將它一個一個字元反向輸出。

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str;
```

```
    cin >> str;

    for(int i=str.length()-1; i>=0; i--)  // 最後一個字元的索引是 str.length()-1
    {
        cout << str[i];
    }
    cout << endl;

    return 0;
}
```

```
Hello
olleH
```

練習：Reverse a string - 反向一個字串

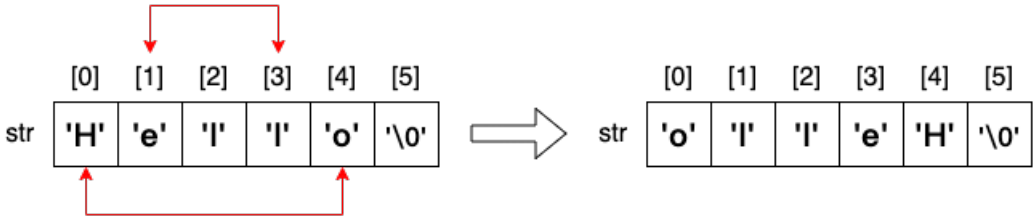
讀取一個不含空白字元的字串，**真的將其內容反向** 後再輸出。

範例輸入：

Hello

範例輸出：

olleH




```
#include <iostream>

using namespace std;

int main()
{
    string str;

    cin >> str;

    cout << "Before reverse: [" << str << "]" << endl;

    int len = str.length();  // length of str
    for(int i=0; i<len/2; i++)
    {
        char ch = str[i];
        str[i] = str[len-i-1];
        str[len-i-1] = ch;
    }

    cout << "After reverse:  [" << str << "]" << endl;

    return 0;
}
```

```
Hello
Before reverse: [Hello]
After reverse:  [olleH]
```

組成字串的字元，在記憶體裡也就是數字而已

組成字串的字元，在記憶體裡也就是數字而已，操作這些數字可以做出一些很有意思的事情。

> 上網查一下 ASCII，看看每個英文字元對應的編碼數值為何。 https://zh.wikipedia.org/zh-tw/ASCII

仔細觀察一下，大寫字母的字碼加上 32 就是小寫字母的字碼。

A	B	C	D	E	F	...	W	X	Y	Z
65	66	67	68	69	70	...	87	88	89	90

a	b	c	d	e	f	...	w	x	y	z
97	98	99	100	101	102	...	119	120	121	122

大小寫轉換

練習：to lower - 把英文單字轉換成全部小寫

讀取一個不含空白字元的字串，將其中的大寫字母都改成小寫。

範例輸入：

YouTube

範例輸出：

youtube

```
#include <iostream>

using namespace std;

int main()
{
    string str;

    cin >> str;

    cout << "Before: [" << str << "]" << endl;

    int len = str.length();  // length of str
    for(int i=0; i<len; i++)
    {
        if(str[i]>='A' && str[i]<='Z') // 這樣比大小是可以的，因為每個字元都是數字
        {
            str[i] = str[i]+32;
        }
    }

    cout << "After:  [" << str << "]" << endl;

    return 0;
}
```

```
YouTube
Before reverse: [YouTube]
After reverse:  [youtube]
```

大家可以自己試試看

- 全部轉大寫

- 大寫小寫互換

char <--> int

既然字串裡的字元，其實都是以數值方式儲存在記憶體中。如果我們想把字串 "abcdefg" 的字碼像這樣依序列出。

```
97 98 99 100 101 102 103
```

是不是這樣就可以了？

```
    string str = "abcdefg";

    for(int i=0; i<str.length(); i++)
    {
        cout << str[i] << " ";
    }
    cout << endl;
```

不行！我們得到這個。

```
a b c d e f g
```

因為 cin 判別 str[i] 是一個字元(char)，所以會把它以字元方式輸出。

必須讓 cin 把它視為 int 才能順利輸出數值。

我們可以使用 `int( )` 強制將 char 轉型為 int。

```
    string str = "abcdefg";

    for(int i=0; i<str.length(); i++)
    {
        cout << int(str[i]) << " ";  // 強制轉型為 int
    }
    cout << endl;
```

這樣就沒問題了。

```
97 98 99 100 101 102 103
```

反過來也可以用 `char( )` 把 int 強制轉型為 char。要注意的是 char 的大小為 1 Byte，所以只能接受 0~255。

```
    int data[7] = {97, 98, 99, 100, 101, 102, 103};

    for(int i=0; i<7; i++)
    {
        cout << char(data[i]);  // 強制轉型為 char
    }
    cout << endl;
```

輸出結果如下：

```
abcdefg
```

讀取一整行

在之前的例子中，我們無法輸入 "This is a test." 這樣的句子。因為 cin 在讀取 "This" 之後遇到空白字元，就中斷讀取。也就是一次只能讀進一個單字。

我們試一下這個程式

```
#include <iostream>
```

```
using namespace std;

int main()
{
    string line;
    int i = 1;

    while(cin >> line)
    {
        cout << i << ": " << line << endl;
        i++;
    }

    return 0;
}
```

輸入以下兩行字串

Hello world!
This is a test.

我們得到的輸出會是

```
1: Hello
2: world!
3: This
4: is
5: a
6: test.
```

使用 getline 讀取一行

使用 `getline()` 函數可以讀入一整行的字串，也就是讀到換行為止。

```
#include <iostream>

using namespace std;

int main()
{
    string line;
    int i = 1;

    while(getline(cin, line))
    {
        cout << i << ": " << line << endl;
        i++;
    }

    return 0;
}
```

同樣的輸入，這次的輸出為

```
1: Hello world!
2: This is a test.
```

cin 和 getline 搭配使用會遇到的問題

如果題目的輸入是這樣，第一行是 3 表示接下來有 3 行字串。

3
Hello, world.
This is a test.
Good morning

使用以下的程式讀取後，依序輸出各行字串。

```
#include <iostream>

using namespace std;

int main()
{
    string line;

    int n;
    cin >> n;

    for(int i=0; i<n; i++)
    {
        getline(cin, line);
        cout << line << endl;
    }

    return 0;
}
```

我們預期的輸出是

```
Hello, world.
This is a test.
Good morning
```

實際得到的是

```

Hello, world.
This is a test.
```

前面多一行空白行，後面少一行 "Good morning"

原因如下：

- 我們的輸入包含按下的[Enter]是長這個樣子

3`\n`Hello, world.`\n`This is a test.`\n`Good morning`\n`

- 第 10 行的 cin >> n; 會把 3 讀進 n，於是剩下

`\n`Hello, world.`\n`This is a test.`\n`Good morning`\n`

- 接下來第 14 行的 getline(cin, line); 會把一行字串讀入 line 中，但是因為一開始就遇到換行，於是 line 裡面是個空字串 ""。但迴圈已繞了一圈，現在剩下的是

Hello, world.`\n`This is a test.`\n`Good morning`\n`

- 所以接下來第二圈的 getline 讀完一行後，剩下

This is a test.`\n`Good morning`\n`

- 最後一圈的 getline 讀完一行後，還剩下

Good morning`\n` 沒被讀取，也沒被印出。

解決的方式是，用 cin 讀完 3 這個整數後，想辦法把後面的換行字元 '\n' 也先讀取掉。

```
#include <iostream>

using namespace std;

int main()
{
    string line;
```

```
    int n;
    cin >> n >> ws;  // 注意這裡的 ws

    for(int i=0; i<n; i++)
    {
        getline(cin, line);
        cout << line << endl;
    }

    return 0;
}
```

請注意，在這裡的 ws 指的是 white space，意思就是把 空白/換行/tab 這些「空白」字元都先讀光光。

5.3 多維陣列

二維陣列

把索引值擴展為 2 維，我們就可以得到二維陣列。

一個大小為 m x n 的二維陣列，可以這樣宣告。

```
// 宣告一個 4 x 6 的 int 二維陣列
int A[4][6];
```

和一維陣列一樣，可以在宣告時給定初值。

```
int A[4][6] = {
  {1, 2, 3, 4, 5, 6},
  {5, 12, 7, 11, 9, 8},
  {10, 21, 13, 22, 23, 16},
  {4, 78, 13, 45, 51, 11}
};
```

陣列 A

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	1	2	3	4	5	6
[1]	5	12	7	11	9	8
[2]	10	21	13	22	23	16
[3]	4	78	13	45	51	11

A[1][3]

A[3][5]

搭配雙層迴圈遍歷其值

我們可以使用雙層迴圈，把前面那個二維陣列的值印出來。

```
for(int i=0; i<4; i++) {
  for(int j=0; j<6; j++) {
    cout << A[i][j] << " ";
  }
  cout << endl;
}
```

```
1 2 3 4 5 6
5 12 7 11 9 8
10 21 13 22 23 16
4 78 13 45 51 11
```

練習：2D 地圖

給定一張 m x n 大小的地圖，以及各地貌的代表數字，請輸出該地圖。

$ 1 \le m, n \le 100 $

輸入說明:

第一行是兩個正整數 m n，表示陣列的 列數(row)、行數(column)。

接下來是共 m 列，每列有 n 個整數的地圖資訊，表示該位置的地貌代碼。

接著是一個整數 k，表示有 k 種地貌。

最後是 k 列，每列為一個整數 i 與一個字元 c，表示代碼 i 的地貌為 c。

輸出說明:

輸出該地圖的地貌，如範例輸出。

範例輸入:

```
3 4
1 1 1 1
2 2 0 1
1 2 0 1
3
0 _
1 #
2 *
```

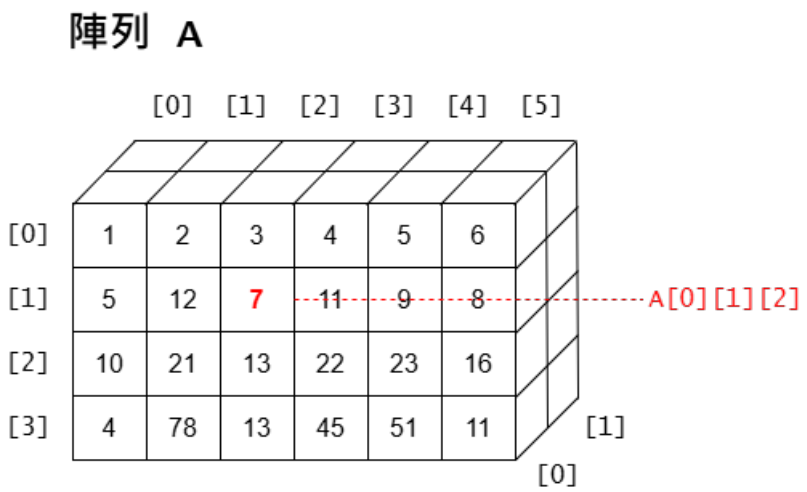
範例輸出:

```
####
**_#
#*_#
```

```
#include <iostream>

using namespace std;

int main()
{
    int m, n;
    cin >> m >> n;

    int M[100][100];  // 這樣比較安全，若 size 太大可考慮宣告在全域區
    // int M[m][n];   // C99 的 VLA 允許這麼宣告，但是若 size 太大會有問題

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            cin >> M[i][j];
        }
    }

    int k;
    cin >> k;
    int N[k];    // 地貌代碼
    char T[k];   // 地貌

    for(int i=0; i<k; i++) {
        cin >> N[i] >> T[i];
    }

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            for(int u=0; u<k; u++) {
                if(M[i][j]==N[u]) {    // 在 N 中找出代碼 M[i][j] 的位置 u
                    cout << T[u];      // 輸出地貌 T[u]
                    break;
                }
            }
        }
        cout << endl;
    }

    return 0;
}
```

多維陣列

如果把二維陣列想像成一個平面，那麼三維陣列就可以想像成一個長方體。



平常我們很少使用超過 3 維的陣列，除非你很確定自己需要，否則在你宣告一個大於 3 維的陣列之前，可以想一想，有沒有更好的方式可以不要用到這麼高維的陣列。

06-函數

6-1 函數

隨著寫程式經驗愈來愈多，你會發現有些程式碼會不斷重複出現，就像例行性工作一樣，例如：求平方根、將資料排序、驗證帳號密碼.....等等。一次又一次的輸入這些程式碼會讓人很不耐煩。對於這些經常出現的程式碼片段，我們可以使用函數來把它們包裝起來。C/C++裡面的函數就像數學裡面的函數，例如：

$f(x)=2x^2+3x+4$

它有一個輸入：x，有一個輸出：f(x)。你給它一個輸入 3，它在運算後會給你一個輸出31；你給它另一個輸入 2，它會給你另一個相應的輸出18。不管你給的輸入是什麼，它都會很忠實的去完成該做的事 $2x^2 + 3x + 4$ ，並把結果輸出給你。

定義函數

以上面那個 f(x) 為例，我們可以這樣在 C++ 裡定義它。

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

其架構如下：

```
回傳值型別 函數名稱(參數1型別 參數1名稱, 參數2型別 參數2名稱, ...)
{
    // 實作程式碼
    return 回傳值;
}
```

接下來我們就可以使用這個函數了。

```
#include <iostream>

using namespace std;

int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69

    return 0;
}
```

請注意我們把 函數f 定義在 main() 的前面。如同變數在使用前要先宣告，函數也是一樣。

我們在第 17 行首次使用到 函數f，所以在這之前必須先知道 函數f 長什麼樣子。

如果把 函數f 定義在後面，在編譯時就會發生錯誤。

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);  // f( ) 是什麼？往前看不到有人告訴我 f( ) 是什麼。
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69

    return 0;
}

// 定義在後面
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

```
main.cpp: In function 'int main()':
main.cpp:11:11: error: 'f' was not declared in this scope
    ans = f(n);
          ^
```

宣告函數

有沒有注意到，前面我們一直說定義函數，而不是宣告函數(declare)。

我們以同一個 函數f 為例，宣告這個函數的作法為：

```
int f(int x);
```

或

```
int f(x);
```

宣告函數只要講清楚這幾個重點即可：

1. 函數名稱
2. 參數列 (每個參數的型別，可以沒有名字)
3. 回傳值型別

我們把上面的範例程式改成只有宣告試試。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡
```

```
int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);    // 使用到 函數f
    cout << ans << endl;   // 18

    ans = f(3);    // 使用到 函數f
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69,  使用到 函數f

    return 0;
}
```

建置(build)這個程式時，出現了沒看過的錯誤。

```
main.cpp:13: undefined reference to `f(int)'
main.cpp:16: undefined reference to `f(int)'
main.cpp:20: undefined referenc
```

這個 `undefined reference to 'f(int)'` 是什麼意思呢？

我們的程式碼要經過「編譯(compile)」、「連結(link)」兩個步驟，才能生成最終的可執行檔。

在編譯階段，編譯器看到叫用(call)函數時，只會確認之前宣告過的函數

1. 名稱是否相符
2. 參數列的數量和型別是否相符
3. 回傳值型別是否相符

如果都符合，會在叫用函數的地方留個「空位」，然後編譯將會成功完成，進入連結階段。

在連結階段必須真的有一個函數被定義過，才能把這個函數「身體」所在的位置填入之前編譯階段留下的「空格」。

我們修改程式，在末端補上 函數f 的定義，即可順利建置。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);    // 使用到 函數f
    cout << ans << endl;   // 18

    ans = f(3);    // 使用到 函數f
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69,  使用到 函數f

    return 0;
}

// 定義在後面
int f(int x)
{
```

```
    int result = 2*x*x +3*x + 4;
    return result;
}
```

或許有同學會覺得把它拆成兩段一個放前面、一個放後面，不是多此一舉嗎？

這個設計的考量是，我們在開發大專案時，不會把所有程式碼寫在同一個檔案裡，而是會分散在多個檔案裡。

如果有 10,000 行程式碼放在同一檔案裡，只要有一行修改，這 10,000 行都要重新編譯、連結、產出執行檔。

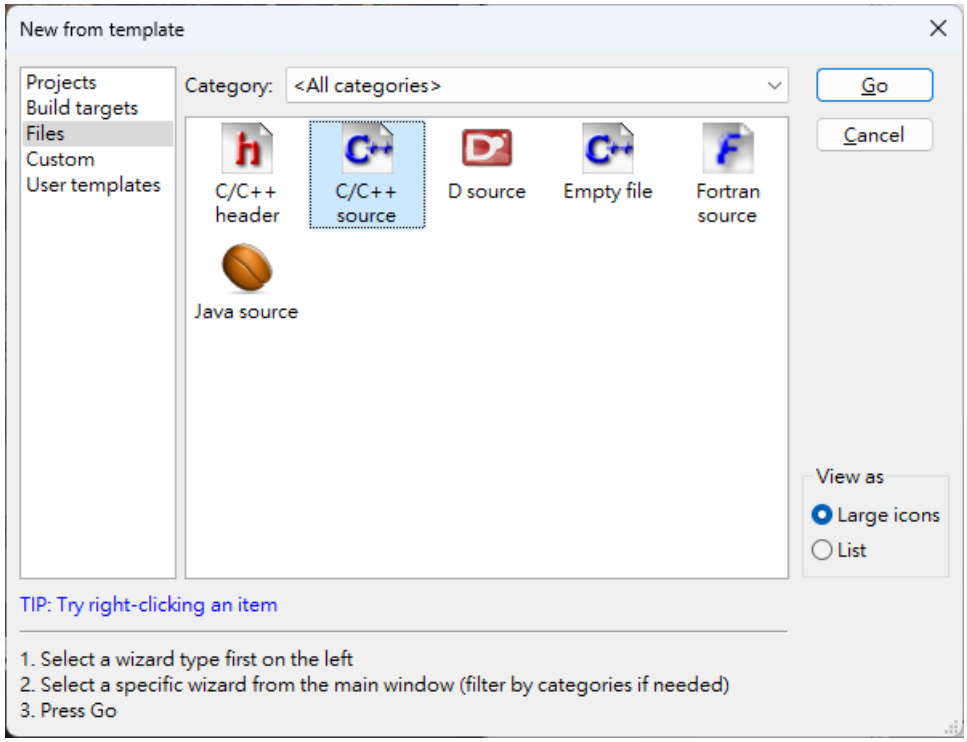
但若是把它拆成 10 個 1,000 行的檔案，當其中一行修改時，只有包含那行檔案 的 1,000 行需要重新編譯，然後把 10 個編譯後的檔案連結產出執行檔即可。

多檔案專案

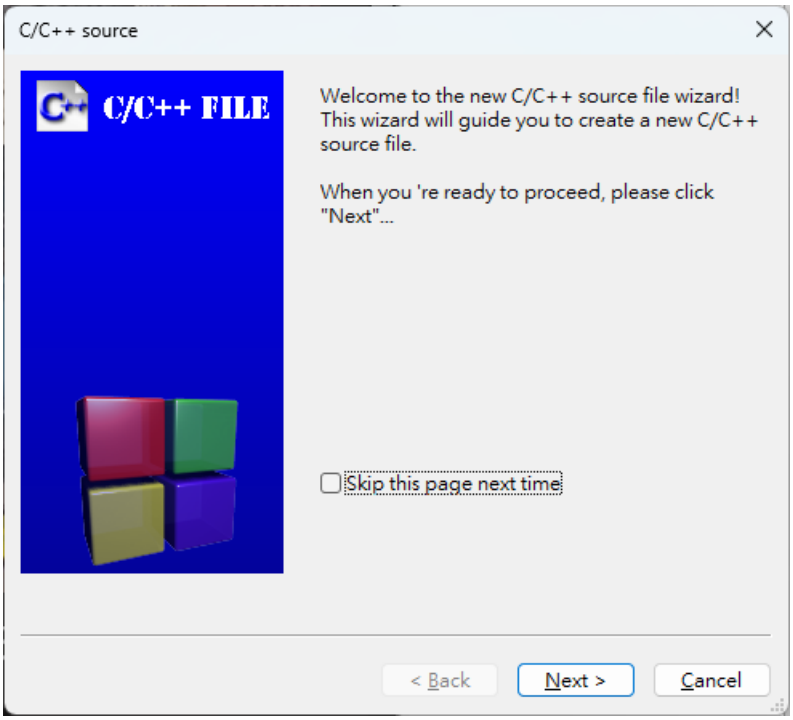
我們來實作一下把範例程式拆成兩個 .cpp 檔案。

目前我們有一個 main.cpp，接下來新增一個 myfuntion.cpp。

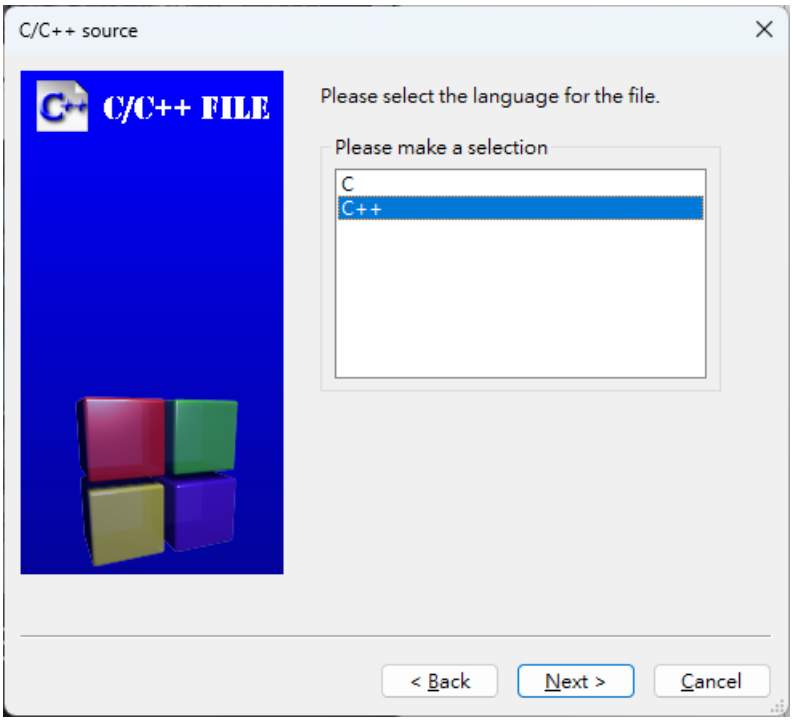
1. 首先依序點選 Code::Blocks 選單 [File]->[New]->[file...]
2. 選擇 [C/C++ source]->[Go]



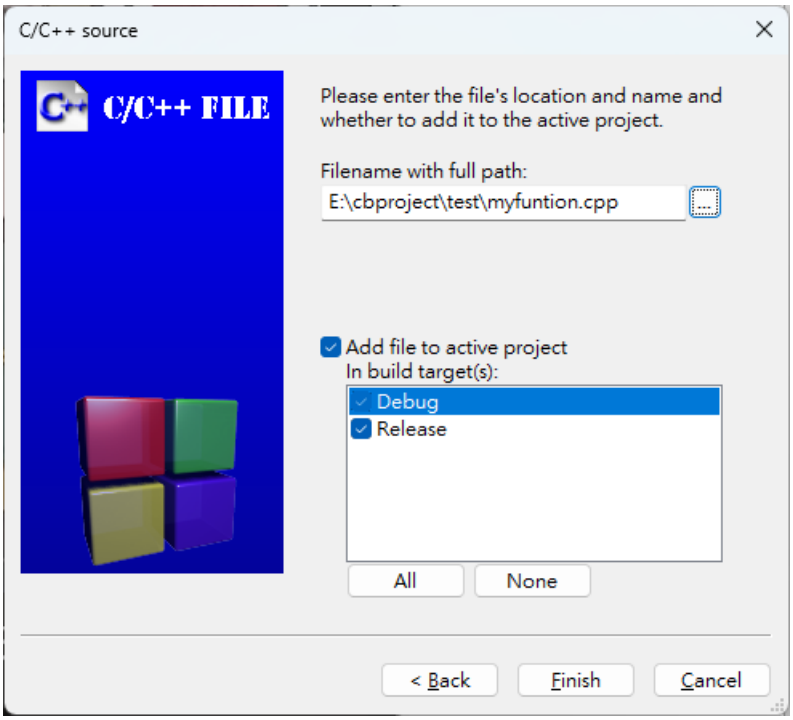
[Next]



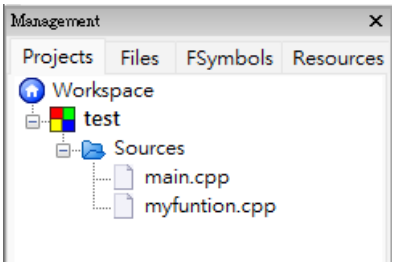
[Next]



點選 [...] 檔名輸入 "myfunction.cpp"，接著點選 [All]->[Finish]



3. 現在專案裡就可以多一個 function.cpp 檔了。



在 [function.cpp] 裡定義好函數f。

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

在 [main.cpp] 裡宣告函數f 並使用它。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69
```

```
    return 0;
}
```

試著建置並執行，應該可以順利完成。

[練習] 增加一個 $g(x) = x(x-1)$

在 [myfunction.cpp] 裡定義 g(x) 函數

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}

int g(int x)
{
  return x*(x-1);
}
```

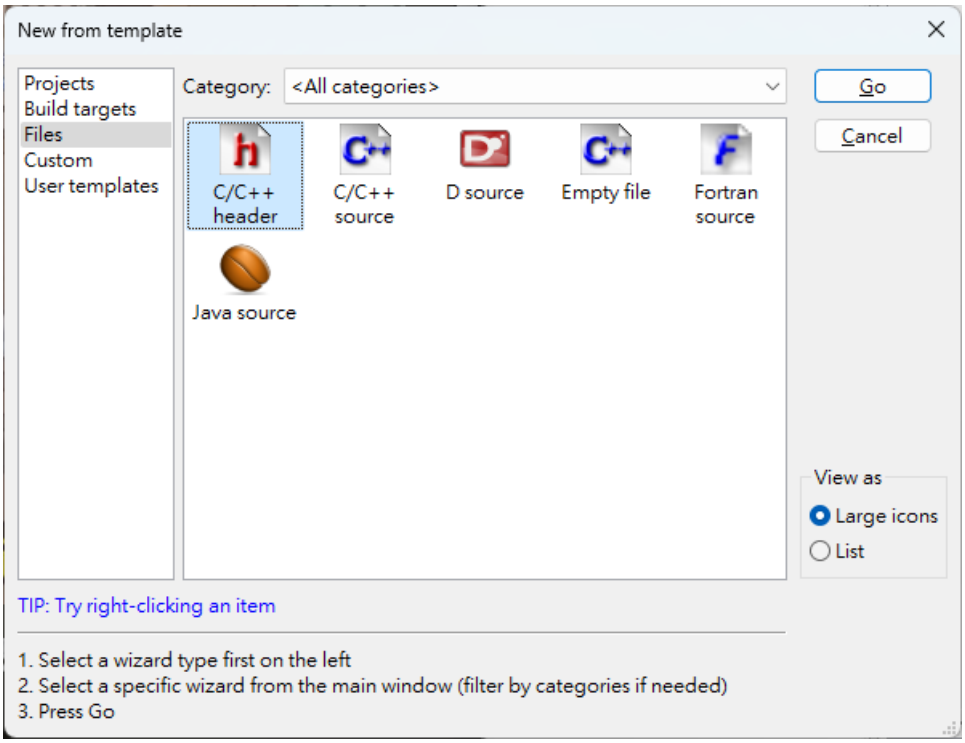
在 [main.cpp] 裡宣告並使用 g(x) 函數

```
#include <iostream>

using namespace std;

int f(int x);
int g(int x); // 宣告 g(x)

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;

    cout << g(3) << endl;  // 6, 使用 g(x)

    return 0;
}
```

標頭檔(header file)

隨著自己定義的函數愈來愈多，[main.cpp] 前面的宣告會愈來愈多行。我們可以把這些宣告移到另一個檔案裡。

類似之前我們新增 [C/C++ source]檔 的方式，這次我們新增一個 **[C/C++ header]** 檔，並命名為 "myfunction.h"。



把 [main.cpp] 裡的宣告移到 [myfunction.h] 裡。

```
int f(int x);
int g(int x);
```

在 [main.cpp] 裡引入(include)標頭檔 [myfunction.h] 。在編譯時，編譯器會把 myfunction.h 檔案的內容抄到這個引入的地方。

```
#include <iostream>
#include "myfunction.h"  // 引入標頭檔 myfunction.h

using namespace std;

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;

    cout << g(3) << endl;

    return 0;
}
```

建置並執行後，程式應該可以順利運行。

我們從一開始學 C++ 就在程式的開頭有一行 `#include <iostream>`。現在你應該可以了解它的作用了，它裡面放的就是和輸入、輸出相關的宣告。

至於為什麼它用角括號 `< >`，我們自己寫的用雙引號 `" "` 呢？

這跟標頭檔所在的位置有關，用角括號 `< >` 編譯器會去內建函式庫的資料夾找標頭檔，用雙引號 `" "` 編譯器會去目前這個專案的資料夾去找標頭檔。

6-2 在函數中使用函數

相同名稱的函數

原則上函數的名稱不能重覆，但是只要其參數列不同，就可以使用相同的名稱。

以下面的程式為例，我們可以觀察到叫用函數時，編譯器會檢查函數名稱和參數列數量和型別。

```
#include <iostream>

using namespace std;

// 回傳 2 整數中的最小值
int MIN(int a, int b)
{
    cout << "回傳 2 整數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 2 浮點數中的最小值
double MIN(double a, double b)
{
    cout << "回傳 2 浮點數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 3 整數中的最小值
int MIN(int a, int b, int c)
{
    cout << "回傳 3 整數中的最小值" << endl;
    if(a<=b && a<=c)
        return a;
    else if(b<=a && b<=c)
        return b;
    else
        return c;
}

int main()
{
    int x=2, y=5, z=3;
    double i=5.3, j=2.1, k=4.3;

    cout << MIN(x, z) << endl;       // 回傳 2 整數中的最小值
    cout << MIN(i, j) << endl;       // 回傳 2 浮點數中的最小值
    cout << MIN(x, y, z) << endl;    // 回傳 3 整數中的最小值

    return 0;
}
```

```
回傳 2 整數中的最小值
2
回傳 2 浮點數中的最小值
2.1
回傳 3 整數中的最小值
2
```

在函數中叫用函數

在前例中我們為了求 3 整數中的最小數，又另外寫了一個 3 參數的 MIN 函數，其內容也是整個重寫。

我們的另一種選擇是利用已寫好的 2 參數 MIN 函數，來實作出 3 參數的 MIN 函數。

```
#include <iostream>

using namespace std;

// 回傳 2 整數中的最小值
int MIN(int a, int b)
{
    cout << "回傳 2 整數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 3 整數中的最小值
int MIN(int a, int b, int c)
{
    cout << "回傳 3 整數中的最小值" << endl;
    return MIN(MIN(a, b), c);  // 利用 MIN(int , int)
}


int main()
{
    int x=2, y=5, z=3;

    cout << "Step 1:" << endl;
    cout << MIN(x, z) << endl;          // 回傳 2 整數中的最小值

    cout << "Step 2:" << endl;
    cout << MIN(x, y, z) << endl;       // 回傳 3 整數中的最小值

    cout << "Step 3:" << endl;
    cout << MIN(x, MIN(y, z)) << endl; // 回傳 3 整數中的最小值

    return 0;
}
```

由輸出結果我們可以看到，叫用 MIN(int , int, int) 時，MIN(int, int) 被叫用了 2 次。

```
Step 1:
回傳 2 整數中的最小值
2
Step 2:
回傳 3 整數中的最小值
回傳 2 整數中的最小值
回傳 2 整數中的最小值
2
Step 3:
回傳 2 整數中的最小值
回傳 2 整數中的最小值
2
```

練習：求 a, b 兩正整數的最大公因數(GCD)

設計一個 GCD 函數，求 2 正整數的最大公因數。

---

1. 用迴圈慢慢找

```
int GCD(int a, int b)
{
```

```
  if(a>b)
    swap(a, b);
  int ans = 1;
  for(int i=1; i<=a; i++) {
    if(a%i==0 && b%i==0) {
      ans = i;
    }
  }
  return ans;
}
```

2. 超級快的「輾轉相除法」

```
int GCD(int a, int b)
{
  int r;
  while(a%b>0) {
    r = a%b;
    a = b;
    b = r;
  }
  return b;
}
```

練習：求 a, b 兩正整數的最小公倍數(LCM)

設計一個 LCM 函數，求 2 正整數的最小公倍數。

利用之前的 GCD 函數

```
int LCM(int a, int b)
{
  return a/GCD(a, b)*b;
}
```

我們不使用 `a*b/GCD(a,b)` 的原因是，若先把 a*b ，其相乘後數值溢位的可能性更大，先把 a 除以兩數的公因數，再乘上 b，可以減低溢位的風險。

6-3 傳值呼叫 與 傳參考呼叫

參數與引數

在提到函數與呼叫使用函數時，我們會用到 **參數(parameter)** 和 **引數(argument)** 這兩個名詞。

我們可以簡單的用這張圖來區分他們。

- **參數(parameter)** 是在定義函數時，用來承接傳入資料的變數。
- **引數(argument)** 是在呼叫使用函數時，傳入的資料。

```
                parameter
int Max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}

int main()
{
                    argument
    cout << Max(3, 7);

}
```

然而在大多數的情況下，大家並不會區分的那麼清楚，很多時候我們都會用 **參數** 來意指兩者。在後續的內容裡除非特別需要指出其不同，否則我們會使用 **參數** 這個詞。

傳值呼叫(call by value)

在叫用函數時，我們通常都會傳入數個參數給該函數，例如底下這個求等差數列第 n 項的函數 An( )。

```
int An(int a, int d, int n)
{
  return a+(n-1)*d;
}

int main()
{
  cout << An(1, 2, 10) << endl;  // 19
  cout << An(2, 3, 5) << endl;   // 14

  return 0;
}
```

你可以這樣想像，在第 9 行叫用 An(1, 2, 10) 的時候

1. An 函數產生了 a, d, n 這三個變數，用來承接傳入的引數
2. a 接收到了 1, d 接收到了2, n 接收到了 10
3. 回傳 a+(n-1)*d 的運算結果
4. An 函數之前產生的 a, d, n 三個變數消滅不再存在
5. 返回叫用函數的地方(第9行)，繼續執行下去。

當第 10 行叫用 An(2, 3, 5) 的時候，以上流程會再發生一次。請注意 2 個重點：

1. a, d, n 都是區域變數，當 An() 被叫用時會產生一份區域變數，返回時這些區域變數就會消滅。
2. 叫用 An( ) 時，參數的「值」被複製一份給 a, d, n。所以我們叫它 **傳「值」呼叫 (call by value)**。

接下來這個 exchange 函數會讓你把這個機制的第2個重點看得更清楚。

```
void exchange(int a, int b)
{
  int t = a;
  a = b;
  b = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 3
  cout << "b = " << b << endl;  // b = 5

  return 0;
}
```

第 13 行叫用 exchange(a, b)時，在 main( ) 裡的 a, b 和 exchange( ) 裡的 a, b 是互不相關的。

外面的(main的) a, b 只是把它當下的值複製一份傳給**裡面的**(exchange的) a, b。

在函數裡的 a, b 在 t 的協助下互相交換其值，並且在離開函數回到 main 裡繼續執行前，函數裡的 a, b, t 都消滅了。

函數結束回到 main 裡，接著用 cout 輸出 a, b，這個被輸出的是 main 的 a, b。由於剛才互相交換值的是 exchange 函數內的 a, b，和現在 main 的 a, b 一點關係都沒有，所以輸出的 a 還是 3，b 還是 5。

傳參考呼叫(call by referance)

如果我們真的需要一個函數，能夠幫我們把外面的兩個變數值交換，必須使用 **「傳參考呼叫(call by reference)」**。

唯一不同的地方是在函數的參數列裡，把要被傳入的變數前面加上 `&`。

```
void exchange(int &a, int &b)
{
  int t = a;
  a = b;
  b = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 5
  cout << "b = " << b << endl;  // b = 3

  return 0;
}
```

使用傳參考時，你可以想像外面的變數真的被傳進去了，你在函數裡對它做什麼，實際上真的會作用在外面的變數上。

你也會看到有人會這麼描述傳參考呼叫 **「參考就是別名(alias)」**。用下面這個例子比較容易理解這個別名的概念。

我們把傳入的`引數 a` 取個`別名叫 c`，把傳入的 `引數b` 取個 `別名叫 d`。於是在函數裡提到的 c 實際上就是外面的 a，在函數裡提到的 d 實際上就是外面的 b。

```
void exchange(int &c, int &d)
{
```

```
  int t = c;
  c = d;
  d = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 5
  cout << "b = " << b << endl;  // b = 3

  return 0;
}
```

6-4 將陣列傳入函數

傳址呼叫(call by address)

除了「傳值呼叫」、「傳參考呼叫」外，還有一種參數傳遞方式叫「傳址呼叫」。

為什麼叫「傳址」呢？因為這種方式是直接把變數在記憶體中的 **「位址(address)」** 傳進去給函數，在函數裡我們直接到記憶體中的相應位置去操作這個變數的值。所以傳址呼叫和傳參考呼叫一樣可以動到外面變數的值。

關於傳址呼叫，因為會接涉到記憶體位置和指標(pointer)，比較複雜，我們會稍後再來看這個主題。

不過由於大家可能會有需要把一個陣列傳入函數裡，所以我們先來看要如何做到。

一個陣列裡面的元素可能會有非常多個，把它的值全部複製一份再傳給函數未免太浪費時間。由於陣列裡的每個元素都是相同型別，所佔記憶體大小相同，又在記憶體中連續緊密排列，所以 C/C++ 裡採取的是把陣列開頭的位址傳進去即可。

但是只有開頭，不知道陣列結束在哪裡，所以我們還得把陣列的長度也一併做為引數傳入。

範例：將陣列傳入函數

```
int showArray(int A[], int n)
{
  for(int i=0; i<n; i++)
  {
    cout << A[i] << " ";
  }
  cout << endl;
}

int main()
{
  int data[5] = {1, 3, 5, 7, 9};

  showArray(data, n);  // 1 3 5 7 9

  return 0;
}
```

```
1 3 5 7 9
```

6-5 全域變數與靜態變數

全域變數(Global variable)

一般來說，我們使用函數時會將操作到的變數限制在函數裡，也就是以區域變數的方式使用。如有需要操作到函數外面的變數，我們會用傳參考或傳址的方式來處理。

我們以一個抽號碼牌的程式來示範。

練習：抽號碼牌(1)

```
#include <iostream>

using namespace std;

int getTicket(int &num)  // 以傳參考方式遞增外面的 num 變數值
{
    num++;
    return num;
}

int main()
{
    int num = 0;  // 記錄目前發到幾號

    cout << "I have ticket No." << getTicket(num) << endl;
    cout << "I have ticket No." << getTicket(num) << endl;
    cout << "I have ticket No." << getTicket(num) << endl;

    return 0;
}
```

```
I have ticket No.1
I have ticket No.2
I have ticket No.3
```

使用這種方式沒什麼問題，但是每次都要傳遞變數 num。如果想避免這個麻煩，可以使用全域變數，也就是把 num 宣告在所有函數(包含 main)的外面。

練習：抽號碼牌(2)

```
#include <iostream>

using namespace std;

int num = 0;  // 記錄目前發到幾號。宣告在這裡是全域變數

int getTicket()  // 沒有參數
{
    num++;  // 因為 num 是全域變數，所以到處都可以存取它
    return num;
}

int main()
{
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數

    return 0;
}
```

```
I have ticket No.1
I have ticket No.2
I have ticket No.3
```

使用全域變數雖然很方便，但是它有一個極大的缺點，就是大家都可以動到它。

有時候你會很納悶，明明我沒動它，它的值怎麼變了。找了半天才發現在某個不起眼角落或函數裡的程式碼動到它的值。

函數裡的靜態變數(static variable)

一般來說宣告在函數裡的變數都是區域變數(local variable)，一但離開函數後就會消滅，下次被呼叫時才會重新產生出來。

但是如果在宣告時，在前面加上 `static` 修飾詞，它就會是個靜態變數，在離開函數時變數會記得當下的值，不會消滅。下次函數被呼叫時，它依然活著不會被重新產生和給定初值。

練習：抽號碼牌(3)

```
#include <iostream>

using namespace std;

int getTicket()  // 沒有參數
{
 static int num = 0;  // 靜態變數，只在程式開始時指定一次初值
    num++;
    return num;
}

int main()
{
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數

    return 0;
}
```

```
I have ticket No.1
I have ticket No.2
I have ticket No.3
```

在某些情況下，靜態變數是很好用的！

07-指標

7-1 指標(pointer)

記憶體-位址

當宣告一個變數並賦予它初值後，我們可以確定這個值一定存放在電腦記憶體的某個地方，問題是它到底放在哪裡呢？在地球表面上我們可以用經緯度來標定一個位置，而在電腦裡要標定記憶體中的某個位置則是要靠「位址(address)」

我們在寫程式時可以用 `cout << a` 來印出變數 a 的值，但大家必須了解背後的實際動作是將儲存 a 的那塊記憶體內容印出來。

眼尖的同學應該注意到了上圖中的位址每一個相差 4，這是因為我們以 int 型別的變數為例，而 int 的大小是 4 byte，所以每個 int 都要在記憶體中佔掉 4 byte 的空間。若是我們使用 double 型別，則每個變數都會佔掉 8 byte 的空間。

由於每次程式載入記憶體執行時可能都在不同的位置，因此這次變數 a 儲存在 0x22ff18 不表示下次執行時它也會儲存在 0x22ff18。使用取址(address-of)運算子 &可以取得變數目前在記憶體中的位址。

```
int a = 2, b = 3;
cout << &a << endl;
cout << &b << endl;
```

```
0x22ff18
0x22ff14
```

指標變數

在 C/C++ 中用來儲存位址的是一種特殊型別的變數 - 指標(pointer)變數

宣告

```
資料型別 *變數名稱; // 注意前面有個 * 號
```

範例

```
int *pNumber;         // 宣告一個名為 pNumber 的指標變數，用來指向一個 int 型別的變數

float *pF = nullptr; // 宣告一個名為 pF 的指標變數，用來指向一個 float 型別的變數，
                      //並給定指標的初值為 nullptr，即不指向任何地方的空指標。
```

取址(address-of)運算子 &

在變數名稱前加上一個取址運算子(&)可以取得該變數的位址。

```
int a = 6;
int *pA = nullptr;
pA = &a; // 取得變數 a 的位址並儲存在指標 pA 中
```

提領(dereference)運算子 *

在指標變數名稱前加上一個提領運算子 *， 可以 讀/寫 它所指向變數的值。

```
int a = 6, b = 5;
int *pNum = nullptr;

pNum = &a;
cout << *pNum << endl;
pNum = &b;
cout << *pNum << endl;
```

```
6
5
```

```
int a = 6, b = 5;
int *pNum = nullptr;

cout << "a = " << a << ", b = " << b << endl;
pNum = &a;
*pNum = 5;
pNum = &b;
*pNum = 6;
cout << "a = " << a << ", b = " << b << endl;
```

```
a = 6, b = 5
a = 5, b = 6
```

```
int a = 6, b = 5;
int *pNum1 = nullptr;
int *pNum2 = nullptr;

cout << "a = " << a << ", b = " << b << endl;

pNum1 = &a;
pNum2 = pNum1;
*pNum2 = 3;

cout << "a = " << a << ", b = " << b << endl;
```

```
a = 6, b = 5
a = 3, b = 5
```

動態配置記憶體

截至目前為止，我們的程式都在一開始就將需要使用的記憶體（如：變數、陣列）大小寫死在程式碼中。

```
int a = 0, b = 0; // 兩個 int 變數，共 2*4=8 byte
int score[50];    // 一個包含 50 個 int 的陣列，共 50*4=200 byte
```

但是有時候我們在寫程式時並不知道使用者執行時需要多大的空間。例如我們要寫一個讀入學生成績並依成績高低排序的程式，你可能會想這樣寫：

```
int numOfStd=0;
int score[50];

cout << "請輸入學生人數：";
cin >> numOfStd;

for(int i=0; i<numOfStd; i++) {
    cin >> score[i];
}
// 排序
......
```

宣告 50 個整數大小的陣列來存於成績似乎是個合理的作法，因為目前高中以下的每班人數多不超過 50 人，但.....要是超過了怎麼辦？那就設成 100吧！要是人家拿來做全校學生的排序怎麼辦？那改成 10000吧！這是個大問題，因為設大了浪費，設小了又無法運作。

使用 C99 的可變長度陣列是個方法。

```
int numOfStd=0;
cout << "請輸入學生人數：";
cin >> numOfStd;

int score[numOfStd];  // C99 的可變長度陣列

for(int i=0; i<numOfStd; i++) {
 cin >> score[i];
}......
```

但它不是 C++ 標準裡的必要特性，不是所有的 C++ 編譯器都支援，而且只能在函數內部使用，無法放在全域區，再者使用到的是堆疊記憶體，大小較受限。

為了解決前述的兩難狀況，我們必須有一個能在程式執行間動態依需求配置記憶體的方法。

在 C++ 中，我們可以用 new 這個關鍵字來要求配置一定大小的記憶體，若是成功要到指定大小的記憶體，它會回傳這塊記憶體的開頭位址，我們可用指標把它存起來。

配置

```
new 資料型別;       // 配置單一變數
new 資料型別[數量]; // 以陣列方式配置
```

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];
......
```

存取

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];

for(int i=0; i<numOfStd; i++) {
    cin >> score[i];
}
// 排序
......
```

也可以這麼做

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];
```

```
for(int i=0; i<numOfStd; i++) {
    cin >> *(score+i);
}
// 排序
......
```

釋回

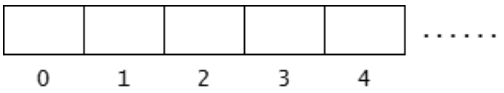
當不在需要使用到先前配置的記憶體時，記得要用 delete 將記憶體還給系統，讓其他程式可以使用該記憶體。

```
delete 指標名稱;      // 釋回單一變數所配置記憶體

delete [] 指標名稱;  // 釋回陣列所配置記憶體
```

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];

for(int i=0; i<numOfStd; i++) {
 cin >> score[i];
}
// 排序
......
delete [] score;
```

位址空間

在電腦裡面儲存資料的最基本單位是位元(bit)。而在記憶體中，我們存取資料的基本單位則是位元組(Byte)。我們可以把電腦的記憶體想像成是一連串的小盒子，每一個小盒子裡面可以放 1 Byte 的資料，這些盒子被按照順序加以編號，這個編號我們稱之為「位址」。



我們若是用 4 Byte 來儲存位址，則編號的範圍也就是位址的範圍可由 00 00 00 00 到 FF FF FF FF，共有 $ 2^{32} $ Byte，也就是 4 GB 的空間。

這就是為什麼 32 位元的電腦和作業系統無法存取超過 4 GB 記憶體的原因。64位元的系統用 8 Byte 來儲存位址，他可以定址的範圍為 00 00 00 00 00 00 00 00 到 FF FF FF FF FF FF FF FF，共有 $ 2^{64} $ Byte，即 $ 2^{34} $ GB 的空間。

Code::Blocks 近期的版本預設安裝 64 位元的編譯器，所以編譯出來的程式是 64 位元的。用下面這段程式碼可以看到整數的指標是 8 Byte。

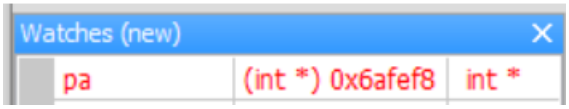
```
cout << sizeof(int*) << endl;
```

觀察資料在記憶體中的存放

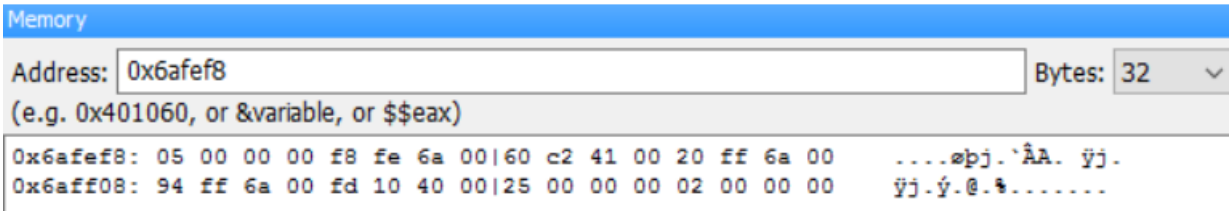
接下來我們使用 code::blocks 的 memory 視窗來觀察資料在記憶體中的存放方式。我們在下面這段程式的第 05 列設一個中斷點，用 debug 模式執行到該處。

```
int main()
{
 int a = 5;
 int *pa = &a; // &a 表示取得變數 a 在記憶體中的位址
 cout << a << endl;
 return 0;
}
```

用 watch 觀察 pa 的值，我們可以看到儲存 a 的記憶體位址，你看到的值可能有所不同，這是因為每次程式載入時會在記憶體的哪個地方是不一定的。



點選 [Debug]→[Debugginh windows] →[Memory dump]，打開記憶體檢視視窗。在Address 的地方輸入 pa 的值，從 0x6afef8 開始的連續4個 Byte 就是 a 的值。



你可能會覺得得怪為什麼是 05 00 00 00，而不是 00 00 00 05，這個和 Intel、AMD 的 CPU 設計有關，它會剛好反過來存放。

接下來我們觀察一下陣列中的資料存放方式。在這裡我們要觀察的重點是：

- 陣列中的資料是緊臨儲存在一起的
- 在程式碼中陣列的名稱就等於指向第一個元素的指標
- 指標加 1 後，值會變成多少？

```
int main()
{
 int a[5] = {100,200,400,600,800};
 int *pa = &a[0];
 pa = pa+1;
 pa = pa+1;
 pa = pa+1;
 pa = pa+1;
 return 0;
}
```

陣列名稱 a 其實就是 a[0] 的位址，也就是陣列在記憶體中的開始位址。在上面那段程式的第 05 行之後，使用 pa[0], pa[1], pa[2], ... ，就等於 a[0], a[1], a[2], ...。

觀察傳值與傳址呼叫

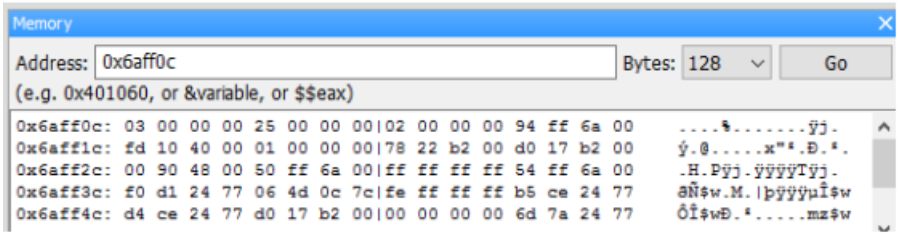
接下來我們藉著 Memory dump 視窗來觀察傳值呼叫與傳址呼叫的行為。

```
void f(int a)
{
 a = 5;
 return;
}

void g(int *a)
{
 *a = 5;
 return;
}

int main()
{
 int a = 3;
 f(a);
 g(&a);
 return 0;
}
```

將中斷點設在 16 列的地方。執行到中斷點時，把 &a 加入 watch。

&a	(int *) 0x6aff0c	int *

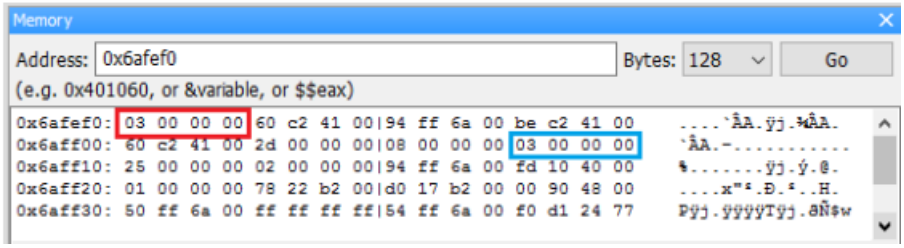
接著到 Memory dump 中找到 0x6aff0c 的地方，確認 03 00 00 00 在那裡。



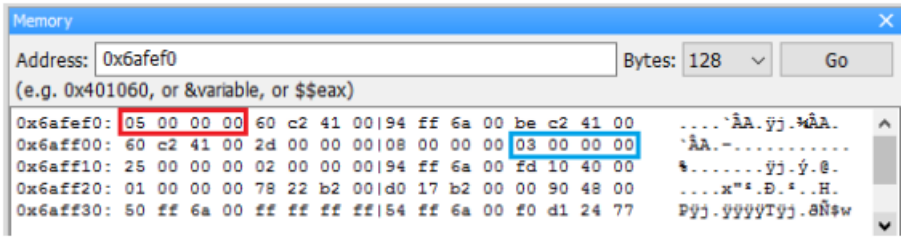
用 step into 追蹤到函數 f 裡面，這時你會發現在 watch 中的 &a 變了，因為這裡的 a 是函數 f 裡的區域變數 a，他的位址是 0x6afef0。

&a	(int *) 0x6afef0	int *

從這裡可以看到 f(3) 裡的參數 3 被複製到區域變數 a 裡，右下角 0x6affc0 是 main 函數裡的區域變數 a 。

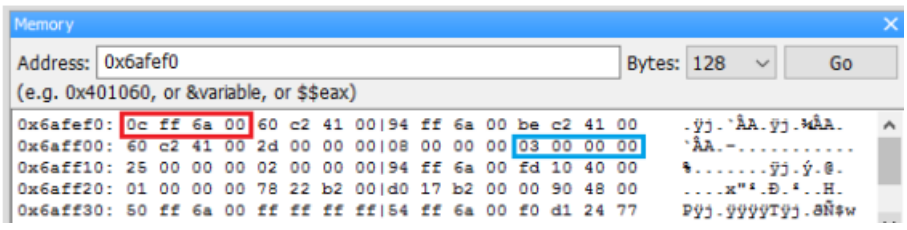


用 Next Line 執行下一行 “ a = 5; ”，可以看到函數 f 裡的區域變數 a (0xfafef0) 變成 5，而main 函數裡的區域變數 a (0xfaff0c) 值不變，依然還是 3。

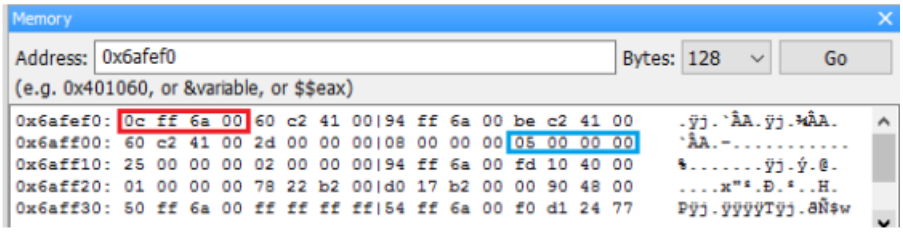


繼續用 Next Line 執行，直到返回 main 裡面。這時 watch 裡 &a 的值又變回 0xfaff0c，表示這時看到的 a 是 main 函數的區域變數 a。

比照剛才的做法，用 Step into 蹤至函數 g 裡，很巧的這次函數 g 裡的區域變數 a，放在0x6afef0。不過它的值不是 3喔，我們傳入 g 的是 main 函數裡區域變數 a 的位址，所以在Memory dump 中，可以看到區域變數的值是 0c ff 6a 00，即 main 函數中區域變數 a 的位 址。



用 Next Line 執行 “ *a=5; ” 這行，可以看到 0x006aff0c 這個位址的資料由 03 00 00 00 變成了 05 00 00 00。



這是因為我們使用 * 運算子，操作 a 所儲存位址(0x6aff0c)內的值。

繼續用 Next Line 執行，直到返回 main 裡面。這時 watch 裡 &a 的值又變回 0xfaff0c，表示這時看到的 a 是 main 函數的區域變數 a。

至此我們可以觀察到以下幾點：

1. 不管是傳值呼叫的 f，亦或是傳呼叫的 g，都是把呼叫函數時的引數值複製到被呼叫函數的參數(也是它的區域變數)中。只是前者複製的是變數的值，後者複製的是變數在記憶體中的位址。
2. 相對於傳入變數的值，傳入變數的位址讓我們可以直接藉由編輯該位址記憶體的值，達到修改外界變數值的目的。

08-自訂型別 (struct)

8-1 struct

1. 自訂型別 struct

在 C++ 中，我們可以把多個彼此相關的資料包在一起，創造出一個全新的型別。

例如，一位學生可能有以下資料：

- 姓名
- 年齡
- 成績

如果分別用三個變數來表示，資料容易分散，必須想辦法維持追蹤同一個學生的姓名、年齡和成績。

`struct` 是一種**自訂型別**，可以把多個彼此相關的資料包在一起，創造出一個全新的型別。

```
struct Student {
    string name;
    int age;
    double score;
};
```

這表示我們建立了一個名為 `Student` 的型別，它裡面有三個成員：`name`、`age`、`score`。

---

2. 為什麼需要 struct

`struct` 很適合用來表示「一筆完整資料」，例如：

- 學生資料
- 座標點
- 商品資訊
- 日期時間

學會 `struct` 之後，你會開始從「很多零散變數」進步到「有結構的資料設計」，這是寫大型程式的重要基礎。

---

3. 基本語法

宣告

```
struct Person {
    std::string name;
    int age;
};
```

語法重點：

- `struct` 是關鍵字
- `Person` 是型別名稱
- 大括號裡面放的是成員變數
- 最後要有分號 `;`

---

4. 建立變數與存取成員

宣告完 `struct` 之後，就可以像一般型別一樣建立變數。

```
Person p1;
```

使用 `.` 來存取成員：

```
p1.name = "Alice";
p1.age = 18;
```

範例：

```
#include <iostream>

using namespace std;

struct Person {
    string name;
    int age;
};

int main() {
    Person p1;
    p1.name = "Alice";
    p1.age = 18;

    cout << "Name: " << p1.name << endl;
    cout << "Age: " << p1.age << endl;

    return 0;
}
```

執行結果：

```
Name: Alice
Age: 18
```

5. 初始化（給定初值）

除了先宣告再指定值，也可以在建立時直接初始化。

```
Person p1 = {"Bob", 20};
```

這相當於：

- `p1.name = "Bob"`
- `p1.age = 20`

```
Person p2 = {"Cindy", 22};
cout << p2.name << ", " << p2.age << endl;
```

6. struct 用於函數的參數

你可以把整個 `struct` 傳進函式，讓程式更清楚。

```
#include <iostream>

using namespace std;

struct Person {
    string name;
    int age;
};

void printPerson(Person p) {
```

```
    cout << "Name: " << p.name << endl;
    cout << "Age: " << p.age << endl;
}

int main() {
    Person p = {"David", 25};
    printPerson(p);

    return 0;
}
```

由於函數預設是以 **傳值呼叫(call by value)** 方式將參數傳入，如果該 struct 裡的資料很大，例如包含學生照片，通常會改用 **傳參考呼叫(call by reverence)**，避免整份資料被複製：

```
void printPerson(Person& p) {
    cout << "Name: " << p.name << endl;
    cout << "Age: " << p.age << endl;
}
```

由於傳參考會讓被傳入的外部 struct 變數，可以在函數內被修改。若要確保不被修改，可以像這樣在參數前加上 `const` 標示其為常數。

```
void printPerson(const Person& p) {
    cout << "Name: " << p.name << endl;
    cout << "Age: " << p.age << endl;
}
```

---

7. 陣列中的 `struct`

`struct` 很常和陣列搭配使用，用來儲存多筆同類型資料。

```
#include <iostream>

using namespace std;

struct Student {
    string name;
    int score;
};

int main() {
    Student students[3] = {
        {"Amy", 90},
        {"Brian", 85},
        {"Chris", 92}
    };

    for (int i = 0; i < 3; i++)
    {
        cout << students[i].name << ": " << students[i].score << endl;
    }

    return 0;
}
```

這種寫法在成績系統、通訊錄、商品清單中都很常見。

---

8. 巢狀 struct

`struct` 裡面還可以再放另一個 `struct`。

```
struct Date {
```

```
    int year;
    int month;
    int day;
};

struct Student {
    string name;
    Date birthday;
};
```

使用方式：

```
Student s;
s.name = "Helen";
s.birthday.year = 2005;
s.birthday.month = 8;
s.birthday.day = 15;
```

這讓複雜資料可以分層整理，結構更清楚。

---

9. 指標與 `struct`

如果你有一個指向 `struct` 的指標，在存取其成員時要用 `->` 而不是 `.`。

```
Person p = {"Ivy", 21};
Person* ptr = &p;

cout << ptr->name << endl;
cout << ptr->age << endl;
```

比較

```
p.name.       // 物件存取成員
(*ptr).name   // 物件存取成員
ptr->name     // 指標存取成員
```

---

10. 一個完整綜合範例

```
#include <iostream>

using namespace std;

struct Student {
    string name;
    int age;
    double score;

    void printInfo() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Score: " << score << endl;
    }
};

int main() {
    Student s1 = {"Jack", 18, 95.5};
    s1.printInfo();

    return 0;
}
```

這個例子同時用到了：

- `struct` 宣告
- 成員變數
- 初始化
- 成員函式
- 物件呼叫函式

---

11. `struct` 的使用時機

當你遇到以下情境時，可以考慮使用 `struct`：

- 一筆資料有多個欄位
- 這些欄位彼此有關聯
- 你希望程式更容易閱讀與維護

例如：

- `Point`：座標 `(x, y)`
- `Book`：書名、作者、價格
- `Employee`：姓名、編號、薪資

---

12. 小專案練習：

專案背景

你要做一個小型的收銀系統。系統會：

- 建立一張發票（含建立時間與稅率）
- 掃描商品（品名與價格）
- 若商品重複出現，數量要累加
- 計算未稅金額與含稅總金額
- 最後列印完整發票

你會學到什麼

- 使用 struct 表示資料模型（Item、Date、Time、Invoice）
- 用函式分工（prepareInvoice、scanItem、printInvoice）
- 使用固定大小陣列儲存品項
- 在迴圈中搜尋重複資料
- 更新數量與金額
- 使用 C++ 標準函式取得目前日期時間

專案檔案

- main.cpp：主要程式
- in.txt：測試輸入資料（可用重導向測試）

功能需求

1. 初始化發票

- 建立空白發票
- 讀取目前日期與時間
- 設定稅率（例如 0.05）
- amount 初始為 0
- totalDue 初始為 amount * (1 + taxRate)

2. 掃描商品

- 每次輸入一筆：name price
- 若 name 已存在：
  - quantity + 1
  - amount 只加上本次 price
- 若 name 不存在：

    - 新增一筆 Item{name, price, 1}
    - amount 加上 price
- 每次更新 amount 後都要重算 totalDue

3. 列印發票

- 顯示日期與時間
- 顯示所有品項（品名、價格、數量）
- 顯示 amount、taxRate、totalDue

建議輸入格式

第一行輸入整數 n（要掃描幾筆） 接著輸入 n 行，每行一筆：

- name price

範例：

```
10
Apple 35.5
Banana 12
Milk 45
Bread 30
Apple 35.5
Eggs 68
Coffee 120
Bread 30
Orange 25
Milk 45
```

建議輸出格式

```
Date: 2026/4/10
Time: 10:7:56
------------------------------
Items:
- Apple: $35.5 x 2
- Banana: $12 x 1
- Milk: $45 x 2
- Bread: $30 x 2
- Eggs: $68 x 1
- Coffee: $120 x 1
- Orange: $25 x 1
------------------------------
Amount: $446
Tax Rate: 5%
Total Due: $468.3
```

如何測試

你可以用輸入重導向測試：

```
main.exe < in.txt
```

驗收標準

- 程式可成功編譯執行
- 重複品項會正確累加 quantity
- amount 不會因重複品項而重複倍算
- totalDue 計算正確（amount * (1 + taxRate)）
- 發票輸出格式清楚可讀

提示

- 先完成 prepareInvoice，再做 scanItem，最後做 printInvoice
- scanItem 先用 for 迴圈找同名品項

- 找到就更新並 return，沒找到才新增
- 注意陣列上限（itemList 最多 100 筆）
- 若輸入失敗（cin 失敗）要考慮防呆

額外挑戰

- 若同名商品出現不同價格，定義你自己的規則（拒絕、覆蓋、或視為不同品項）
- 顯示小計欄位（price * quantity）
- 將稅金獨立列印（tax = totalDue - amount）
- 改用 vector 取代固定長度陣列 - 需使用到還沒教的 STL 容器 vector
- 支援含空白的品名（例如 "Green Tea"） - 需使用到還沒教的 getline 函數

8-2 小專案參考解答

```
#include <iostream>
#include <ctime>

using namespace std;

struct Item
{
    string name;
    double price;
    int quantity;
};

struct Date
{
    int year;
    int month;
    int day;
};

struct Time
{
    int hh;
    int mm;
    int ss;
};

struct Invoice
{
    Item itemList[100];
    int countOfItemList;
    Date date;
    Time time;

    double amount;

    double taxRate;
    double totalDue;
};

Invoice prepareInvoice(double taxRate)
{
    Invoice invoice{};

    // get & set current date/time
    time_t now = time(nullptr);
    tm localTime{};
    localtime_s(&localTime, &now);

    invoice.date.year = localTime.tm_year + 1900;
    invoice.date.month = localTime.tm_mon + 1;
    invoice.date.day = localTime.tm_mday;

    invoice.time.hh = localTime.tm_hour;
    invoice.time.mm = localTime.tm_min;
    invoice.time.ss = localTime.tm_sec;

    invoice.countOfItemList = 0;
    invoice.amount = 0.0;
    invoice.taxRate = taxRate;
    invoice.totalDue = invoice.amount * (1.0 + invoice.taxRate);

    return invoice;
}
```

```
void printInvoice(const Invoice& invoice)
{
    printf("Date: %d/%d/%d\n", invoice.date.year, invoice.date.month, invoice.date.day);
    printf("Time: %d:%d:%d\n", invoice.time.hh, invoice.time.mm, invoice.time.ss);

    printf("------------------------------\n");
    printf("Items:\n");
    for (int i = 0; i < invoice.countOfItemList; ++i)
    {
        const Item& item = invoice.itemList[i];
        printf("- %s: $%.2f x %d\n", item.name.c_str(), item.price, item.quantity);
    }

    printf("------------------------------\n");
    printf("Amount: $%.2f\n", invoice.amount);
    printf("Tax Rate: %.2f%%\n", invoice.taxRate * 100);
    printf("Total Due: $%.2f\n", invoice.totalDue);
}

void scanItem(Invoice& invoice)
{
    string name;
    double price;

    cin >> name;
    cin >> price;

    for (int i = 0; i < invoice.countOfItemList; ++i)
    {
        if (invoice.itemList[i].name == name)
        {
            invoice.itemList[i].quantity += 1;
            invoice.amount += price;
            invoice.totalDue = invoice.amount * (1.0 + invoice.taxRate);
            return;
        }
    }

    Item newItem{name, price, 1};
    invoice.itemList[invoice.countOfItemList++] = newItem;
    invoice.amount += price;
    invoice.totalDue = invoice.amount * (1.0 + invoice.taxRate);
}

int main()
{
    double taxRate = 0.05;

    Invoice invoice = prepareInvoice(0.05);

    int n;
    cin >> n;
    for(int i=0; i<n; ++i)
    {
        scanItem(invoice);
    }
    printInvoice(invoice);

    return 0;
}
```

09-STL 容器 - vector

9-1 vector

1. 為什麼需要 vector？

1.1 傳統陣列的限制

在 C++ 中，我們熟悉的傳統陣列有一個根本的問題：**大小必須在編譯時決定，且無法改變。**

```
int scores[100];   // 固定 100 格，多了浪費，少了不夠用
```

傳統陣列的痛點：

- 宣告時必須給定大小（或使用 `new`，但要自己管理記憶體）
- 不知道目前有幾個元素（需要額外的變數追蹤）
- 插入、刪除元素需要手動搬移資料
- 忘記 `delete[]` 就記憶體洩漏(memory leak)

```
// 傳統做法：又長又容易出錯
int* scores = new int[100];  // 動態跟作業系統要一塊記憶體
int count = 0;

scores[count++] = 90;  // 注意這裡的 count++，count 值先被評估，再遞增 1
scores[count++] = 85;

cout << "We have " << count << " items in scores[]" << endl;

for(int i=0; i<count; i++)
{
    cout << "scores[" << i << "] = " << scores[i] << endl;
}

// 用完還要記得把記憶體還作業系統
delete[] scores;
```

1.2 vector 是什麼？

`vector` 是 C++ **標準模板庫（STL）**中的一種**動態陣列容器**。

它的核心優勢：

- 大小可以**自動擴張**，不需要手動管理記憶體
- 提供豐富的**成員函數**，操作方便
- 支援與陣列相同的**下標存取** `[]`
- 離開作用域後**自動釋放記憶體**

1.3 圖解：記憶體動態擴張

```
初始狀態 (capacity = 1) ：
┌───┐
| 3 |
└───┘
  size=1, capacity=1

push_back(7)，空間不足 → 自動擴張為 capacity=2：
┌───┬───┐
| 3 | 7 |
└───┴───┘
  size=2, capacity=2

push_back(5)，空間不足 → 自動擴張為 capacity=4：
┌───┬───┬───┬───┐
| 3 | 7 | 5 |   |
└───┴───┴───┴───┘
```

```
  size=3, capacity=4（預留空間）

push_back(1)，空間足夠 → 直接寫入：
┌───┬───┬───┬───┐
│ 3 │ 7 │ 5 │ 1 │
└───┴───┴───┴───┘
  size=4, capacity=4
```

重點：size 是目前元素數量，capacity 是已分配的記憶體空間。vector 通常以倍增方式擴張，以攤平擴張的成本。

9-2 vector 的基礎操作

vector 基礎操作

1. 引入標頭檔與宣告

使用 `vector` 前，需要引入標頭檔：

```
#include <iostream>
#include <vector>

using namespace std;
```

2. 宣告與初始化

```
// 方式一：宣告空的 vector
// capacity:0, size: 0
vector<int> v1;

// 方式二：指定初始大小（元素值為 0）
// capacity:5, size: 5
vector<int> v2(5);          // [0, 0, 0, 0, 0]

// 方式三：指定大小與預設值
// capacity:5, size: 5
vector<int> v3(5, 100);     // [100, 100, 100, 100, 100]

// 方式四：用初始化列表
// capacity:5, size: 5
vector<int> v4 = {3, 1, 4, 1, 5};  // [3, 1, 4, 1, 5]

// 方式五：複製另一個 vector
// capacity:5, size: 5
vector<int> v5(v4);         // [3, 1, 4, 1, 5]

// 也可以存其他型別
vector<double> scores;
vector<string> names;
vector<bool>   flags;
```

3. 查詢大小與清空

```
vector<int> v = {1, 2, 3, 4, 5};

cout << v.size();    // 5：目前元素個數
cout << v.empty();   // 0 (false) ：是否為空

v.clear();           // 清空所有元素
cout << v.size();    // 0
cout << v.empty();   // 1 (true)
```

4. 新增與移除元素

在末尾新增元素

vector 的成員函數 `push_back(val)`，可以將一個值新增到目前 vector 的尾端。

```
vector<int> v;
v.push_back(10);   // [10]
v.push_back(20);   // [10, 20]
v.push_back(30);   // [10, 20, 30]
```

移除末尾元素

vector 的成員函數 `pop_back()`，可以將 vector 尾端的那個元素移除。

```
v.pop_back();      // [10, 20]
```

⚠ **注意**：對空的 vector 呼叫 pop_back() 是未定義行為（UB），請先確認 !v.empty()。

5. 存取元素

下標運算子 `[]`

```
vector<int> v = {10, 20, 30};
cout << v[0];   // 輸出 10
v[1] = 99;      // 修改第二個元素
```

帶邊界檢查的存取 `at(i)`

```
cout << v.at(0);   // 輸出 10
cout << v.at(5);   // 超出範圍 → 丟出 std::out_of_range 例外
```

方式	速度	邊界檢查	建議使用時機
`v[i]`	較快	□ 沒有	確定索引合法時
`v.at(i)`	稍慢	□ 有	需要安全保障時

6. 範例：動態收集成績

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> scores;
    int n, score;

    cout << "請輸入學生人數：";
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cout << "輸入第 " << i + 1 << " 位學生成績：";
        cin >> score;
        scores.push_back(score);   // 動態新增
    }

    // 計算總分
    int total = 0;
    for (int i = 0; i < scores.size(); i++) {
        total += scores[i];
    }

    cout << "平均分數：" << (double)total / scores.size() << endl;
    return 0;
}
```

9-3 走訪 vector

走訪 vector

1. 方法一：下標 for 迴圈

最直覺的方式，適合需要使用索引時：

```
vector<int> v = {10, 20, 30, 40, 50};

for (int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
// 輸出：10 20 30 40 50
```

⚠ v.size() 回傳型別為 size_t（無號整數），若用 int i = v.size() - 1 倒著走時要小心，空 vector 的 0 - 1 會變成一個很大的數。

2. 方法二：Range-based for（範圍式 for）

C++11 起支援，最簡潔：

```
// 唯讀走訪
for (int x : v) {
    cout << x << " ";
}

// 修改元素（加上 &）
for (int& x : v) {
    x *= 2;   // 每個元素乘以 2
}
```

3. 方法三：迭代器（Iterator）

迭代器的概念類似**指標**，這也是為什麼熟悉指標的你能很快上手。

圖解：迭代器與指標的對比

```
傳統陣列指標：
  int arr[] = {10, 20, 30};
  int* ptr = arr;       // ptr 指向第一個元素

  ┌────┬────┬────┐
  │ 10 │ 20 │ 30 │
  └────┴────┴────┘
    ↑
   ptr

  ptr++;                // 指向下一個
  *(ptr) = 99;          // 解參考後修改

vector 迭代器：
  vector<int> v = {10, 20, 30};
  auto it = v.begin();  // it 指向第一個元素

  ┌────┬────┬────┬───────┐
  │ 10 │ 20 │ 30 │(end)  │
  └────┴────┴────┴───────┘
    ↑                ↑
  begin()          end()（指向最後一個之後）
```

```
  it++;             // 指向下一個（同 ptr++）
  *it = 99;         // 解參考後修改（同 *ptr）
```

迭代器的基本用法

```
vector<int> v = {10, 20, 30, 40, 50};

// 傳統的寫法，宣告 iterator 的型別有點複雜
// vector<int>::iterator it = v.begin();

// 自 C++ 11 起可以用 auto 自動推導型別
auto it = v.begin();    // 指向第一個元素

while (it != v.end()) {
    cout << *it << " ";  // 解參考取值
    it++;                // 移動到下一個
}
// 輸出：10 20 30 40 50
```

更常見的寫法（for 迴圈）：

```
for (auto it = v.begin(); it != v.end(); it++) {
    cout << *it << " ";
}
```

三種走訪方式比較

方式	優點	缺點
下標 `v[i]`	直覺，可使用索引	需要注意型別
range-based for	最簡潔	無法直接取得索引
迭代器	與 STL 演算法相容	語法略繁瑣

✅ 實務建議：平常用 range-based for，需要插入/刪除時用迭代器，需要索引時用下標。

9-4 常用成員函數

其他常用成員函數

1. front() 與 back()

```
vector<int> v = {10, 20, 30, 40, 50};

cout << v.front();  // 10：第一個元素
cout << v.back();   // 50：最後一個元素

v.front() = 99;     // 可以修改
v.back()  = 1;
// v 現在是 {99, 20, 30, 40, 1}
```

2. insert()：在指定位置插入

```
vector<int> v = {1, 2, 3, 4, 5};

// insert(位置迭代器, 值)
auto it = v.begin() + 2;   // 指向索引 2（值為 3）
v.insert(it, 99);

// v 現在是 {1, 2, 99, 3, 4, 5}
```

圖解：

```
插入前：
┌───┬───┬───┬───┬───┐
│ 1 │ 2 │ 3 │ 4 │ 5 │
└───┴───┴───┴───┴───┘
          ↑
        it（索引 2）

插入後（後面的元素全部後移）：
┌───┬───┬────┬───┬───┬───┐
│ 1 │ 2 │ 99 │ 3 │ 4 │ 5 │
└───┴───┴────┴───┴───┴───┘
```

⚠️ insert() 後，原本的迭代器可能**失效**，請重新取得。

3. erase()：移除指定位置的元素

```
vector<int> v = {1, 2, 99, 3, 4, 5};

// 移除單一元素
v.erase(v.begin() + 2);   // 移除索引 2（值 99）
// v 現在是 {1, 2, 3, 4, 5}

// 移除一個範圍 [begin+1, begin+3)（左閉右開）
v.erase(v.begin() + 1, v.begin() + 3);
// 移除索引 1 和 2（值 2 和 3）
// v 現在是 {1, 4, 5}
```

4. resize() 與 reserve()

這兩個函數常被搞混，用下圖區分：

resize(n)：改變 size（元素個數）

```
vector<int> v = {1, 2, 3};  size=3, capacity=3

v.resize(5);
→ {1, 2, 3, 0, 0}           size=5, capacity=5（新元素填 0）

v.resize(2);
→ {1, 2}                    size=2, capacity=5（縮小 size，不影響 capacity）
```

reserve(n)：改變 capacity（預留記憶體）

```
vector<int> v = {1, 2, 3};  size=3, capacity=3

v.reserve(10);
→ {1, 2, 3}                 size=3, capacity=10（只預留空間，不新增元素）
```

何時用 reserve？

當你事先知道大概會新增多少元素時，使用 `reserve` 可以避免多次重新分配記憶體，提升效能：

```
vector<int> v;
v.reserve(1000);          // 預先分配 1000 個空間

for (int i = 0; i < 1000; i++) {
    v.push_back(i);       // 不會觸發記憶體重新分配
}
```

5. capacity()：查詢已分配空間

```
vector<int> v;
cout << v.size()     << endl;   // 0
cout << v.capacity() << endl;   // 0（或平台相關的初始值）

v.push_back(1);
cout << v.capacity() << endl;   // 1（或更大，依平台）

v.push_back(2);
cout << v.capacity() << endl;   // 2（擴張）

v.push_back(3);
cout << v.capacity() << endl;   // 4（擴張為倍數）
```

6. 二維 vector

`vector` 可以嵌套，用來表示矩陣或表格：

```
// 宣告 3x4 的二維 vector，初始值全為 0
vector<vector<int>> matrix(3, vector<int>(4, 0));
```

圖解：

```
matrix：
        [0]  [1]  [2]  [3]
[0] ──► ┌────┬────┬────┬────┐
        │ 0  │ 0  │ 0  │ 0  │
        └────┴────┴────┴────┘
[1] ──► ┌────┬────┬────┬────┐
        │ 0  │ 0  │ 0  │ 0  │
        └────┴────┴────┴────┘
[2] ──► ┌────┬────┬────┬────┐
        │ 0  │ 0  │ 0  │ 0  │
        └────┴────┴────┴────┘
```

```
// 存取與修改
matrix[1][2] = 99;
```

```
// 走訪二維 vector
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}

// 使用 range-based for
for (auto& row : matrix) {
    for (int val : row) {
        cout << val << " ";
    }
    cout << endl;
}
```

注意：二維 vector 的每一列長度可以不同（鋸齒狀陣列），這與傳統二維陣列不同。

9-5 實作練習

練習一：成績管理系統（基礎操作）

題目敘述

請你撰寫一個程式，功能如下：

1. 讀入 N 筆學生成績（整數，0~100）
2. 輸出所有成績
3. 輸出最高分、最低分、平均分數（取到小數點後兩位）
4. 刪除最後一筆成績後，再輸出一次所有成績

範例輸入

```
5
90 75 88 62 95
```

範例輸出

```
成績：90 75 88 62 95
最高分：95
最低分：62
平均：82.00
刪除最後一筆後：90 75 88 62
```

提示

- 使用 `push_back` 加入一筆成績
- 善用 `size()`、`pop_back()`

▼ **參考解答（請先自己嘗試！）**

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

int main() {
    int n;
    cin >> n;

    vector<int> scores;
    for (int i = 0; i < n; i++) {
        int s;
        cin >> s;
        scores.push_back(s);
    }

    // 輸出所有成績
    cout << "成績：";
    for (int s : scores) cout << s << " ";
    cout << endl;

    // 找最大最小
    int maxVal = scores[0], minVal = scores[0];
    int total = 0;
    for (int s : scores) {
        if (s > maxVal) maxVal = s;
        if (s < minVal) minVal = s;
        total += s;
    }
```

```
    cout << "最高分：" << maxVal << endl;
    cout << "最低分：" << minVal << endl;
    cout << fixed << setprecision(2);
    cout << "平均：" << (double)total / scores.size() << endl;

    scores.pop_back();

    cout << "刪除最後一筆後：";
    for (int s : scores) cout << s << " ";
    cout << endl;

    return 0;
}
```

練習二：移除特定元素（insert / erase）

題目敘述

給定一個整數序列，請你：

1. 讀入 N 個整數
2. 讀入一個目標值 K
3. 移除序列中**所有等於 K 的元素**
4. 在序列**開頭**插入一個數字 0
5. 輸出最終序列

範例輸入

```
8
1 3 5 3 2 3 7 9
3
```

範例輸出

```
0 1 5 2 7 9
```

提示

- 移除元素後，迭代器的位置會改變，請注意 `erase()` 的回傳值（它會回傳被刪除元素的下一個位置）
- 在開頭插入用 `v.insert(v.begin(), 0)`

常見陷阱

```
// ❌ 錯誤寫法：erase 後 it 已失效，不可 it++
for (auto it = v.begin(); it != v.end(); it++) {
    if (*it == K) v.erase(it);
}

// ✅ 正確寫法：erase 回傳下一個有效迭代器
for (auto it = v.begin(); it != v.end(); ) {
    if (*it == K)
        it = v.erase(it);  // 不 it++
    else
        it++;
}
```

▼ 參考解答（請先自己嘗試！）

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

int main() {
```

```
    int n;
    cin >> n;

    vector<int> v;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        v.push_back(x);
    }

    int K;
    cin >> K;

    // 移除所有等於 K 的元素
    for (auto it = v.begin(); it != v.end(); ) {
        if (*it == K)
            it = v.erase(it);
        else
            it++;
    }

    // 在開頭插入 0
    v.insert(v.begin(), 0);

    // 輸出
    for (int x : v) cout << x << " ";
    cout << endl;

    return 0;
}
```

練習三：矩陣加法（二維 vector）

題目敘述

給定兩個 N×M 的矩陣 A 和 B，請計算 C = A + B，並輸出結果矩陣 C。

範例輸入

```
2 3
1 2 3
4 5 6
7 8 9
1 0 1
```

（第一行為 N M，接下來 N 行為矩陣 A，再 N 行為矩陣 B）

範例輸出

```
8 10 12
5 5 7
```

提示

- 使用 `vector<vector<int>>` 來儲存矩陣
- 初始化：`vector<vector<int>> C(n, vector<int>(m, 0));`

▼ **參考解答（請先自己嘗試！）**

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

int main() {
```

```
    int n, m;
    cin >> n >> m;

    vector<vector<int>> A(n, vector<int>(m));
    vector<vector<int>> B(n, vector<int>(m));
    vector<vector<int>> C(n, vector<int>(m, 0));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> A[i][j];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> B[i][j];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            C[i][j] = A[i][j] + B[i][j];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << C[i][j];
            if (j < m - 1) cout << " ";
        }
        cout << endl;
    }

    return 0;
}
```

重點總覽

常用成員函數速查表

函數	功能	時間複雜度
`push_back(val)`	在末尾新增元素	O(1) 均攤
`pop_back()`	移除末尾元素	O(1)
`v[i]`	存取索引 i 的元素（無檢查）	O(1)
`v.at(i)`	存取索引 i 的元素（有檢查）	O(1)
`size()`	回傳元素個數	O(1)
`empty()`	是否為空	O(1)
`clear()`	清空所有元素	O(n)
`front()`	第一個元素	O(1)
`back()`	最後一個元素	O(1)
`insert(it, val)`	在迭代器位置前插入	O(n)
`erase(it)`	移除迭代器位置的元素	O(n)
`resize(n)`	調整元素個數	O(n)
`reserve(n)`	預留記憶體空間	O(n)
`capacity()`	已分配的空間大小	O(1)
`begin()`	指向第一個元素的迭代器	O(1)
`end()`	指向最後一個之後的迭代器	O(1)

三大常見陷阱

1. **`erase` 後迭代器失效**：請使用 `it = v.erase(it)` 取得新的有效迭代器。
2. **`pop_back` 空 `vector`**：呼叫前先確認 `!v.empty()`。
3. **`size()` 型別為 `size_t`**：比較或相減時注意無號整數的行為。

10-類別(class)

10-1 類別(class)與物件(object)

一、物件（object）與類別（class）

在 C++ 中物件和類別有很嚴謹的定義，我們在這裡僅使用簡單例子來做介紹。

什麼是物件(Object)？你的iPhone是個物件、你現在坐著的這張椅子是個物件，小狗小黃也是個物件。

那類別呢(Class)？你的iPhone是個物件，隔壁同學的HTC One也是個物件，而它們同屬於「手機」這個類別。小黃是個物件、小黑也是個物件，牠們同屬於「狗」這個類別。

二、建立類別與產生物件

在產生物件之前必須先建立類別，在C++中建立類別要使用到 class 這個關鍵字，一個最簡單的類別長這個模樣(注意：最後有一個分號)。

```
class 類別名稱 {
};
```

假設我們要建立一個dog類別，可以這樣寫。

```
class dog {
};
```

接下來我們可以這樣產生兩個 dog 類別的物件：shiro（小白） 和 kuro（小黑）。

```
dog shiro;
dog kuro;
```

或

```
dog shiro, kuro;
```

就像在定義 int 或 float 等原生型別變數一樣。

三、屬性 (attribute)/資料成員(data member)

前面這個 dog類別是一個很單純的類別，它產生的兩個物件也沒什麼用處。現在我們要給它一點變化，讓狗有不同的顏色和叫聲。

```
class dog {
  public:
    string color;
    string sound;
};
```

public 關鍵字表示下面出現的屬性都是「公開的」，你可以在程式的任何地方來改變其值。

在這個例子中我們有兩個 string 型別的屬性 color 和 sound。我們可以用 `物件名稱.屬性名稱` 的型式來設定或取用特定物件的某個屬性值。

```
dog shiro;
dog kuro;

shiro.color = "白色";
kuro.color = "黑色";

cout << "shiro 的顏色是" << shiro.color << endl;
cout << "kuro 的顏色是" << kuro.color << endl;
```

四、成員函數(member function)

到目前為止，你會覺得 class 和之前學的 struct 一樣。

但是除了屬性之外，我們還可以在類別中加入成員函數，讓該類別的物件可以「做些事情」，例如：我們可以在 dog 類別中加入 bark 這個成員函數，讓小白和小黑可以叫。

```
class dog {
  public:
    string color;
    string sound;

    void bark() {
        cout << sound << endl;
    }
};

int main()
{
    dog shiro, kuro;

    shiro.color = "白色";
    shiro.sound = "汪！";
    kuro.color = "黑色";
    kuro.sound = "汪汪汪！";

    cout << "shiro 咬他！" << endl;
    shiro.bark();                    // 汪！
    cout << "kuro 咬他！" << endl;
    kuro.bark();                     // 汪汪汪！

    return 0;
}
```

五、特殊的成員函數- 建構(constructor), 解構(destructor)

constructor 是一個和類別同名的成員函數，在一個物件被產生時會自動被呼叫，而且 `沒有傳回值`

```
class dog {
  public:
    string color;
    string sound;

    void bark() {
        cout << sound << endl;;
    }
    dog() {
        cout << "有一隻小狗誕生了！" << endl;
    }
};

int main()
{
    dog shiro, kuro;

    return 0;
}
```

一個類別可以有很多個 constructor，每個 constructor 的參數列都必須不一樣，在產生物件時，根據你使用參數的不同，相對應的 constructor 會自動被呼叫。

沒有參數的 constructor 稱為 `default constructor`。

```
class dog {
  public:
    string color;
    string sound;

    void bark() {
        cout << sound << endl;;
    }
    dog() {
        cout << "有一隻小狗誕生了！" << endl;
    }
    dog(string _color, string _sound) {
        color = _color;
        sound = _sound;
        cout << "有一隻" << color << "小狗誕生了！" << sound << endl;
    }
};

int main()
{
    dog kuro;
    dog shiro("白色","汪！");

    return 0;
}
```

destructor 是物件「死亡」時會被自動呼叫的成員函數，每個類別只有一個 destructor，它的名字是 ~類別名稱，**`destructor 沒有參數也沒有傳回值`**。

```
class dog {
  public:
    string color;
    string sound;
    string name;

    void bark() {
        cout << sound << endl;;
    }

    dog(string _name, string _color, string _sound) {
        name = _name;
        color = _color;
        sound = _sound;
        cout << "有一隻名為 " << name << " 的 " << color << " 小狗誕生了！"
                << sound << endl;
    }
    ~dog() {
        cout << name << " 上天堂了。" << endl;
    }
};

int main()
{
    dog shiro("小白", "白色", "汪！");
    dog *kuro = new dog("小黑", "黑色","喵！");

    delete kuro;

    return 0;
}
```

六、使用指標操作物件

使用指標操作物件與使用指標操作 struct 變數一樣，必須用到「->」。

```
int main()
```

```
{
    dog *kuro = new dog("小黑", "黑色","喵！");

    kuro->sound = "汪！汪！";
    kuro->bark();

    delete kuro;

    return 0;
}
```

七、this 指標

this 是一個指向「自己這個物件」的指標。

成員函數的參數名稱和屬性名稱完全一樣時，不加 this-> 編譯器會把兩邊都當作參數，屬性永遠不會被賦值。

```
#include <iostream>

using namespace std;

class Student {
public:
    string name;
    int age;

    void setName(string name) {
        //name = name;         // 自己賦值給自己，成員變數沒有被改到
        this->name = name;     // 左邊是成員變數，右邊是參數
    }
    void setAge(int age) {
        //age = age;           // 自己賦值給自己，成員變數沒有被改到
        this->age = age;       // 同上
    }

    void printInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};


int main()
{
    Student std;
    std.setName("Alice");
    std.setAge(20);

    std.printInfo();

    return 0;
}
```

使用 this

```
Name: Alice, Age: 20
```

不使用 this

```
Name: , Age: -103249728
```

10-2 存取控制——public 與 private

一、class 裡的東西，不是誰都可以動的

先看一個銀行帳戶的例子。

```
#include <iostream>

using namespace std;

class BankAccount {
public:
    string owner;
    int balance;
};

int main() {
    BankAccount acc;
    acc.owner   = "小明";
    acc.balance = 1000;

    acc.balance = -99999;   // 沒有任何阻擋，帳戶餘額變成負的！

    cout << acc.balance << endl;
}
```

這是因為我們之前在 class 裡用了 `public:` 這個存取修飾詞。 public 的意思就是「公開的」，誰想看、誰想改都可以。

用 `private:` 將存取限制改為「私有的」看看差別。

```
#include <iostream>

using namespace std;

class BankAccount {
private:
    string owner;
    int balance;
};

int main() {
    BankAccount acc;
    acc.owner   = "小明";
    acc.balance = 1000;

    acc.balance = -99999;   // 沒有任何阻擋，帳戶餘額變成負的！

    cout << acc.balance << endl;
}
```

修改之後，連編譯都失敗，出現了大量如下的錯誤訊息。

因為使用 privte 修飾的屬性和成員函數，在該 class 之外(第 5~9 行之外)都無法存取，在編譯時就攔下了不合法的存取。

```
main.cpp:13:9: error: 'std::string BankAccount::owner' is private within this context
   13 |     acc.owner   = "小明";
      |         ^~~~~
main.cpp:7:12: note: declared private here
    7 |     string owner;
      |            ^~~~~
main.cpp:14:9: error: 'int BankAccount::balance' is private within this context
```

```
   14 |     acc.balance = 1000;
      |         ^~~~~~
main.cpp:8:9: note: declared private here
    8 |     int balance;
      |         ^~~~~~
main.cpp:16:9: error: 'int BankAccount::balance' is private within this context
   16 |     acc.balance = -99999;   // 沒有任何阻擋，帳戶餘額變成負的！
      |         ^~~~~~
main.cpp:8:9: note: declared private here
    8 |     int balance;
      |         ^~~~~~
main.cpp:18:17: error: 'int BankAccount::balance' is private within this context
   18 |     cout << acc.balance << endl;
      |                 ^~~~~~
main.cpp:8:9: note: declared private here
    8 |     int balance;
      |         ^~~~~~
```

二、提供外界有限的操作

設成 public 太危險，設成 private 又動不了，怎麼辦呢?

我們可以

- 把內部屬性設為 private
- 用 public 開放幾個成員函數，給外界有限能力操作內部屬性

```
#include <iostream>

using namespace std;

class BankAccount {
private:
    string owner;
    int balance;    // 外部無法直接存取

public:
    // 以下成員函數，開放給外部使用，提供存取 balance 的管道
    bool withdraw(int amount) {
        if (amount > balance) {
            cout << "餘額不足！" << endl;
            return false;
        }
        balance -= amount;
        return true;
    }

    void deposit(int amount) {
        if (amount <= 0) {
            cout << "存款金額必須大於 0！" << endl;
            return;
        }
        balance += amount;
    }

    int getBalance() { return balance; }
};

int main()
{
    BankAccount account;
    account.deposit(1000);  // 存款 1000 元
    cout << "目前餘額: " << account.getBalance() << " 元" << endl;

    if (account.withdraw(500)) {  // 提款 500 元
        cout << "提款成功！" << endl;
    }
```

```
    cout << "目前餘額: " << account.getBalance() << " 元" << endl;

    if (!account.withdraw(600)) {  // 嘗試提款 600 元，應該會失敗
        cout << "提款失敗！" << endl;
    }
    cout << "目前餘額: " << account.getBalance() << " 元" << endl;

    return 0;
}
```

現在外部程式只能透過 `withdraw()` 和 `deposit()` 來改變餘額，透過 `getBalance()` 來取得餘額，非法操作會被擋下來。

public、private 存取修飾詞的作用範圍

```
class Example {
// 這裡沒寫 → class 預設是 private
    int secret;

public:
    int visible;
    void doSomething() { }

private:
    int alsoSecret;
};
```

`public:` 和 `private:` 後面的所有成員，都套用該修飾子，直到遇到下一個修飾子為止。

三、封裝(encapsulate)的概念

有時候我們不希望 class 的使用者直接去更動某些 data member 的資料，這時我們會將這些 data member 宣告在 private 區域中，並在 public 區域中提供 member function 來操作這些 data member。

以手錶為例，一般來說製表師傅不會希望使用者自己去撥動錶內的齒輪，所以他會把手錶內部的複雜機構用錶殼緊緊的封裝起來，不過他還是會提供一些按鈕或旋鈕來讓我們調整時間。

- private – 手錶的內部複雜機構
- public – 調整時間的按鈕或旋鈕

當我們試著在非成員函數裡存取 private 的 data member 或 member function 時，在編譯時期就會出現錯誤訊息。

這就是「物件導向程式設計」中很重要的「封裝」概念。我們設計了一個很精巧的 class，包含了很多的屬性和成員函數，但是對外界來說，它被一個「盒子」封裝起來，你只操作這個盒子外漏的那幾個界面。

10-3 如何建立複雜的類別

一、成員函數可以定義在 class 之外

我們可以只在 class body 中 `宣告` member function，再將它 `定義` 在 class body 之外。

```
class sprite {
private:
    string name;
public:
    sprite(string _name);  // 只有宣告
    string getName();      // 只有宣告
};

// 定義 sprite 類別的 sprite 成員函數
sprite::sprite(string _name)
{
    name = _name;
}

// 定義 sprite 類別的 getName 成員函數
string sprite::getName()
{
    return name;
}
```

以下是一個遊戲中角色的範例

```
#include <iostream>

using namespace std;

class sprite
{
private:
    string name;
    string status;
    int HP;
    int maxHP;

    void setStatus(void);  // 根據 hp 設定健康狀態
    void drawHpBar(void);  // 繪製 hp 長條圖
public:
    sprite(string _name);
    string getName();      // 取得姓名
    void decHP(int n);     // 減少 HP
    void addHP(int n);     // 增加 HP
    void show(void);       // 顯示狀態
};

void sprite::setStatus(void)
{
    if(HP >= maxHP *0.9)
        status = "健康";
    else if(HP >= maxHP *0.6)
        status = "受傷";
    else if(HP >= maxHP *0.3)
        status = "重傷";
    else if(HP > 0)
        status = "瀕死";
    else
        status = "死亡";
}
```

```
sprite::sprite(string _name)
{
    name = _name;
    HP = maxHP = 20;
    status = "健康";
}

string sprite::getName()
{
    return name;
}

void sprite::decHP(int n)
{
    HP = HP - n;
    if(HP <0)
        HP = 0;
    setStatus();
}

void sprite::addHP(int n)
{
    HP = HP + n;
    if(HP > maxHP)
        HP = maxHP;
    setStatus();
}

void sprite::drawHpBar(void)
{
    int a = 10*HP/maxHP;
    cout << "[";
    for(int i=0; i<a; i++)
        cout << "#";
    for(int i=a; i<10; i++)
        cout << "-";
     cout << "] ";
}

void sprite::show(void)
{
    cout << "[" << name << "] ";
    cout << "狀態：" << status << endl;
    drawHpBar();
    cout << "HP: " << HP  << "/" << maxHP << endl;
    cout << endl;
}

void attack(sprite &s, int n)
{
    cout << s.getName() << " 受到 " << n << " 點的傷害。" << endl;
    cout << endl;
    s.decHP(n);
}


int main()
{
    sprite fighter1("David");
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
```

```
    return 0;
}
```

```
David 受到 6 點的傷害。
[David] 狀態：受傷
[#######---] HP: 14/20


David 受到 6 點的傷害。


[David] 狀態：重傷
[####------] HP: 8/20


David 受到 6 點的傷害。


[David] 狀態：瀕死
[#---------] HP: 2/20


David 受到 6 點的傷害。


[David] 狀態：死亡
[----------] HP: 0/20
```

二、將類別放在獨立的檔案中

在前一個例子裡，我們把 sprite 類別和 main 放在同一個檔案裡，在程式碼短的情形下是沒問題的。但在動輒使用到數十乃至上百個類別的程式裡，這麼做就不切實際了，會造成維護和分工上的困擾。

我們可以將sprite類別移到獨立的檔案裡，這樣修改 sprite 時就可以專注在 sprite 的程式碼，main.cpp 裡也不會顯得凌亂，建置專案時的效率也會更好。

sprite.h

```
#ifndef SPRITE_H_INCLUDED
#define SPRITE_H_INCLUDED

#include <iostream>

using namespace std;

class sprite
{
private:
    string name;
    string status;
    int hp;
    int maxHp;

    void setStatus(void);  // 根據 hp 設定健康狀態
    void drawHpBar(void);  // 繪製 hp 長條圖
public:
    sprite(string _name);
    string getName();      // 取得姓名
    void decHP(int n);     // 減少 HP
    void addHP(int n);     // 增加 HP
    void show(void);       // 顯示狀態
};

#endif // SPRITE_H_INCLUDED
```

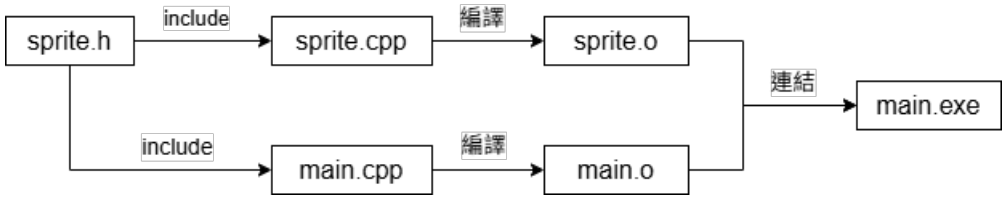
sprite.cpp

```
#include "sprite.h"

void sprite::setStatus(void)
{
    if(HP >= maxHP*0.9)
        status = "健康";
    else if(HP >= maxHP*0.6)
        status = "受傷";
    else if(HP >= maxHP*0.3)
        status = "重傷";
    else if(HP > 0)
        status = "瀕死";
    else
        status = "死亡";
}

sprite::sprite(string _name)
{
    name = _name;
    HP = maxHP = 20;
    status = "健康";
}

string sprite::getName()
{
    return name;
}

void sprite::decHP(int n)
{
    HP = HP - n;
    if(HP<0)
        HP = 0;
    setStatus();
}

void sprite::addHP(int n)
{
    HP = HP + n;
    if(HP > maxHP)
        HP = maxHP;
    setStatus();
}

void sprite::drawHpBar(void)
{
    int a = 10*HP/maxHP;
    cout << "[";
    for(int i=0; i<a; i++)
        cout << "#";
    for(int i=a; i<10; i++)
        cout << "-";
     cout << "] ";
}

void sprite::show(void)
{
    cout << "[" << name << "] ";
    cout << "狀態：" << status << endl;
    drawHpBar();
    cout << "HP: " << HP  << "/" << maxHP << endl;
    cout << endl;
}
```

main.cpp

```
#include <iostream>
#include "sprite.h"

using namespace std;

void attack(sprite &s, int n)
{
    cout << s.getName() << " 受到 " << n << " 點的傷害。" << endl;
    cout << endl;
    s.decHP(n);
}

int main()
{
    sprite fighter1("David");
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();
    attack(fighter1, 6);
    fighter1.show();

    return 0;
}
```

建置時的檔案相依性



10-4 Class 練習題

練習一：電影票券（`Ticket`）

情境說明

你正在設計一套電影院售票系統。每張票券記錄了電影名稱、座位號碼、票價，以及是否已被使用。票券一旦使用就不能再次入場；票價不能設為負數。

規格列表

成員變數（皆為 `private`）

變數名稱	型態	說明
`movieName`	`string`	電影名稱
`seatNumber`	`int`	座位號碼
`price`	`int`	票價（元）
`used`	`bool`	是否已使用

建構子

建構子	初始值
`Ticket()`	`movieName="未命名"`、`seatNumber=0`、`price=0`、`used=false`
`Ticket(string movieName, int seatNumber, int price)`	依參數設定，`used=false`

> “兩個建構子的參數名稱與成員變數相同，需使用 `this->` 或初始化列表。

Getter（皆加 `const`）

函式	回傳型態	說明
`getMovieName()`	`string`	回傳電影名稱
`getSeatNumber()`	`int`	回傳座位號碼
`getPrice()`	`int`	回傳票價
`isUsed()`	`bool`	回傳是否已使用

Setter（含驗證）

函式	驗證規則
`setPrice(int price)`	`price < 0` 時印出 `"票價不能為負數！"` 並放棄修改

其他成員函式

函式	回傳	說明
`use()`	`bool`	若已使用，印出 `"此票券已使用過！"` 並回傳 `false`；否則將 `used` 設為 `true`，印出入場訊息並回傳 `true`
`print() const`	`void`	印出票券所有資訊（格式見預期輸出）

`main()`

```
int main() {
    Ticket t1("星際大戰", 12, 280);
    Ticket t2;
```

```
    t1.print();
    t2.print();

    t1.use();
    t1.use();           // 重複使用

    t1.setPrice(-100);  // 非法
    t1.setPrice(250);   // 合法

    cout << t1.getMovieName() << " 目前票價：" << t1.getPrice() << " 元" << endl;

    t1.print();
}
```

預期輸出

```
[票券] 星際大戰 | 座位 12 | 票價 280 元 | 狀態：未使用
[票券] 未命名 | 座位 0 | 票價 0 元 | 狀態：未使用
入場成功！電影：星際大戰，座位：12
此票券已使用過！
票價不能為負數！
星際大戰 目前票價：250 元
[票券] 星際大戰 | 座位 12 | 票價 250 元 | 狀態：已使用
```

練習二：水壺（`Pitcher`）

情境說明

你設計一個登山水壺管理程式。每個水壺有名稱、最大容量和目前水量。注水時若超過容量會溢出；倒水時若水量不足則失敗；容量不能設得比目前水量還小。

規格列表

成員變數（皆為 `private`）

變數名稱	型態	說明
`name`	`string`	水壺名稱
`capacity`	`int`	最大容量（ml）
`current`	`int`	目前水量（ml）

建構子

建構子	初始值
`Pitcher()`	`name="水壺"`、`capacity=1000`、`current=0`
`Pitcher(string name, int capacity)`	依參數設定，`current=0`

Getter（皆加 `const`）

函式	回傳型態	說明
`getName()`	`string`	回傳水壺名稱
`getCapacity()`	`int`	回傳最大容量
`getCurrent()`	`int`	回傳目前水量

Setter（含驗證）

函式	驗證規則
`setCapacity(int capacity)`	`≤ 0` 印出 `"容量必須大於 0！"`；新容量 `< current` 印出 `"新容量小於目前水量，無法設定！"`

其他成員函式

函式	回傳	說明
`fill(int amount)`	`void`	注水；若超出容量則補滿並印出溢出量，否則正常注水
`pour(int amount)`	`bool`	倒水；水量不足回傳 `false` 並印出提示，否則正常倒出回傳 `true`
`status() const`	`void`	印出水壺狀態，含百分比（格式見預期輸出）

`main()`

```
int main() {
    Pitcher p1("運動水壺", 750);
    Pitcher p2;

    p1.status();
    p2.status();

    p1.fill(500);
    p1.fill(400);       // 會溢出

    p1.pour(200);
    p1.pour(600);       // 水量不足

    p1.setCapacity(-1); // 非法
    p1.setCapacity(100);// 非法（小於目前水量 550）

    cout << p1.getName() << " 容量：" << p1.getCapacity() << " ml" << endl;

    p1.status();
}
```

預期輸出

```
[運動水壺] 0 / 750 ml ( 0% )
[水壺] 0 / 1000 ml ( 0% )
運動水壺 注入 500 ml。
運動水壺 已裝滿，多餘 150 ml 溢出。
運動水壺 倒出 200 ml。
運動水壺 水量不足，無法倒出 600 ml！
容量必須大於 0！
新容量小於目前水量，無法設定！
運動水壺 容量：750 ml
[運動水壺] 550 / 750 ml ( 73% )
```

練習三：遊戲計分板（`ScoreBoard`）

情境說明

你正在設計一款小遊戲的計分系統。計分板記錄玩家名稱、歷史最高分、當前分數和剩餘生命數。每次得分若超過歷史最高分則更新紀錄；失去所有生命時宣告遊戲結束；重新開始時分數歸零但最高分保留。

規格列表

成員變數（皆為 `private`）

變數名稱	型態	說明
`playerName`	`string`	玩家名稱
`highScore`	`int`	歷史最高分
`currentScore`	`int`	當前分數
`lives`	`int`	剩餘生命數

建構子

建構子	初始值
`ScoreBoard()`	`playerName="玩家"`、`highScore=0`、`currentScore=0`、`lives=3`
`ScoreBoard(string playerName, int lives)`	依參數設定，`highScore=0`、`currentScore=0`

Getter（皆加 `const`）

函式	回傳型態	說明
`getPlayerName()`	`string`	回傳玩家名稱
`getHighScore()`	`int`	回傳歷史最高分
`getCurrentScore()`	`int`	回傳當前分數
`getLives()`	`int`	回傳剩餘生命數

Setter（含驗證）

函式	驗證規則
`setPlayerName(string playerName)`	名稱為空字串時印出 `"玩家名稱不能為空！"` 並放棄修改

其他成員函式

函式	回傳	說明
`addScore(int points)`	`void`	加分（`≤0` 印出提示拒絕）；若 `currentScore > highScore` 則更新並印出新紀錄訊息
`loseLife()`	`bool`	扣一條命；`lives=0` 時印出遊戲結束訊息並回傳 `false`，否則回傳 `true`
`reset()`	`void`	`currentScore` 歸零、`lives` 回復 3，`highScore` 保留，印出提示
`print() const`	`void`	印出所有資訊（格式見預期輸出）

`main()`

```
int main() {
    ScoreBoard sb("小明", 3);
    ScoreBoard sb2;

    sb.print();
    sb2.print();

    sb.addScore(100);
    sb.addScore(250);
    sb.addScore(50);

    sb.loseLife();
    sb.loseLife();
    sb.loseLife();      // 第三條命，遊戲結束

    sb.setPlayerName(""); // 非法
    sb.setPlayerName("小明Pro");

    sb.reset();
    sb.print();

    sb.addScore(500);   // 超越舊最高分
    sb.print();
}
```

預期輸出

```
== 小明 == 當前：0 | 最高：0 | 命：3
== 玩家 == 當前：0 | 最高：0 | 命：3
新紀錄！最高分更新為 100
小明 得 100 分，目前：100
```

```
新紀錄！最高分更新為 350
小明 得 250 分，目前：350
新紀錄！最高分更新為 400
小明 得 50 分，目前：400
小明 失去一條命，剩餘：2
小明 失去一條命，剩餘：1
小明 失去一條命，剩餘：0
遊戲結束！小明 最終得分：400
玩家名稱不能為空！
小明Pro 重新開始，最高分保留：400
== 小明Pro == 當前：0 | 最高：400 | 命：3
新紀錄！最高分更新為 500
小明Pro 得 500 分，目前：500
== 小明Pro == 當前：500 | 最高：500 | 命：3
```

11-自己實作一個 vector 類別

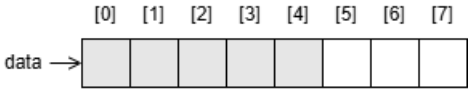
11-1 規劃我們的 Vec 類別

一、Vec 類別需要什麼？

在這個章節裡，我們嘗試自己建立一個簡單版的 vector，一個叫做 Vec 的類別。

我們至少需要以下三項屬性：

1. `data`: 指向一塊儲存資料的記憶體空間
2. `size`: 記錄目前 Vec 裡的元素數量
3. `capacity`: 記錄目前 Vec 裡已向作業系統要求配置的空間大小



size: 5
capacity: 8

接下來考慮需要對外公開的成員函數。

1. `Vec()`: 建構函數
2. `~Vec()`: 解構函數
3. `size()`: 回傳目前 Vec 裡有幾個元素
4. `capacity()`: 回傳目前 Vec 配置的記憶體大小最多可放幾個元素
5. `push_back(val)`: 將 val 附加在 Vec 的尾端
6. `pop_back()`: 將 Vec 尾端的元素移除

現在我們可以先完成這個類別的原型宣告如下:

[vec.h]

```
class Vec {
private:
    int *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(int val);
    void pop_back();
};
```

在成員函數的部分，我們使用 m_ 開頭的名稱，一方面用來提醒自己這是個 data member。一方面也是因為 data member 和 member function 不能用相同的名稱，我們不能有一個叫做 size 的 data member，同時又有一個叫做 size 的 member function。

11-2 實作 Vec 的細節

一、建構與解構函數

在建構函數中，我們要初始化 Vec 的 data member。因為剛建立好的 Vec 會是一個空的容器，所以一開始 m_size 和 m_capacity 都是 0。而 m_data 則先賦予它 nullptr 值，表示目前沒有指向任何記憶體。

```
Vec::Vec()
{
    m_data = nullptr;
    m_size = 0;
    m_capacity = 0;
}
```

在解構函數部分，我們先檢查是不有向作業系統要求配置記憶體，若有的話 m_data 將不會是 nullptr，而是會指向那塊記憶體。我們要釋放配置的記憶，把它還給作業系統。

```
Vec::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}
```

二、size 和 capacity

Vec 的 size 和 capacity 會隨著 push_back 放入愈來愈多的元素而成長，這兩個 data member 不應該被使用者修改，所以我們只提供兩個相應的成員函數用來回傳其值。

```
int Vec::size()
{
    return m_size;
}
```

```
int Vec::capacity()
{
    return m_capacity;
}
```

三、push_back

在 Vec 空間需要成長時，我們採用一個簡單的策略 - 每次成長為之前的 2倍大。

要留意的是，由於一開始 capacity 是 0，乘以 2 之後還是 0，所以要特別處理這個狀況，在第一次 push_back 時，capacity 要成長為 1。

在成長的部分，流程如下。

1. 配置一塊原來 2 倍大的記憶體空間
2. 把舊資料搬到新空間
3. 釋放舊空間記憶體，還給作業系統
4. 將 m_data 指向新的這塊記憶體

```
void Vec::push_back(int val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
```

```
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        int *new_data = new int[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}
```

四、pop_back()

pop_back 部分很簡單，只要把 size - 1 即可，不必清除那塊記憶體。下次 push_back 時，新資料就會蓋掉它。

```
void Vec::pop_back()
{
    m_size--;
}
```

11-3 測試 Vec 類別

目前我們的 Vec 類別如下：

[vec.h]

```
#ifndef VEC_H_INCLUDED
#define VEC_H_INCLUDED

class Vec {
private:
    int *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(int val);
    void pop_back();
};

Vec::Vec()
{
    m_data = nullptr;
    m_size = 0;
    m_capacity = 0;
}

Vec::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}

int Vec::size()
{
    return m_size;
}

int Vec::capacity()
{
    return m_capacity;
}

void Vec::push_back(int val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        int *new_data = new int[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
```

```
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}

void Vec::pop_back()
{
    m_size--;
}

#endif // VEC_H_INCLUDED
```

在主程式中引入其標頭檔來使用看看。

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    return 0;
}
```

注意看 capacity 和 size 的變化。

```
Cap:1 Size:1
Cap:2 Size:2
Cap:4 Size:3
Cap:4 Size:4
Cap:8 Size:5
```

11-4 重載 [] 運算子

現在我們還缺一個重要的功能，那就是存取 Vec 裡的值。

試著執行這段程式看看。

[main.cpp]

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    for(int i=0; i<a.size(); i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;

    return 0;
}
```

我們會得到這個錯誤訊息。

error: no match for 'operator[]' (operand types are 'Vec' and 'int')|

因為在第 23 行中的`cout << a[i]`，編譯器不知道 `Vec []` 要怎麼處理。

`[]`是一個運算子，每種資料型別的 `[]` 運算可能有所不同，我們必須自己實作 Vec 的 `[]` 運算子運算。

在 Vec 宣告的尾端加入這行重載 `[]` 運算子的成員函數。

```
    ...
    void pop_back();
    int& operator[] (int index); // <== 加入這行
};
```

在函數實作部分加入。

```
int& Vec::operator[] (int index)
{
    return m_data[index];
}
```

這段程式碼是在做什麼呢? 我們把它拆解成四個部分來看。

- `int&`（回傳型態）：注意那個 & 符號，它代表「參照」（Reference）。意思是它不只傳回那個數字的值，而是直接把那塊記憶體的「真身」給你。這樣你才能寫出像 a[0] = 10; 這樣的程式碼，直接修改裡面的數值。
- `Vec::`（所屬類別）：這代表這個功能是屬於 Vec 這個類別。
- `operator[]`（函數名稱）：這是 C++ 的特殊關鍵字。組合起來就是告訴編譯器：「我要重新定義中括號 `[]` 的功能」。
- `(int index)`（參數）：中括號裡面填的數字（索引值）。例如 a[3]，這個 index 就是 3。

現在可以執行這段程式來驗證了。

[main.cpp]

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    for(int i=0; i<a.size(); i++)
    {
        cout << a[i] << " ";  // 1 3 5 7 9
        a[i] = a[i]*2; // 把值修改為 2 倍
    }
    cout << endl;

    for(int i=0; i<a.size(); i++)
    {
        cout << a[i] << " ";  // 2 6 10 14 18
    }
    cout << endl;

    return 0;
}
```

```
Cap:1 Size:1
Cap:2 Size:2
Cap:4 Size:3
Cap:4 Size:4
Cap:8 Size:5
1 3 5 7 9
2 6 10 14 18
```

11-5 讓 Vec 可以儲存 int 以外的資料型別

目前我們 Vec 雖然可以動態成長，但是只能存放 int 型別的資料，這讓它變得很沒用。

明明只有型別不同，難道我們要寫一個 Vec_int 給 int 用，寫一個 Vec_double 給 double 用，.....。

在這裡我們要使用一個新東西 `樣版(template)`，class 是用來產生物件的 [模版]，而 template 則是用來產生 class 的 [模版]。

你可以這樣想，在宣告時寫 `Vec<int>` 這時 template 就會幫我們生成一個可以儲存 int 的 class，宣告時寫 `Vec<double>` 這時 template 就會幫我們生成一個可以儲存 double 的 class。

在程式碼中插入 `template<typename T>` 表示接下來這個 類別、函數 裡，看到 `T` 時，把它替代成指定的型別。

例如：

```
template<typename T>
class Vec {
private:
  T* m_data;
  int m_size;
  int m_capacity;
};
```

想像在主程式中，我們宣告一個 `Vec<double> a;`，會自動產生一個像這樣的 Vec class。

```
class Vec<double> {
private:
  double* m_data;
  int m_size;
  int m_capacity;
};
```

如此一來，我們就可以讓 Vec 裡儲存各種型別的元素，甚至是我們自己設計的型別。

修改後的 [vec.h]

```
#ifndef VEC_H_INCLUDED
#define VEC_H_INCLUDED

template<typename T>
class Vec {
private:
    T *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(T val);
    void pop_back();
    T& operator[] (int index);
};

template<typename T>
Vec<T>::Vec()
{
    m_data = nullptr;
```

```
    m_size = 0;
    m_capacity = 0;
}

template<typename T>
Vec<T>::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}

template<typename T>
int Vec<T>::size()
{
    return m_size;
}

template<typename T>
int Vec<T>::capacity()
{
    return m_capacity;
}

template<typename T>
void Vec<T>::push_back(T val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        T *new_data = new T[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}

template<typename T>
void Vec<T>::pop_back()
{
    m_size--;
}

template<typename T>
T& Vec<T>::operator[] (int index)
{
    return m_data[index];
}

#endif // VEC_H_INCLUDED
```