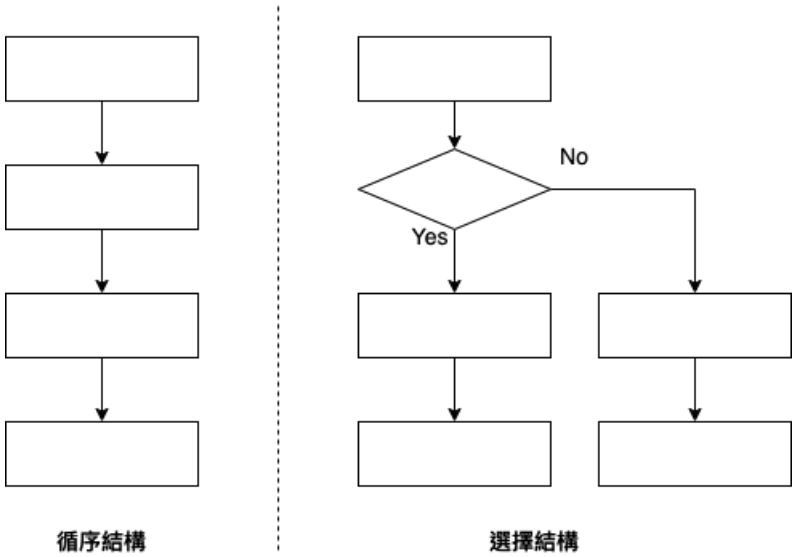
04-重覆結構

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

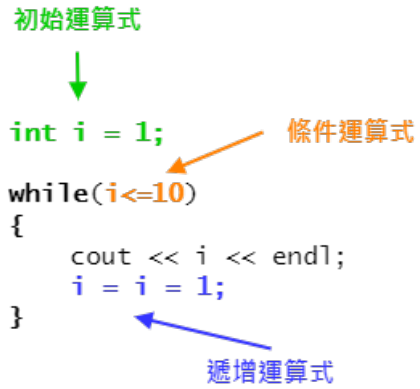
while 和 do...while 迴圈很適合用在「你知道什麼條件下迴圈要繼續或停止」，因為決定是否再繞一圈的就是一個條件判斷式。

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。