傳統 C 程式碼主題

F

We'll use a signal I have tried and found far-reaching and easy to yell.Waa-hoo!
—Zane Grey
It is quite a three-pipe problem.
—Sir Arthur Conan Doyle
But yet an union in partition.
—William Shakespeare

學習目標

在本章中，你將學到：

- 將鍵盤輸入重導成檔案輸入、將螢幕輸出重導成檔案輸出。
- 開發使用不定長度引數列的函式。
- 處理命令列引數。
- 處理程式中的非預期事件。
- 使用 C 風格的動態記憶體配置方法，替陣列動態配置記憶體。
- 使用 C 風格的動態記憶體配置方法，改變動態配置記憶體的大小。

本章綱要

F.1 簡介

本章將介紹幾個一般來說不會包含在初階課程中的主題，其中有許多功能是針對特殊的作業系統，特別是 UNIX/LINUX/Mac OS X 和/或 Windows。本章大部分的內容是針對其工作會接觸到傳統 C 程式碼的 C++程式設計師而設計的。

F.2 UNIX/Linux/Mac OSX 和 Windows 系統上的輸入/輸出重新導向

一般來說，程式的輸入是來自鍵盤 (標準輸入)，而輸出是到螢幕 (標準輸出)。在大多數的作業系統上 — UNIX、LINUX、Mac OS X 和 Windows — 我們可以將輸入**重新導向 (redirect)**，讓它來自檔案，或將輸出重新導向到檔案。我們可以不用到標準函式庫的檔案處理功能，就完成這兩種型態的重新導向。

有很多方法可以從 UNIX 命令列將輸入和輸出重新導向。假如有一個可執行檔 `sum`，它會一次輸入一個整數並將它們加總，直到設定檔案結束符號指示器爲止，接著會印出結果。通常使用者會從鍵盤輸入整數，然後輸入檔案結束符號組合鍵，表示沒有

資料要輸入了。我們也可以將輸入重新導向，把輸入儲存在檔案裡。例如，假如資料儲存在檔案 input 中，則以下指令

```
$ sum < input
```

會執行程式 sum；**重導輸入符號 (<，redirect input symbol)** 表示檔案 input 中的資料會被此程式當成輸入之用 (替代鍵盤)。Windows 系統的輸入重導方式也是一樣的。

注意，$代表的是 UNIX 命令列的提示符號。(UNIX 的提示符號會隨著系統或 shells 的不同而改變)。重新導向是屬於作業系統的功能，而非 C++的功能。

重導輸入的第二種方式稱爲**管線 (piping)**。**管線 (|，pipe)**可以將某一個程式的輸出重導成另一個程式的輸入。假設程式 random 輸出一連串的亂數，random 的輸出可以用以下的 UNIX 命令列，「管線式」導向至程式 sum。

```
$ random | sum
```

這會計算 random 所產生的整數的總和。管線可以在 UNIX、LINUX、Mac OS X 和 Windows 上執行。

程式的輸出可使用**重導輸出符號 (>，redirect output symbol)** 重導至檔案。(同樣的符號可以在 UNIX、LINUX、Mac OS X 和 Windows 上使用。) 例如，若想要將程式 random 的輸出導向至新檔案 out，請使用

```
$ random > out
```

此外，程式的輸出也可用**附加輸出符號 (>>，append output symbol)** 附加到某個現存的檔案的尾端。(同樣的符號可以在 UNIX、LINUX、Mac OS X 和 Windows 上使用。) 例如，若想要將程式 random 的輸出附加到上一個命令列所產生的檔案 out，請使用

```
$ random >> out
```

F.3　不定長度的引數列 (Variable-Length Argument Lists)

我們可以建立具有不定引數個數的函式。函式原型中的省略符號 (...，ellipsis) 表示此函式會接收不定個數的任何型別的引數 [1]。請注意省略詞一定要放在參數列的最後面，

---

[1] C++的程式設計師會使用函式多載來取代 C 程式設計師使用不定長度引數列來執行的大部分工作。

而且在省略詞前面至少要有一個引數。**不定引數標頭檔 <cstdarg> (variable arguments header)** 的巨集和定義 (圖 F.1)，提供建構不定長度引數列所需要的功能。

識別字	說明
va_list	存放 va_start、va_arg 和 va_end 等巨集所需資訊的型別。要存取不定長度引數列中的引數，程式必須先宣告一個型別 va_list 的物件。
va_start	在存取不定長度引數列之引數之前需呼叫的巨集。這個巨集初始化 va_list 型別的物件。而該物件可以用於 va_arg 和 va_end 巨集。
va_arg	在不定長度引數列中，將下一個引數的數值和型別展開成運算式的巨集。每次呼叫 va_arg 都會修改以 va_list 宣告的物件，所以這個物件會指向引數列中的下一個引數。
va_end	此巨集會讓具有不定引數列的函式（其不定引數列是以 va_start 巨集所參照）進行資源回收。

圖 F.1　　標頭檔 <cstdarg> 中定義的型別與巨集

圖 F.2 的程式示範接收不定個數引數的 average 函式。average 的第一個引數一定是進行平均值計算的數值個數，剩下的引數則必須爲型別 double。

函式 average 使用標頭檔 <cstdarg> 裡所有的定義和巨集。型別 **va_list** 的物件 list 會用於 **va_start**、**va_arg** 和 **va_end** 等巨集，用來處理函式 average 的不定長度引數列。此函式首先會呼叫巨集 va_start 來初始化 va_arg 和 va_end 所使用的物件 list。這個巨集會接收兩個引數－物件 list 以及引數列中位於...前最右邊的引數－在本例中爲 count (va_start 利用 count 來判斷不定長度引數列的起始位置)。

```
1  // Fig. F.2: figF_02.cpp
2  // Using variable-length argument lists.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdarg>
6  using namespace std;
7
8  double average( int, ... );
9
10 int main()
11 {
```

圖 F.2　　使用不定長度引數列

```
12     double double1 = 37.5;
13     double double2 = 22.5;
14     double double3 = 1.7;
15     double double4 = 10.2;
16
17     cout << fixed << setprecision( 1 ) << "double1 = "
18          << double1 << "\ndouble2 = " << double2 << "\ndouble3 = "
19          << double3 << "\ndouble4 = " << double4 << endl
20          << setprecision( 3 )
21          << "\nThe average of double1 and double2 is "
22          << average( 2, double1, double2 )
23          << "\nThe average of double1, double2, and double3 is "
24          << average( 3, double1, double2, double3 )
25          << "\nThe average of double1, double2, double3"
26          << " and double4 is "
27          << average( 4, double1, double2, double3, double4 )
28          << endl;
29  }  // end main
30
31  // calculate average
32  double average( int count, ... )
33  {
34     double total = 0;
35     va_list list;  // for storing information needed by va_start
36
37     va_start( list, count );
38
39     // process variable-length argument list
40     for ( int i = 1; i <= count; i++ )
41        total += va_arg( list, double );
42
43     va_end( list ); // end the va_start
44     return total / count;
45  }  // end function average
```

```
double1 = 37.5
double2 = 22.5
double3 = 1.7
double4 = 10.2

The average of double1 and double2 is 30.000
The average of double1, double2, and double3 is 20.567
The average of double1, double2, double3 and double4 is 17.975
```

圖 F.2　使用不定長度引數列 (續)

接下來，函式 `average` 重覆將不定長度引數列裡的引數加到變數 `total`。加到 `total` 的數值會藉著呼叫巨集 `va_arg`，從引數列中取出。巨集 `va_arg` 會接收兩個引數－物件 `list` 以及引數列中期望的數值型別－在本例中爲 `double`，並回傳引數的值。在返回之前，函式 `average` 會以 list 物件作爲引數，呼叫 `va_end` 巨集。最後，計算平均值然後返回 `main`。注意，在不定長度引數列中，我們只使用 `double` 引數做爲不定長度的部分。

不定長度引數列會將型別爲 `float` 的變數提升爲型別 `double`。這些引數列也會將比 `int` 小的整數型別變數提升爲 `int` 型別 (`int`、`unsigned`、`long` 和 `unsigned long` 型別的變數則不予理會)。

軟體工程的觀點 F.1

不定長度的引數列只能使用基本型別的變數和 `struct` 型別作爲引數，但是不包含 C++ 專屬的功能，像是 `virtual` 函式、建構子、解構子、參照、`const` 資料成員，以及 `virtual` 基本類別。

常見的程式設計錯誤 F.1

將省略符號 (`...`) 放在函式參數列的中間，是一種語法錯誤。省略符號只可以放在參數列的最後面。

F.4　使用命令列的引數

在許多系統中，我們可在 `main` 的參數列上加入 `int argc` 和 `char*argv[]`這兩個參數，進而從命令列傳遞引數給 `main`。參數 **`argc`** 會接收命令列引數的個數。參數 **`argv`** 則指向存放眞正命令列引數的 `char*` 陣列。命令列引數通常會用來列印引數、傳遞選項給程式、以及傳遞檔案名稱給程式。

圖 F.3 每次會逐字元地將某個檔案複製到另一個檔案。程式的執行檔稱爲 `copyFile`。在 UNIX 系統中，`copyFile` 程式的一般命令列如下：

```
$ copyFile input output
```

```
1  // Fig. F.3: figF_03.cpp
2  // Using command-line arguments
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main( int argc, char *argv[] )l
8  {
9     // check number of command-line arguments
10    if ( argc != 3 )
11       cout << "Usage: copyFile infile_name outfile_name" << endl;
12    else
13    {
14       ifstream inFile( argv[ 1 ], ios::in );
15
16       // input file could not be opened
17       if ( !inFile )
18       {
```

圖 F.3　使用命列列引數

```
19             cout << argv[ 1 ] << " could not be opened" << endl;
20             return -1;
21          }  // end if
22
23          ofstream outFile( argv[ 2 ], ios::out );
24
25          // output file could not be opened
26          if ( !outFile )
27          {
28             cout << argv[ 2 ] << " could not be opened" << endl;
29             inFile.close();
30             return -2;
31          } // end if
32
33          char c = inFile.get(); // read first character
34
35          while ( inFile )
36          {
37             outFile.put( c );   // output character
38             c = inFile.get();   // read next character
39          }  // end while
40       }  // end else
41    }  // end main
```

圖 F.3　使用命列列引數 (續)

這個命令列表示檔案 input 要複製到檔案 output。當程式執行時，如果 argc 不是 3 (copyFile 本身也算是一個引數)，則程式會印出錯誤訊息 (第 11 行)。否則，陣列 argv 會包含字串 "copyFile"、"input"和"output"。程式會將命令列第二個引數和第三個引數當成檔案名稱，藉著建立 ifstream 物件 inFile 和 ofstream 物件 outFile，開啓檔案 (第 14 和 23 行)。如果兩個檔案都開啓成功的話，則使用成員函式 get 從檔案 input 讀出字元，然後使用成員函式 put 寫入檔案 output，直到 input 的 end-of-file 標示設定爲止 (第 35–39 行)。然後程式便會結束執行。其結果就是檔案 input 的副本。注意，不是所有的電腦系統都支援跟 UNIX、LINUX、Mac OS X 和 Windows 一樣的命令列引數。例如，某些 VMS 和舊的麥金塔系統需要特殊的設定才能處理命令列引數。關於命令列引數的詳細資訊，請參閱系統的使用手冊。

F.5　編譯多個原始檔程式的注意事項

之前的章節曾經提過，我們可以建構含有多個原始程式檔的程式 (請參考第 9 章)。當我們建構這種程式時，必須考慮某些注意事項。例如，函式定義必須完整存放在一個檔案中－不能夠散佈到兩個或多個檔案中。

在第 6 章中，我們介紹過儲存類別和使用域的觀念。我們曾經學過宣告在任何函式定義之外的變數，其儲存類別預設爲 static，並且屬於全域變數。全域變數可以讓相同檔案中，此變數宣告之後的所有函式進行存取。全域變數也可以由其它檔案的函式加以存取，但是全域變數必須宣告在每個使用它的檔案中。例如，如果我們在某個檔案裡定義全域整數變數 flag，而在第二個檔案裡也想參照它的話，則第二個檔案必須含有以下的宣告：

```
extern int flag;
```

它必須放在該檔案使用該變數的位置之前。在這個宣告中，儲存類別修飾詞 extern 會告訴編譯器，變數 flag 可能定義在本檔案稍後的位置，或定義在其它檔案中。編譯器則會通知連結器，本檔案中含有指向變數 flag 的未解析參照。(編譯器並不知道定義 flag 的位置，因此它會讓連結器找出 flag)。如果連結器無法找到正確的函式定義，則它會產生一個錯誤訊息。如果找到正確的全域定義，則連結器會標示 flag 所在的位置，來解析這項參照。

增進效能的小技巧 F.1

全域變數可以增進效能，這是因爲它們可以被任何函式直接存取，不需要額外傳遞資料到函式中。

軟體工程的觀點 F.2

除非應用程式的效能是重要考量，或者變數代表的是必須分享的全域性資源，否則應該避免使用全域變數。它們會破壞最小權限原則，讓軟體變得難以維護。

如同其它程式檔的全域變數可以使用 extern 進行宣告，函式原型也可以將它的範圍擴展到其它檔案 (函式原型不需要使用 extern 修飾詞) 。藉著將函式原型放到每個呼叫此函式的檔案中，並且將每個原始碼檔案編譯，再將目的碼檔案連結，即可以達成此目的。函式原型告訴編譯器，它所指定的函式可能定義在本檔案稍後的位置，或定義在不同的檔案中。此外，編譯器不會嘗試解析這個函式的參照－這件工作將留給連結器處理。如果連結器無法找到函式定義，則它會產生一個錯誤訊息。

舉例說明使用函式原型來擴增函式的範圍：請考慮任何一個含有前置處理器命令 #include <cstring> 的程式，指令會將含有 strcmp 和 strcat 等函式的函式原型含入檔案。檔案內的其它函式便能夠使用 strcmp 和 strcat 來完成他們的工作。strcmp 和 strcat 定義於其它的檔案。我們不需知道他們定義的位置。我們只是在我們

的程式中重覆使用這些程式碼而已。連結器會自動解析這些函式的參照。這種程序讓我們能夠使用標準函式庫中的函式。

軟體工程的觀點 F.3

建立多個原始檔的程式可以增進軟體重覆使用性與良好的軟體工程。函式可以由許多的應用程式進行共用。在這種情況下，函式應該存放在他們自己的原始檔中，並且每個原始檔應該只包含該函式原型的標頭檔。這讓不同應用程式的程式設計師只要含入正確的標頭檔，然後再將他們的應用程式與對應的原始檔一起編譯，就可以重覆使用相同的程式碼。

可攜性的小技巧 F.1

某些系統不支援多於六個字元的全域變數名稱或函式名稱。當我們想讓程式具有可攜性時，應該要考慮到這個問題。

我們也可以將全域變數或函式的範圍限制在定義它的檔案中。當儲存類別修飾詞 `static` 應用到全域使用域變數或函式時，這可以避免它被不是定義在該檔案中的任何函式使用。這就稱爲**內部連結 (internal linkage)** 。前面沒有 `static` 的全域變數 (除非是 `const`) 或函式，會具有**外部連結 (external linkage)** －它們可以在其他的檔案中進行存取，只要該檔案含有正確的宣告或函式原型即可。

全域變數宣告

```
static double pi = 3.14159;
```

會產生一個型別 `double` 的變數 `pi`，將其初始值設定 3.14159，並且指出 `pi` 只能夠由相同檔案內的函式進行存取。

`static` 修飾詞通常會用於只由某個檔案內的函式所呼叫的公用函式。如果在某個檔案之外不需要使用某個函式，則我們可以用 `static` 來實行最小權限原則。如果檔案裡的某個函式在使用之前就已經定義過，則 `static` 必須加到該函式定義。否則，`static` 應該加到函式原型中。定義在不具名命名空間中的識別字也具有內部連結。C++標準中建議我們使用不具名命名空間來替代 `static`。

當我們建立含有多個原始檔的大型程式時，編譯該程式會很麻煩，且如果有一個檔案進行小幅度的修改，則整個程式都需要重新編譯。許多系統提供一種特殊的工具，可以用來編譯修改過的程式檔案。在 UNIX 系統上，這種工具就稱爲 `make`。`make` 會讀取 `makefile` 檔案，此檔案含有編譯和連結程式時所使用的指令。PC 上的 Borland C++和 Microsoft Visual C++則提供 make 工具和「projects」。請參考你的系統手冊，以獲得更多 `make` 工具的說明。

F.6　以 exit 和 atexit 結束程式的執行

一般的公用函式庫 (<cstdlib>) 提供其它方法來結束程式，而不是以 main 函式返回的方式來結束程式。函式 exit 強迫程式在正常執行狀況下結束。此函式通常會在偵測到輸入錯誤時，或程式所處理的檔案無法開啓時，才會結束程式的執行。

函式 atexit 則會向系統**註冊 (register)** 一個函式，當程式成功結束時便會呼叫該函式。也就是說，不管程式執行到 main 的最後，或是當呼叫 exit 終止執行，都會呼叫該函式。函式 **atexit** 以一個指向函式的指標 (亦即函式名稱) 做爲引數。程式結束時呼叫的函式不能夠有引數，也不能夠傳回數值。

函式 exit 具有一個引數。這個引數通常是符號常數 **EXIT_SUCCESS** 或符號常數 **EXIT_FAILURE**。如果 exit 是以 EXIT_SUCCESS 來進行呼叫，則系統定義的成功結束值會傳回給呼叫的環境。如果 exit 是以 EXIT_FAILURE 來進行呼叫，則函式會傳回系統定義的不成功結束值。當呼叫 exit 函式時，先前以 atexit 註冊的所有函式，都將按照他們註冊的相反順序逐次呼叫，所有與此程式有關的資料流都會清除並且關閉，然後將控制權傳回作業環境 (host environment)。圖 F.4 測試 exit 和 atexit 函式。程式提示使用者決定以 exit 或是以到達 main 尾端來結束程式的執行。請注意，在此兩種情況下，都會執行 print 函式。

```
1  // Fig. F.4: figF_04.cpp
2  // Using the exit and atexit functions
3  #include <iostream>
4  #include <cstdlib>
5  using namespace std;
6
7  void print();
8
9  int main()
10 {
11    atexit( print ); // register function print
12
13    cout << "Enter 1 to terminate program with function exit"
14         << "\nEnter 2 to terminate program normally\n";
15
16    int answer;
17    cin >> answer;
18
19    // exit if answer is 1
20    if ( answer == 1 )
21    {
```

圖 F.4　使用 exit 和 atexit 函式

```
22          cout << "\nTerminating program with function exit\n";
23          exit( EXIT_SUCCESS );
24       }  // end if
25
26       cout << "\nTerminating program by reaching the end of main"
27            << endl;
28    }  // end main
29
30    // display message before termination
31    void print()
32    {
33       cout << "Executing function print at program termination\n"
34            << "Program terminated" << endl;
35    }  // end function print
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

圖 F.4　使用 exit 和 atexit 函式 (續)

使用 exit 函式結束程式時，只會執行程式中 static 物件和全域物件的解構子。使用 abort 函式結束程式時，不會執行任何解構子。

F.7　volatile 型別修飾字

volatile 型別修飾字是用來定義可能會在程式之外被改變的變數 (不完全由程式內部控制的變數)。因此，編譯器不能執行最佳化 (像是加快程式執行或減少記憶體用量)，在執行這些最佳化時，編譯器必須確定變數的行爲只會被它可以觀察到的程式活動所影響。

F.8　整數常數和浮點常數的接尾詞

C++提供整數和浮點數的接尾詞，來指定整數和浮點常數的型別。整數接尾詞有：**u** 或 **U** 代表 **unsigned** 整數，**l** 或 **L** 代表 **long** 整數，而 **ul** 或 **UL** 代表 **unsigned long** 整數。以下常數的型別分別爲 unsigned、long 和 unsigned long：

```
174u
8358L
28373ul
```

假如某個整數常數沒有接尾詞，它的型別爲 int；假如此常數不能儲存在 int 中，就會儲存在 long 中。

浮點數的接尾詞有：**f** 或 **F** 代表 **float**，**l** 或 **L** 代表 **long double**。以下常數的型別分別爲 long double 和 float：

```
3.14159L
1.28f
```

沒有接尾詞的浮點常數，其型別會自動設定爲 double。假如常數的接尾詞不恰當，編譯器會發出警告或錯誤訊息。

F.9　訊號處理

外部的非預期事件 (event) 或**訊號 (signal)** 可能會讓程式永久終止。某些事件包括了**中斷** (**interrupts**，在 UNIX、LINUX、Mac OS X 或 Windows 系統裡輸入<Ctrl> C)、**非法指令 (illegal instructions)**、**分段錯誤 (segmentation violations)**、**作業系統的結束命令 (termination orders from the operating system)**、以及**浮點例外** (**floating-point exceptions**，除以 0 或乘上很大的浮點數值)。**訊號處理函式庫 (signal handling library)** 能夠以函式 signal 來**捕捉非預期事件 (trap unexpected events)**。函式 signal 有兩個引數，一個整數的訊號號碼以及一個指向訊號處理函式的指標。訊號可以由函式 raise 產生，它會以一個整數的訊號號碼來當成引數。圖 F.5 摘要列出定義在標頭檔 **<csignal>** 的標準訊號。接下來的範例會示範函式 signal 和 raise。

圖 F.6 的程式使用函式 signal 來捕捉中斷訊號 (SIGINT)。程式以 SIGINT 和一個指向 signalhandler 函式的指標來呼叫 signal。(還記得，函式的名稱就是指向函式起始位置的指標)。當產生型別爲 SIGINT 的訊號時，會呼叫函式 signalhandler，此函式會印出一段訊息，並且讓使用者選擇是否要繼續執行程式。如果使用者希望繼續執行程式，則訊號處理程式會呼叫 signal 來進行重新初始化 (某些系統會要求訊號處

理函式必須重新初始化)，然後控制權會傳回程式偵測到訊號的位置。本程式使用函式 `raise` 來模擬中斷訊號。程式會選取 1 到 50 之間的亂數。如果這個亂數等於 25，則會呼叫 `raise` 來產生訊號。一般而言，中斷訊號是由程式外部引發的。例如，在 UNIX、LINUX、Mac OS X 和 Windows 系統中，程式執行期間按下 <Ctrl> C 會產生中斷訊號來結束程式的執行。我們可以使用訊號處理來捕捉中斷訊號，並且避免程式終止執行。

識別字	說明
va_list	存放 va_start、va_arg 和 va_end 等巨集所需資訊的型別。要存取不定長度引數列中的引數，程式必須先宣告一個型別 va_list 的物件。
va_start	在存取不定長度引數列之引數之前需呼叫的巨集。這個巨集初始化 va_list 型別的物件。而該物件可以用於 va_arg 和 va_end 巨集。
va_arg	在不定長度引數列中，將下一個引數的數值和型別展開成運算式的巨集。每次呼叫 va_arg 都會修改以 va_list 宣告的物件，所以這個物件會指向引數列中的下一個引數。
va_end	此巨集會讓具有不定引數列的函式（其不定引數列是以 va_start 巨集所參照）進行資源回收。

圖 F.5　定義在標頭檔 `<csignal>` 中的訊號

```
1  // Fig. F.6: figF_06.cpp
2  // Using signal handling
3  #include <iostream>
4  #include <iomanip>
5  #include <csignal>
6  #include <cstdlib>
7  #include <ctime>
8  using namespace std;
9
10 void signalHandler( int );
11
12 int main()
13 {
14    signal( SIGINT, signalHandler );
15    srand( time( 0 ) );
16
17    // create and output random numbers
18    for ( int i = 1; i <= 100; i++ )
19    {
20       int x = 1 + rand() % 50;
21
22       if ( x == 25 )
```

圖 F.6　使用訊號處理

```
23            raise( SIGINT ); // raise SIGINT when x is 25
24
25         cout << setw( 4 ) << i;
26
27         if ( i % 10 == 0 )
28            cout << endl; // output endl when i is a multiple of 10
29      } // end for
30   } // end main
31
32   // handles signal
33   void signalHandler( int signalValue )
34   {
35      cout << "\nInterrupt signal (" << signalValue
36           << ") received.\n"
37           << "Do you wish to continue (1 = yes or 2 = no)? ";
38
39      int response;
40
41      cin >> response;
42
43      // check for invalid responses
44      while ( response != 1 && response != 2 )
45      {
46         cout << "(1 = yes or 2 = no)? ";
47         cin >> response;
48      }  // end while
49
50      // determine if it is time to exit
51      if ( response != 1 )
52         exit( EXIT_SUCCESS );
53
54      // call signal and pass it SIGINT and address of signalHandler
55      signal( SIGINT, signalHandler );
56   }  // end function signalHandler
```

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 100
```

```
   1   2   3   4
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

圖 F.6　使用訊號處理

F.10　使用 calloc 和 realloc 動態配置記憶體

在第 10 章，我們討論了如何利用 new 和 delete 運算子來進行 C++ 風格的動態記憶體配置。C++ 程式設計師應該使用 new 和 delete，而不是 C 的 **malloc** 和 **free** 函式 (標頭檔 <cstdlib>)。然而，大多數的 C++ 程式設計師會閱讀到許多傳統 C 程式碼，因此我們在這裡也介紹 C 風格的動態記憶體配置方法。

一般的公用程式庫 (<cstdlib>) 提供另外兩種動態記憶體配置函式－calloc 和 realloc。這些函式可以用來建立和修改**動態陣列 (dynamic arrays)**。第八章中曾提到，指向陣列的指標可以跟陣列一樣用索引來存取。因此，一個指向 calloc 所配置的連續記憶體的指標，可以如同陣列一樣地進行操作。函式 **calloc** 可以爲陣列動態配置記憶體空間，並且將記憶體的初值設定爲 0。calloc 的原型如下：

```
void *calloc( size_t nmemb, size_t size );
```

函式 calloc 會接收兩個引數，元素的個數 (nmemb) 以及每個元素的大小 (size) －並且將陣列所有元素的初始值都設定爲 0。函式會傳回一個指標指向配置的記憶體，如果無法配置記憶體，則程式會傳回 NULL 指標 (0)。

函式 **realloc** 會改變由 malloc、calloc 或 realloc 所配置的物件的大小。如果新配置的記憶體比先前配置的記憶體大，則原來物件中的內容並不會改變。否則，不被改變的內容僅止於新物件的大小。realloc 的原型如下：

```
void *realloc( void *ptr, size_t size );
```

realloc 函式會接收兩個引數，亦即指向原始物件的指標 (ptr) 以及新物件的大小 (size)。如果 ptr 爲 NULL，則 realloc 的動作會與 malloc 相同。如果 size 爲 0 而 ptr 不爲 0 的話，則程式會釋放此物件的記憶體空間。否則，如果 ptr 不爲 0 且 size 大於 0，則 realloc 會嘗試爲此物件配置一塊新的記憶體。如果無法配置新的記憶體空間，則 ptr 指向的物件並不會改變。函式 realloc 會傳回一個指向重新配置的記憶體空間的指標，或是一個 NULL 指標。

常見的程式設計錯誤 F.2

假如你在 malloc、calloc 或 realloc 所產生的指標上使用 delete 運算子，或是在 new 運算子產生的指標上使用 realloc 或 free，則會發生執行時期錯誤。

F.11　無條件分支：goto

在本書中，我們一直強調結構化程式設計的重要性，它可以建立容易除錯、維護和修改的穩固軟體。但在某些情況下，效能的考量比堅持結構化程式設計更加重要。此時，我們會使用某些非結構化程式設計的技巧。例如，我們可以使用 break，在迴圈測試條件式變成 false 之前，終止重複結構。這樣一來，假如在迴圈終止之前，任務就已經完成，我們就可以節省不需要的重複迴圈。

另外一個非結構化程式設計的例子是 **goto 敘述** — 無條件分支。goto 敘述會改變程式的控制流程，跳到 goto 敘述所指定的**標籤 (lable)** 之後的第一個敘述。標籤是一個識別字後面加一個分號。標籤必須與指定它的 goto 敘述出現在同一個函式中。圖 F.7 使用 goto 敘述執行 10 個迴圈，每次印出計數器的值。將 count 的初始值設定爲 1 之後，程式會測試 count 來判斷它是否大於 10。(標籤 start 會被跳過，因爲標籤不會執行任何動作)。假如 count 大於 10，控制權會從 goto 轉移到 end 標籤之後的第一個敘述。否則，程式會印出 count 並將它遞增，接著控制權會從 goto 轉移到 start 標籤之後的第一個敘述。

```
1  // Fig. F.7: figF_07.cpp
2  // Using goto.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int count = 1;
10
11    start: // label
12       // goto end when count exceeds 10
13       if ( count > 10 )
14          goto end;
15
16       cout << setw( 2 ) << left << count;
17       ++count;
18
19       // goto start on line 17
20       goto start;
21
22    end: // label
23       cout << endl;
24 }  // end main
```

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

圖 F.7　使用 goto

在第 4 章和第 5 章中，我們曾說過，任何程式只需要三種控制結構就可以完成 — 循序敘述、選擇敘述、以及重複敘述。當我們遵守結構化程式設計的原則時，可能會產生很深的巢狀控制結構，很難有效率地跳離。在這些情況下，某些程式設計師會使用 goto 敘述，快速地跳離深層巢狀敘述。這樣可以避免跳離控結構時，需測試多重條件。

增進效能的小技巧 F.2

goto 敘述可以用來有效率地跳離深層控制結構，但是會讓程式碼難以閱讀和維護。我們非常不鼓勵使用它。

測試和除錯的小技巧 F.1

除非使用在以效能爲導向的應用程式中，否則不應該使用 goto 敘述。使用 goto 敘述是非結構化的，這也會讓程式較難以除錯、維護、修改和理解。

F.12 Unions

union (使用關鍵字 **union** 加以定義) 是一個記憶體區域，隨著時間的不同，可以存放不同型別的物件。然而，在任何一個時間點，union 最多只能包含一個物件，因爲 union 中的成員分享同一個儲存空間。你必須保證 union 中的資料是以正確資料型別的成員名稱進行參照。

常見的程式設計錯誤 F.3

假如你不是以最後儲存到 union 的成員來做參照，其結果是未定義的。程式會以不同的型別來對待儲存在其中的資料。

可攜性的小技巧 F.2

如果以某種型別將資料存到 union 裡，而卻以另一種型別來參照該資料，則其結果會隨系統而有所差異。

在程式執行的不同時間點，有些物件可能是彼此毫無關連的，但是有一些卻有關聯。因此，union 會共用儲存空間，以免非使用中的物件浪費了儲存空間。儲存一個 union 所需的位元組數至少要能夠放得下此 union 最大的成員。

增進效能的小技巧 F.3

使用 union 以節省儲存空間。

可攜性的小技巧 F.3

儲存 union 所需的空間是視系統實作而定的。

宣告 union 的格式和 struct 或 class 一樣。例如

```
union Number
{
   int x;
   double y;
};
```

表示 Number 是一個 union 型別，它的成員爲 int x 和 double y。union 定義必須位在所有使用到它的函式之前。

軟體工程的觀點 F.4

和 struct 或是 class 宣告一樣，union 宣告也只是產生了一種新的型別而已。將 union 或 struct 宣告放在任何函式之外，並不會因此建立一個全域變數。

可對 union 執行的內建操作有：將某一 union 設定給另一個同型別的 union、取得一個 union 變數的位址(&)、以及使用結構成員運算子 (.) 和結構指標運算子 (->) 來存取 union 的成員。union 不能互相比較。

常見的程式設計錯誤 F.4

比較 union 是一種編譯錯誤，因爲編譯器不知道哪一個是正在作用中的成員，因此不知道要將哪一個成員互相比較。

union 和 class 的類似處在於它們都有建構子，可以初始化其成員。假如 union 沒有建構子，它可以使用另一個同型別的 union 來進行初始化、使用其型別爲第一個成員之型別的運算式來進行初始化，或是使用其型別爲第一個成員之型別的初始值 (以大括號包圍) 來進行初始化。union 可以擁有其他成員函式 (像是解構子)，但是 union 的成員函式不能宣告爲 virtual。union 的成員預設爲 public。

常見的程式設計錯誤 F.5

在宣告中初始化 uniion 時，假如其初始值或運算式的型別與 union 第一個成員的型別不同，則是一個編譯錯誤。

union 不能做爲繼承階層中的基本類別 (類別不能衍生自 union)。union 可以擁有物件做爲其成員，前提是這些物件不能具有建構子、解構子或是多載的指定運算子。union 的資料成員不能宣告爲 static。

圖 F.8 利用型別爲 union Number 的變數 value，顯示儲存在 union 中的值是 int 也是 double。此程式的輸出會隨系統而有所差異。由程式的輸出可知，double 值的內部表示法可能和 int 差異頗大。

```
1  // Fig. F.8: figF_08.cpp
2  // An example of a union.
3  #include <iostream>
4  using namespace std;
5
6  // define union Number
7  union Number
8  {
9     int integer1;
10    double double1;
11 };  // end union Number
12
13 int main()
14 {
15    Number value; // union variable
16
17    value.integer1 = 100; // assign 100 to member integer1
18
19    cout << "Put a value in the integer member\n"
20         << "and print both members.\nint:    "
21         << value.integer1 << "\ndouble: " << value.double1
22         << endl;
23
24    value.double1 = 100.0;  // assign 100.0 to member double1
25
26    cout << "Put a value in the floating member\n"
27         << "and print both members.\nint:    "
28         << value.integer1 << "\ndouble: " << value.double1
29         << endl;
30 }  // end main
```

```
Put a value in the integer member
and print both members.
int:   100
double: -9.25596e+061
Put a value in the floating member
and print both members.
int:   0
double: 100
```

圖 F.8　以兩種成員資料型別印出 union 的值

匿名 union (anonymous union) 是一個不具有型別名稱的 union，它在其結束分號之前，不會嘗試定義物件或指標。這種 union 不會建立型別，但會建立一個不具名的物件。匿名 union 的成員必須在宣告此 union 的使用域中直接存取，就像其他區域變數一樣，不需要使用點號運算子 (.) 或箭號運算子 (->)。

匿名 union 有一些限制：只能包含資料成員，所有的成員都必須是 public，宣告爲全域的匿名 union 必須宣告爲 static。圖 E.9 示範使用匿名 union。

```
// Fig. F.9: figF_09.cpp
// Using an anonymous union.
#include <iostream>
using namespace std;

int main()
{
   // declare an anonymous union
   // members integer1, double1 and charPtr share the same space
   union
   {
      int integer1;
      double double1;
      char *charPtr;
   };  // end anonymous union

   // declare local variables
   int integer2 = 1;
   double double2 = 3.3;
   char *char2Ptr = "Anonymous";

   // assign value to each union member
   // successively and print each
   cout << integer2 << ' ';
   integer1 = 2;
   cout << integer1 << endl;

   cout << double2 << ' ';
   double1 = 4.4;
   cout << double1 << endl;

   cout << char2Ptr << ' ';
   charPtr = "union";
   cout << charPtr << endl;
}  // end main
```

```
1 2
3.3 4.4
Anonymous union
```

圖 F.9　使用匿名 union

F.13　連結規格

我們可能會在 C++ 程式中呼叫使用 C 撰寫和編譯的函式。如 6.17 節所述，C++ 會將函式名稱編碼，以實現型別安全連結。然而 C 不會將它的函式名稱編碼。因此，假如我們嘗試將 C 程式碼與 C++ 程式碼連結，將會無法辨識使用 C 所編譯的函式，因爲 C++ 程式會期望得到一個特殊編碼的函式名稱。C++ 提供**連結規格 (linkage**

specifications)，讓你可以通知編譯器，某個函式是以 C 編譯器所編譯的，避免函式名稱被 C++ 編譯器編碼。當建立了大型的特殊函式庫，或是使用者無法取得原始碼以重新編譯，或是沒有時間將函式庫函式從 C 轉換到 C++ 時，連結規格是很有用的。

要通知編譯器，有一個或多個函式已經使用 C 編譯過了，你可以撰寫以下函式原型：

```
extern "C" function prototype    // single function
extern "C"   // multiple functions
{
   function prototypes
}
```

這些宣告會通知編譯器，指定的函式不是以 C++ 編譯的，因此不應該對連結規格中指定的函式使用名稱編碼。這些函式可以與程式正確連結。C++ 環境通常會包含標準 C 函式庫，你不需要對這些函式使用連結規格。

F.14　總結

本附錄介紹許多傳統 C 程式碼。我們討論了如何將鍵盤輸入重導成檔案輸入、將螢幕輸出重導成檔案輸出。我們也介紹了可變長度引數列、命令列引數，以及如何處理非預期事件。你也學到了如何動態配置記憶體以及改變動態配置記憶體的大小。

摘要

F.2　UNIX/Linux/Mac OSX 和 Windows 系統上的輸入/輸出重新導向

- 在許多電腦系統 (UNIX、LINUX、Mac OS X 或 Windows 系統) 中，我們可以重新導向程式的輸入和輸出。輸入可以在 UNIX、LINUX、Mac OS X 或 Windows 命令列使用重導輸入符號 (<) 或管線 (|) 來執行重新導向。輸出可以在 UNIX、LINUX、Mac OS X 或 Windows 命令列使用重導輸出符號 (>) 或附加輸出符號 (>>) 來執行重新導向。重導輸出訊號會將程式的輸出存放到檔案中，並且將附加輸出符號會將輸出附加到檔案的尾端。

F.3　不定長度的引數列 (Variable-Length Argument Lists)

- 不定引數標頭檔 `<cstdarg>` 的巨集和定義，提供建構不定長度引數列所需要的功能。
- 函式原型中的省略符號 (`...`，ellipsis) 表示此函式會接收不定個數的任何型別的引數。
- `va_list` 型別適合存放 `va_start`、`va_arg` 和 `va_end` 等巨集所需要的資訊。要存取不定長度引數列中的引數，程式必須先宣告一個型別 `va_list` 的物件。

- 在存取不定長度引數列之前，程式必須先呼叫 va_start 巨集。這個巨集初始化 va_list 型別的物件。而該物件可以用於 va_arg 和 va_end 巨集。
- 巨集 va_arg 會展開成，具有不定長度引數列中下一個引數的數值以及型態的運算式。每次呼叫 va_arg 都會修改以 va_list 宣告的物件，所以這個物件會指向引數列中的下一個引數。
- va_end 巨集會輔助具有不定引數列的函式的傳回動作，該引數列會以 va_start 參照。

F.4 使用命令列的引數

- 在許多系統中，特別是 Windows、UNIX、LINUX 和 Mac OS X，我們可在 main 的參數列上加入 int argc 和 char*argv[]這兩個參數，進而從命令列傳遞引數給 main。參數 argc 會接收命令列引數的個數。參數 argv 則是存放眞正命令列引數的 char*陣列。

F.5 編譯多個原始檔程式的注意事項

- 函式定義必須完整存放在一個檔案中－它不能夠散佈在兩個或多個檔案中。
- 全域變數必須宣告在每個使用它的檔案中。
- 函式原型可以用來宣告其他程式檔案中的函式。(函式原型不需要使用 extern 修飾詞)。藉著將函式原型放到每個呼叫此函式的檔案中，並且將每個原始碼檔案編譯，再將目的碼檔案連結，即可以達成此目的。
- 當儲存類別修飾詞 static 應用到全域使用域變數或函式時，這可以避免它被不是定義在該檔案中的任何函式使用。這就稱爲內部連結。前面沒有 static 的全域變數或函式，會具有外部連結 (external linkage) －它們可以在其他的檔案中進行存取，只要該檔案含有正確的宣告或函式原型即可。
- static 修飾詞通常會用於只由某個檔案內的函式所呼叫的公用函式。如果在某個檔案之外不需要使用某個函式，則我們可以用 static 來實行最小權限原則。
- 當我們建立含有多個原始檔的大型程式時，編譯該程式會很麻煩，且如果有一個檔案進行小幅度的修改，則整個程式都需要重新編譯。許多系統提供一種特殊的公用程式，可以用來編譯修改過的程式檔案。在 UNIX 系統上，這種工具就稱爲 make。make 會讀取 makefile 檔案，此檔案含有編譯和連結程式時所使用的指令。

F.6 以 exit 和 atexit 結束程式的執行

- 函式 exit 強迫程式在正常執行狀況下結束。
- 函式 atexit 則會向系統註冊一個函式，當程式成功結束時便會呼叫該函式。也就是說，不管程式執行到 main 的最後，或是當呼叫 exit 終止執行，都會呼叫該函式。

- 函式 atexit 以一個指向函式的指標 (亦即函式名稱) 做爲引數。程式結束時呼叫的函式不能夠有引數，也不能夠傳回數值。
- 函式 exit 會接收一個引數，這個引數通常是符號常數 EXIT_SUCCESS 或符號常數 EXIT_FAILURE。如果 exit 是以 EXIT_SUCCESS 來進行呼叫，則系統定義的成功結束值會傳回給呼叫的環境。如果 exit 是以 EXIT_FAILURE 來進行呼叫，則函式會傳回系統定義的不成功結束值。
- 當呼叫 exit 函式時，先前以 atexit 註冊的所有函式，都將按照他們註冊的相反順序逐次呼叫，所有與此程式有關的資料流都會清除並且關閉，然後將控制權傳回作業環境 (host environment)。

F.7　volatile 型別修飾字

- volatile 修飾字可以防止變數的最佳化，因爲此變數可能在程式外部被修改。

F.8　整數常數和浮點常數的接尾詞

- C++ 提供整數和浮點數的接尾詞，來指定整數和浮點常數的型別。整數接尾詞有：u 或 U 代表 unsigned 整數，l 或 L 代表 long 整數，而 ul 或 UL 代表 unsigned long 整數。如果整數常數沒有接尾詞，則其型別就是第一個放得下該數值的型別 (首先是 int，再來是 long int)。浮點數的接尾詞有：f 或 F 代表 float，l 或 L 代表 long double。沒有接尾詞的浮點常數，其型別會自動設定爲 double。

F.9　訊號處理

- 訊號處理函式庫提供函式註冊的功能，以函式 signal 來捕捉非預期事件。函式 signal 有兩個引數，一個整數的訊號號碼以及一個指向訊號處理函式的指標。
- 訊號也可以由函式 raise 加上一個整數引數來加以產生。

F.10　使用 calloc 和 realloc 動態配置記憶體

- 一般的公用程式庫 (<cstdlib>) 提供兩種動態記憶體配置函式－calloc 和 realloc。這些函式可以用來建立動態陣列。
- 函式 calloc 會接收兩個引數，元素的個數 (nmemb) 以及每個元素的大小 (size) －並且將陣列所有元素的初始值都設定爲 0。函式會傳回一個指標指向配置的記憶體，如果無法配置記憶體，則程式會傳回 NULL 指標。
- 函式 realloc 會改變由 malloc、calloc 或 realloc 所配置的物件的大小。如果新配置的記憶體比先前配置的記憶體大，則原來物件中的內容並不會改變。

- realloc 函式會接收兩個引數，亦即指向原始物件的指標 (ptr) 以及新物件的大小 (size)。如果 ptr 爲 NULL，則 realloc 的動作會與 malloc 相同。如果 size 爲 0 而 ptr 不爲 NULL 的話，則程式會釋放此物件的記憶體空間。否則，如果 ptr 不爲 NULL 且 size 大於 0，則 realloc 會嘗試爲此物件配置一塊新的記憶體。如果無法配置新的記憶體空間，則 ptr 指向的物件並不會改變。函式 realloc 會傳回一個指向重新配置的記憶體空間的指標，或是一個 NULL 指標。

F.11 無條件分支：goto

- goto 敘述會改變程式的控制流程。程式會從 goto 敘述的標籤之後的第一個敘述繼續執行。
- 標籤是一個識別字後面加一個分號。標籤必須與指定它的 goto 敘述出現在同一個函式中。

F.12 Unions

- union 是一種資料型別，它的成員會共用相同的儲存空間。其成員幾乎可以是任何型別。替 union 保留的儲存空間必須能儲存它的最大成員。在大部分的情況下。union 中會含有兩種或更多個資料型別。然而在相同時刻，只有一個成員 (亦即只有一種資料型別) 可進行參照。
- 宣告 union 的格式和 struct 一樣。
- union 可以以此 union 第一個成員之型別的數值來設初始值，或是以另一個具有同樣型別的 union 設定初始值。

F.13 連結規格

- C++ 提供連結規格 (linkage specifications)，讓你可以通知編譯器，某個函式是以 C 編譯器所編譯的，避免函式名稱被 C++ 編譯器編碼。
- 要通知編譯器，有一個或多個函式已經使用 C 編譯過了，你可以撰寫以下函式原型：

```
extern "C" function prototype    // single function

extern "C"   // multiple functions
{
   function prototypes
}
```

- 這些宣告會通知編譯器，指定的函式不是以 C++ 編譯的，因此不應該對連結規格中指定的函式使用名稱編碼。這些函式可以與程式正確連結。
- C++ 環境通常會包含標準 C 函式庫，你不需要對這些函式使用連結規格。

術語

附加輸出符號>> (append output symbol>>)
匿名 (anonymous)
argv
atexit
calloc
命列列引數 (command-line arguments)
const
<csignal>
<cstdarg>
動態陣列 (dynamic arrays)
事件 (event)
exit
EXIT_FAILURE
EXIT_SUCCESS
extern "C"
extem 儲存類別修飾字 (extem storage-class specifier)
外部連結 (external linkage)
float 接尾詞 (f 或 F) (float suffix；f or F)
浮點數例外 (floating-point exception)
free
goto 敘述 (goto statement)
I/O 重導 (I/O redirection)
非法指令 (illegal instruction)
內部連結 (internal linkage)
中斷 (interrupt)
標籤 (label)
連結規格 (Linkage Specifications)
long double 接尾詞 (l 或 L) (long double suffix；l or L)
long int 接尾詞 (l 或 L) (long integer suffix；l or L)
make
Makefile
malloc
管線 (|) [pipe (|)]
管線 (piping)
raise
realloc
重導 (redirect)
重導輸入符號 (<) [redirect input symbol (<)]
重導輸出符號 (>) [redirect output symbol (>)]
暫存器 (registers)
分段錯誤 (segmentation violation)
訊號 (signal)
signal
訊號處理函式庫 (signal handling library)
敘述 (statement)
static 儲存類別修飾字 (static storage-class specifier)
作業系統的結束命令 (termination orders from the operating system)
trap (捕捉)
捕捉非預期事件 (trap unexpected events)
union
union
unsigned 整數接尾詞 (u 或 U) (unsigned integer suffix；u or U)
unsigned long 整數接尾詞 (ul 或 UL) (unsigned long integer suffix；ul or UL)
va_arg
va_end
va_list
va_start
不定引數標頭檔 (variable arguments header)
不定長度的引數列 (variable-length argument list)
揮發性 (volatile)

自我測驗

F.1 填寫以下空格：

a) _____符號可用來將鍵盤輸入導向至檔案。

b) _____符號可用來將螢幕輸出導向至檔案。

c) _____符號可用來將程式的輸出附加到檔案的尾端。

d) _____可以用來將某個程式的輸出導向成另一程式的輸入。

e) 函式參數列上的_____表示此函式可能會接收不定個數的引數。

f) 在存取不定長度引數列之前，程式必須先呼叫_____巨集。

g) _____巨集可以用來存取不定長度引數列上的個別引數。

h) _____巨集提供函式資源回收的工作，該函式的不定長度引數列會以 va_start 巨集進行參考。

i) `main` 的_____引數接收命令列引數的個數。

j) `main` 的_____引數將命令列引數以字元字串加以儲存。

k) UNIX 的公用程式_____會讀取一個稱爲_____的檔案，此檔案中含有如何編譯和連結由多個原始檔所組成的程式。此公用程式只會重新編譯上一次編譯之後有修改過的檔案（或是標頭檔）。

l) 函式_____會強迫程式結束執行。

m) 函式_____會向系統註冊一個函式，當程式正常結束時，便會呼叫這個註冊的函式。

n) 整數和浮點數常數之後可以附加整數或浮點數_____，用來指定常數的型別。

o) 函式_____可以用來註冊函式以捕捉非預期事件。

p) 函式_____可以用在程式中產生訊號。

q) 函式_____可以爲陣列動態配置記憶體空間，並且將所有元素的初值設定爲 0。

r) 函式_____可以改變動態配置的記憶體空間。

s) _____包含一組變數，且這些變數在不同的時間點，共用相同的儲存空間。

t) _____關鍵字_____用來開始一個 `union` 的定義。

自我測驗解答

F.1 a) 重導輸入 (<)。b) 重導輸出 (>)。c) 附加輸出 (>>)。d) 管線 (|)。e) 省略符號 (`...`)。f) `va_start`。 g) `va_arg`。h) `va_end`。 i) `argc`。j) `argv`。k) `make`，`Makefile`。l) `exit`。m) `atexit`。n) 接尾詞。o) `signal`。p) `raise`。q) `calloc`。r) `realloc`。s) `union`。t) `union`。

習題

F.2 撰寫一個程式來計算一連串整數的乘積，這些整數會以不定長度引數列傳給函式 `product`。請以數個不同引數個數的呼叫，來測試你的程式。

F.3 撰寫一個程式，印出程式的命令列引數。

F.4 寫一個程式，將陣列依遞增或遞減順序排序。程式應該使用命令列引數，當收到`-a` 時，請依遞增排序，收到`-d` 時，請依遞減排序。[請注意：這是在 UNIX 中傳遞選項給程式的標準格式。]

F.5 請查閱你的系統說明手冊，找出訊號處理函式庫 (`<csignal>`) 所支援的訊號。請撰寫一個程式，替訊號 `SIGABRT` 和 `SIGINT` 提供訊號處理常式。此程式應該以下列的方式來測試這兩個訊號的捕捉狀況：呼叫函式 `abort` 產生一個 `SIGABRT` 型別的訊號，以及鍵入 <Ctrl> C 產生一個 `SIGINT` 型別的訊號。

F.6 撰寫一個程式來動態配置一個整數陣列，使用 `<cstdlib>` 的函式，不要使用 new 運算子。此陣列的大小應該由鍵盤加以輸入。陣列的元素也應該由鍵盤輸入的數值來加以設定。將陣列的數值列印出來。接下來，重新配置此陣列的記憶體，讓它的元素數目變爲目前的 1/2。列印陣列中的數值，來驗證一下是否目前陣列的前一半元素是相同的。

F.7 撰寫一個程式來接收兩個命令列引數，這兩個引數都是檔案名稱。程式每次都會從第一個檔案讀取一個字元，然後將所有字元以相反順序寫到第二個檔案。

F.8 撰寫一個程式，使用 `goto` 序數來模擬巢狀迴圈，使用星號印出正方形。您的程式只能使用下列 3 個輸出敘述：

```
cout << '*';
cout << ' ';
cout << endl;
```

F.9 請撰寫 `union Data` 的定義，其中包含 `char charcter1`，`short short1`，`long long1`，float float1，以及 `double double1`。

F.10 建立 `union Integer`，包含成員 `char c`、`short s`、`int i` 以及 `long l`。撰寫一個程式輸入型別爲 `char`、`short`、`int` 和 `long` 的數值，並將這些值存放到 `union Integer` 型別的變數。請以 char、`short`、`int` 和 `long` 對每個 `union` 變數進行列印。在每一種狀況下印出的值都是正確的碼？

F.11 建立 `union FloatingPoint`，包含成員 `float float1`、`double double1`、`long double longDouble`。撰寫一個程式輸入型別爲 `float`、`double` 和 `long double` 的數值，並將這些值存放到 `union FloatingPoint` 型別的變數。 請以 `float`、`double` 和 `long double` 對每個 `union` 變數進行列印。在每一種狀況下印出的值都是正確的碼？

F.12 假設以下的 `union` 宣告

```
union A
{
   double y;
   char *zPtr;
};
```

以下哪些敘述可以正確初始化 union？

a) `A p = b;  // b is of type A`
b) `A q = x;  // x is a double`
c) `A r = 3.14159;`
d) `A s = { 79.63 };`
e) `A t = { "Hi There!" };`
f) `A u = { 3.14159, "Pi" };`
g) `A v = { y = -7.843, zPtr = &x };`