Appendix E

其他運算子

為了避免本書過於龐大，本書主要章節並未談到兩類運算子。第一類是位元運算子（bitwise operator），處理值的個別位元，這些運算子皆繼承自 C。第二類是兩個成員提領運算子（dereferencing operator），這是 C++ 新加的運算子。第三類是 C++11 新加入的運算子，分別是 `alignof` 和 `noexcept`。本附錄將簡短地摘要這些運算子。

位元運算子

位元運算子是用來處理整數值的位元。例如，左移運算子將位元移向左方，而位元否定運算子，將每個 `0` 變成 `1`，`1` 變成 `0`。整個算起來 C++ 有 6 個這種運算子：`<<`，`>>`，`~`，`&`，`|`，和 `^`。

移位運算子

左移（left-shift）運算子的語法為：

```
value << shift
```

此處 *`value`* 為要移位的整數值，*`shift`* 為移位的位元數目，例如：

```
13 << 3
```

意思是將值 `13` 的所有位元向左移 3 位元。左移後空出的位元填入 0；移出的位元則捨棄（請參考圖 E.1）。

因為每個位元所在位置代表的值，正好是其右邊位元所代表之值的兩倍（參考附錄 A），所以每向左移動一位元，等於乘以 2，同理，移動兩位元等於乘以 2^2，移動 n 位元等於乘以 2^n。因此，`13 << 3` 之值是 13×2^3，結果是 104。

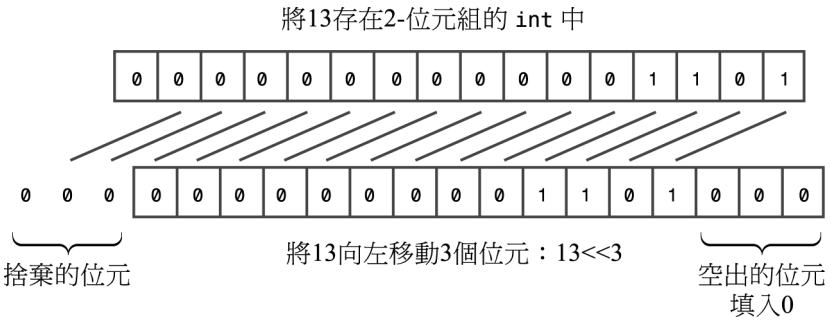

圖 E.1 左移運算子

左移運算子常見於組合語言，但是組合語言的左移運算子，直接更改暫存器內部的值，而 C++ 的左移運算子，可以在不改變原值下產生新值。例如敘述：

```
int x = 20;
int y = x << 3;
```

上例並未改變 x 之值，運算式 `x << 3`，亦如 `x + 3`，是利用 x 的值產生新值，x 本身並未改變。

如果希望運用左移運算子修改變數值，則必須配合指定運算子。使用方式可採用一般的指定運算子，或結合移位與指定的 <<= 運算子：

```
x = x << 4;           //regular assignment
y  <<= 2;              //shift and assign
```

右移（right shift）運算子（>>），則是向右移動位元，它的語法為：

```
value >> shift
```

此處 *value* 是要移位的整數值，*shift* 為移位的位元數目。例如：

```
17 >> 2
```

意思是將 17 值的所有位元，向右移 2 個位元。對於 unsigned 整數，右移後留下的空白填入 0，移出的位元則捨去。至於 signed 整數，右移留下的空白可能填入 0 或填入原先最左位元的值。這得視各編譯程式作法而定（請參考圖 E.2，此例是填入 0）。

向右移一位元等於整數值除以 2。一般來說，右移 n 位元等於整數值除以 2^n。

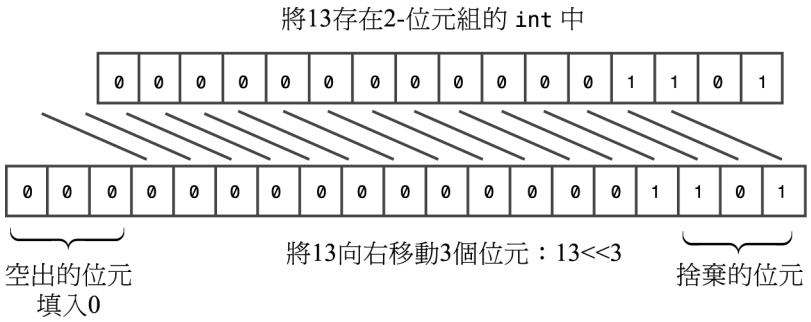

圖 E.2 右移運算子

如果希望移位後同時取代變數值，則 C++ 也提供右移-並-指定（right-shift-and-assign）運算子：

```
int q = 43;
q >>= 2;            //replace 43 by 43 >> 2, or 10
```

在某些系統上，以左移和右移運算子，執行整數乘以 2 或除以 2，結果可能會較一般乘除運算子迅速，但隨著編譯程式最佳化日益成熟，二者差異已變得很小。

邏輯位元運算子

邏輯位元運算子（logical bitwise operator）與一般的邏輯運算子很類似，唯一差別是前者是針對個別的位元，後者是針對整個值。例如一般的否定運算子（!）和位元的否定（或補數）運算子（~）。! 運算子將 true（非零值）值變成 false（零）值，將 false 值變成 true 值。而 ~ 運算子將每一位元值變成相反值（1 變 0，0 變 1）。例如，unsigned char 型態值 3：

```
unsigned char x = 3;
```

運算式 !x 的結果為 0。要看 ~x 之值，先以二進位表示 3：00000011，然後將每個 0 變成 1，每個 1 變成 0，結果變為 11111100，或是十進位值 252（請參考圖 E.3，這為 16 位元的範例）。新值稱為原始值的**補數**（complement）。

位元 OR 運算子（|）結合兩個整數值產生一個新整數值。如果二個值的對應位元皆為 1，或有一值的位元為 1，則新值的該位元為 1；如果二個值的對應位元皆為 0，則新值的該位元為 0（參考圖 E.4）。

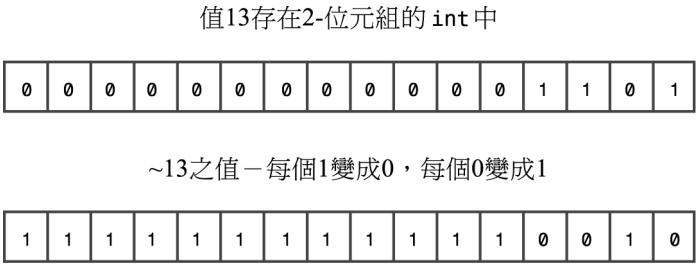

圖 E.3 位元否定運算子

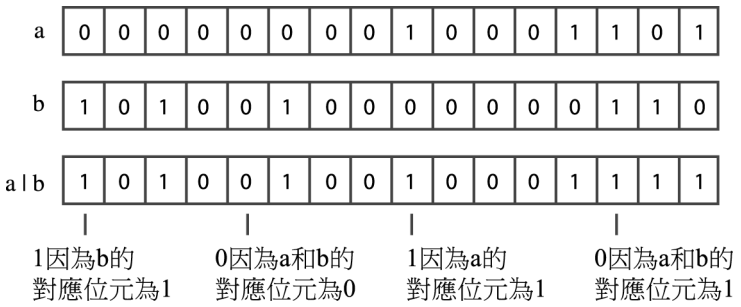

圖 E.4 位元 OR 運算子

表 E.1 總結 | 運算子的組合情形。

表 E.1 **b1 | b2** 之值

位元值	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	1

|= 運算子結合位元 OR 運算子和指定運算子：

```
a |= b;  // set a to a | b
```

位元 XOR 運算子（^）結合兩個整數值產生一個新整數值。如果二個值的對應位元其中有一位元為 1，另外一位元為 0，則新值的該位元為 1，如果二個值的對應位元皆為 0 或皆為 1，則新值的該位元為 0（請參考圖 E.5）。

表 E.2 總結 ^ 運算子的組合情形。

圖 E.5 位元 XOR 運算子

表 E.2 **b1 ^ b2** 之值

位元值	**b1 = 0**	**b1 = 1**
b2 = 0	0	1
b2 = 1	1	1

^= 運算子結合位元 XOR 運算子和指定運算子：

```
a ^= b;  // set a to a ^ b
```

位元 AND 運算子（&）將兩個整數值組合成一個新整數值。如果二個值的對應位元皆為 1，則新值的該位元為 1，其它情形則位元值等於 0（請參考圖 E.6）。

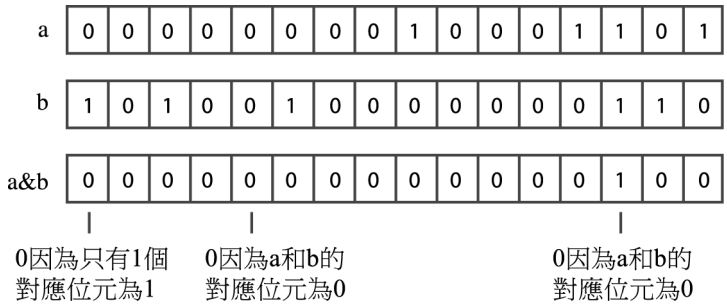

圖 E.6 位元 AND 運算子

表 E.3 總結 & 運算子的組合情形。

表 E.3 **b1 & b2** 之值

位元值	**b1 = 0**	**b1 = 1**
b2 = 0	0	0
b2 = 1	0	1

&= 運算子結合位元 AND 運算子和指定運算子：

```
a &= b;       // set a to a & b
```

另一種表示方式

C++ 提供數種位元運算子的另一種表示方式，如表 E.4 所示。提供字元集中沒有傳統位元運算子的區域。

表 E.4 位元運算子的表示方式

標準表示方式	另一種表示方式
&	bitand
&=	and_eq
|	bitor
|=	or_eq
~	compl
^	xor
^=	xor_eq

這些另一種表示方式的用法如下：

```
b = compl a bitand b;  // same as b = ~a & b;
c = a xor b;           // same as c = a ^ c;
```

一些常見的位元運算技巧

一般控制硬體時，需要將某些位元變成開或關，或者檢查位元狀態。位元運算子正好執行這些運算。我們很快的看過這些方法。

在下面的例子中，lottabits 代表一個平常的值，bit 代表對應特定位元的值。位元的標號方式是從右到左，起始位元為位元 0，所以位元 n 代表的值為 2^n。例如，只有

位元 3 為 1 的整數代表的值為 2^3，或為 8。一般說來，每個位元代表 2 的次方，如附錄 A 描述的二進位數字。所以用 bit 表 2 的次方，這對應特定位元為 1，而其它位元為 0。

打開位元

以下兩敘述將 lottabits 對應到 bit 值，代表的位元設為 1：

```
lottabits = lottabits | bit;
lottabits | = bit;
```

無論先前 lottabits 對應到 bit 的位元為何，最後此位元值均為 1。原因是 OR 運算子，無論 1 與 0 或 1 的組合，結果均為 1。而 lottabits 的其它位元值均未改變，原因是 0 與 0 作 OR 運算，結果為 0；0 與 1 作 OR 結果為 1。

切換位元

以下二敘述會切換 lottabits 對應到 bit 值代表的位元。也就是原為開者（on），現在變成關（off），原為關者，現在變成開：

```
lottabits = lottabits ^ bit;
lottabits ^ = bit;
```

1 與 0 作 XOR，結果為 1，打開關閉的位元；1 與 1 作 XOR，結果為 0，關閉打開的位元。lottabits 其它位元並未改變，原因是 0 與 0 作 XOR，結果為 0，0 與 1 作 XOR，結果為 1。

關閉位元

下面的敘述將 lottabits 對應到 bit 值，代表的位元設為 0：

```
lottabits = lottabits & ~bit;
```

無論先前 lottabits 該位元狀態為何，最後位元值均為 0。首先，~bit 將原先設為 0 的位元變成變成 1，原先只有一個設為 1 的位元變成 0。lottabits 的其它位元並未改變，因為任何值與 1 作 AND，結果與原來之值相同。

下面是執行相同功能更簡潔的作法：

```
lottabits &= ~bit;
```

測試位元值

假設您想知道 lottabits 對應到 bit 值代表的位元，其值為 1 或 0？下面的檢查方法不一定有用：

```
if (lottabits == bit)            //no good
```

原因是如果 lottabits 的對應位元為 1，則其它位元也可能為 1，所以只要 lottabits 值中該位元為 1，則以上敘述成立。修正方法是先將 lottabits 和 bit 作 AND，這產生的值是其他位元皆為 0，因為 0 和任何值作 AND 都是 0。只有對應於 bit 值的位元不會改變，因為 1 與任何值作 AND，就是該值。因此正確的測試如下：

```
if(lottabits & bit == bit)      //test a bit
```

真實世界的程式設計師會將此測試簡化成：

```
if(lottabits & bit)       //testing a bit
```

因為 bit 有一個位元設為 1，而其餘位元設為 0，則 lottabits & bit 不是 0（這測試為 false）就是 bit，這是非 0 值，測試為 true。

成員提領運算子

C++ 允許定義一個指標，指向類別成員，這些指標的宣告和提領都要使用特殊的表示方式。要瞭解涵蓋哪些，我們先看一個範例類別：

```
class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};
```

以 inches 資料成員為例。沒有特定的物件，inches 是一個標籤。也就是說，類別定義 inches 為成員識別字，但是在實際配置記憶體之前，您需要一個物件：

```
Example ob;  // now ob.inches exists
```

因此，您結合識別字 inches 和特定的物件，就可以指定實際的記憶體位置（在成員函數中，您可以省略物件的名稱，但是此物件會被理解為指標所指的物件）。

C++ 允許您定義成員指標指向識別字 inches：

```
int Example::*pt = &Example::inches;
```

這指標有一點不同於一般的指標。一般的指標指向特定的記憶體位置。但是 pt 指標不會指向特定的記憶體位置，因為此宣告沒有標示特定的物件。相反的，指

標 pt 標示在任何 Example 物件中 inches 成員的位置。就像識別字 inches 一樣，pt 的設計是要用來與物件識別字結合。基本上，運算式 *pt 會代表識別字 inches 的角色。因此，您可以用物件識別字標示要存取哪一個物件，以及 pt 指標標示該物件的 inches 成員。例如，類別成員函數可以用下面的程式碼：

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl;  // display inches member of ob1
cout << ob2.*pt << endl;  // display inches member of ob2
cout << po->*pt << endl;  // display inches member of *po
```

此處的 .* 和 ->* 都是成員提領運算子（member dereferencing operator）。當您有了特定的物件，如 ob1，則 ob1.*pt 表示 ob1 物件的 inches 成員。同樣的，pq->*pt 表示 pq 所指物件的 inches 成員。

改變上述範例的物件，就能改變使用哪一個 inches 成員。但是您也可以改變 pt 指標本身。因為 feet 的型態與 inches 相同，您可以將 pt 重設為指向 feet 成員，而不是 inches 成員，然後 ob1.*pt 會參考 ob1 的 feet 成員：

```
pt = &Example::feet;      // reset pt
cout << ob1.*pt << endl;  // display feet member of ob1
```

基本上，*pt 的組合會取代成員名稱，而且可以用來標示不同的成員名稱（相同的型態）。

您也可以使用成員指標標示成員函數。這語法有一點複雜。指向一般 void 型態，無引數之函數的指標其宣告敘述如下：

```
void (*pf)();  // pf points to a function
```

宣告指標指向成員函數，必須表示此函數屬於特定的類別。例如，下面是宣告指標指向 Example 類別的成員函數：

```
void (Example::*pf)() const;  // pf points to an Example member function
```

這表示可以使用 pf 之處與使用 Example 成員函數之處相同。注意 Example::*pf 必須在括號內。您可以將特定成員函數的位址設給此指標：

```
pr = &Example::show_inches;
```

注意，與一般函數指標指定不一樣的地方，是您必須使用位址運算子。作此指定後，您可以使用物件去呼叫此成員函數：

```
Example ob3(20);
(ob3.*pf)();     // invoke show_inches() using the ob3 object
```

您需要將整個 ob3.*pf 包在括號中，以清楚地標示此運算式表示函數名稱。

因為 show_feet() 與 show_inches() 有相同的原型，所以您也可以使用 pf 存取 show_feet() 成員函數：

```
pf = &Example::show_feet;
(ob3.*pf)();     // apply show_feet() to the ob3 object
```

這類別定義在範例程式 E.1 中，包含 use_ptr() 成員函數，它用成員指標存取 Example 類別的資料成員和函數成員。

範例程式 E.1 **memb_pt.cpp**

```
// memb_pt.cpp -- dereferencing pointers to class members
#include <iostream>
using namespace std;

class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};

Example::Example()
{
    feet = 0;
    inches = 0;
}

Example::Example(int ft)
{
    feet = ft;
    inches = 12 * feet;
}

Example::~Example()
{
}
```

```
void Example::show_in() const
{
    cout << inches << " inches\n";
}

void Example::show_ft() const
{
    cout << feet << " feet\n";
}

void Example::use_ptr() const
{
    Example yard(3);
    int Example::*pt;
    pt = &Example::inches;
    cout << "Set pt to &Example::inches:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    pt = &Example::feet;
    cout << "Set pt to &Example::feet:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    void (Example::*pf)() const;
    pf = &Example::show_in;
    cout << "Set pf to &Example::show_in:\n";
    cout << "Using (this->*pf)(): ";
    (this->*pf)();
    cout << "Using (yard.*pf)(): ";
    (yard.*pf)();
}

int main()
{
    Example car(15);
    Example van(20);
    Example garage;

    cout << "car.use_ptr() output:\n";
    car.use_ptr();
    cout << "\nvan.use_ptr() output:\n";
    van.use_ptr();

    return 0;
}
```

範例程式 E.1 的執行結果如下：

```
car.use_ptr() output:
Set pt to &Example::inches:
this->pt: 180
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 15
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 180 inches
Using (yard.*pf)(): 36 inches

van.use_ptr() output:
Set pt to &Example::inches:
this->pt: 240
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 20
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 240 inches
Using (yard.*pf)(): 36 inches
```

這範例在編譯期間指定指標值。在更複雜的類別中，您可以使用成員指標指向資料成員和成員函數，而確實的指標和成員連結會在執行期決定。

alignof(C++11)

電腦系統對資料儲存於記憶體有些限制。例如，有些系統可能會要求 double 值要儲存於偶數位置的記憶體，然而有些系統必需置於 8 倍數的記憶體位置。alignof 運算子以一資料型態為其參數，並傳回一整數，表示其對齊需求型態。對齊需求能決定資訊如何在結構安排的情形，如範例 E.2 所示。

範例程式 E.2 align.cpp

```
// align.cpp -- checking alignment
#include <iostream>
using namespace std;
struct things1
{
    char ch;
    int a;
    double x;
```

```
};
struct things2
{

    int a;
    double x;
char ch;
};

int main()
{
    things1 th1;
    things2 th2;
    cout << "char alignment: " << alignof(char) << endl;
    cout << "int alignment: " << alignof(int) << endl;
    cout << "double alignment: " << alignof(double) << endl;
    cout << "things1 alignment: " << alignof(things1) << endl;
    cout << "things2 alignment: " << alignof(things2) << endl;
    cout << "things1 size: " << sizeof(things1) << endl;
    cout << "things2 size: " << sizeof(things2) << endl;
    return 0;
}
```

以下是其輸出結果：

```
char alignment: 1
int alignment: 4
double alignment: 8
things1 alignment: 8
things2 alignment: 8
things1 size: 16
things2 size: 24
```

每一結構的對齊需求皆為 8。從輸出結果得知，結構的大小需為 8 的倍數，以便儲存。在範例程式 E.2 中，每一結構的大小為 13，但因對齊需求的關係，必須以 8 的倍數來填充。由於在結構中的 double 需以 8 的倍數加以對齊，所以在 thing2 中比 thing1 結構使用更多的內部填充。

noexcept(C++11)

關鍵字 noexcept 用來表明函數不會丟出例外。它也可以當作一運算子，用來決定它的運算元（或運算式）是否會丟出例外。假使運算元會丟出例外，則回傳 false，否則，回傳 true。例如，請考慮以下的宣告：

```
int hilt(int);
int halt(int) noexcept;
```

運算式 noexecept(hilt)應會是 false。因為 hilt()沒有保證例外不會被丟出。但 noexecept(halt)則會是 true。