Chapter 3

C++資料型別與運算子

基本資料型別

- 布林數：`bool`
- 字元　：`char`
- 整數　：`int`
- 浮點數：`float , double`

布林數 boolean

- 專門使用於邏輯運算
- 其值為 **true** 或 **false** 兩者之一

```
bool foo ;          // 定義布林變數 foo
foo = true ;        // foo 的值設定為真(true)
foo = false ;       // foo 的值設定為假(false)
```

- 如果要列印 **bool** 型別的資料值
  則真的布林值會印出 **1**、假的則印出 **0**
- 布林變數仍使用 **1** 個位元組

boolean

字元（一）

- C++使用一個位元組儲存字元資料
- 字元直接用單引號將字元框住，'M'、'3'、'?'
- 以跳離字元 (escape character) '\' 表示的特殊字元

名稱	ASCII名稱	C++ 字元	名稱	ASCII名稱	C++ 字元
換行字元	NL(LF)	\n	反斜線字元	\	\\
對齊字元	HT	\t	單引號	'	\'
重回行首字元	RT	\r	雙引號	"	\"
退格字元	BS	\b	八進位字元	000	\000
嗶聲字元	BEL	\a	十六進位字元	hhh	\xhhh

字元（二）

- 八進位數字字元：\XXX，X 為介於 0 到 7 的整數

\115 = 1 x 8^2 + 1 x 8^1 + 5 x 8^0 = 77

'\115' = ASCII 字元對照表的第77個字元 = 'M'

X 個數最多到 3 個

'\0' 代表 ASCII 字元表的第一個字元，即空字元 (null char)

- 十六進位數字字元：\xhh， h 個數最多兩個

十進位 ： 0 1 ... 9 10 11 12 13 14 15

十六進位： 0 1 ... 9 a b c d e f

\x4d = 4 x 16^1 + d x 16^0 = 77

'\x4d' = 'M'

null character

字串

- 由雙引號將若干字元框住

  "dog" 包含 'd' , 'o' , 'g' 與 '\0' 共 4 個字元

  '\0' 代表字串的終點

- "" 空字串也包含一個空字元 '\0'

- 每個中文字由兩個字元組成

  "山明水秀" 共包含 9 個字元

string

整數

- 無號整數：unsigned (int) [0,2^{32}–1]
- (有號)整數： int [-2^{31},2^{31}–1]
- 一般的整數都是以有號整數來儲存，若要指定數字為無號整數，需在數字後加上 'U' or 'u'

```
unsigned int foo ;
foo = 4294967295U ;
cout << foo << '\n' ;
```

integer

整數（二）

- 八進位整數　：數字之前加上一個 **0**

  $013 = 1 \times 8^1 + 3 \times 8^0 = 11$

- 十六進位整數：數字之前加上一個 **0x**

  $0x13 = 1 \times 16^1 + 3 \times 16^0 = 19$

```
int foo ;
foo = 222 + 0222 + 0x222 ;    // 0222  = 146
// 列印 904                   // 0x222 = 546
cout << foo << endl ;
```

浮點數

- 單精確度浮點數 (**float**) ：**4** 個位元組
- 雙精確度浮點數 (**double**) ：**8** 個位元組
- 數字包含小數點者都以雙精確度浮點數方式儲存

```
1. , .1 , 4.13 , 2.2e18 , -4.5E-27
```

float , double

常數

- 變數之值自初始設定後即不能被更動

```
const double PI = 3.14159 ;
const int STUDENT_NO = 50 ;
```

- 一般而言，常數變數通常以大寫字母表示
- 常數變數須設定初始值

```
const float foo ;
cin >> foo ;                        // 錯誤

double err ;
cin >> err ;
const double ERR = err ;            // 正確
```

constant

資料型別空間大小與 sizeof( )

- **sizeof(型別)**可以知道某型別所佔用的位元組空間大小

```
cout << sizeof(int) ;    // 整數所佔用的空間
```

資料型別	名稱	位元組數
bool	布林數	1
char	字元	1
short	短整數	2
int	整數	4
long	長整數	4
long long	長長整數	8
float	單精確度浮點數	4
double	雙精確度浮點數	8

sizeof

各項型別的最大與最小值

- 最大整數 int : 2147483647
- 最小整數 int : -2147483648

9 位有效數字

- 最大無號整數 unsigned int : 4294967295
- 最小無號整數 unsigned int : 0

9 位有效數字

- 最大長長整數 long long int : 9223372036854775807
- 最小長長整數 long long int : -9223372036854775808

18位有效數字

- 最大單精確度浮點數 float : 3.40282e+38
- 最小單精確度浮點數 float : 1.17549e-38

7 位有效數字

- 最大雙精確度浮點數 double : 1.79769e+308
- 最小雙精確度浮點數 double : 2.22507e-308

15位有效數字

資料型別轉換（一）

- 同型別變數間的數值運算，計算結果也是同型別
- 不同型別變數間的運算，小型別的資料會先以大型別的方式暫存。然後再以同型別方式計算，計算結果以大型別的方式儲存
- 型別大小

整數型別　：**bool → char → short → int → long**

浮點數型別：**float → double → long double**

- 不同型別的資料作運算時，會先將整數轉成浮點數

```
3 * 1.0 → 3.0 * 1.0 → 3.0
```

implicit conversion

資料型別轉換（二）

- 資料由小型別轉以大型別方式儲存，不會造成資料流失。但反之則會造成資料流失

```
double    pi = 3.141592654 ;
float     a  = pi ;                 //  a = 3.1415927
float     b  = 3.141592654 ;        //  b = 3.1415927
int       c  = pi ;                 //  c = 3
```

- 同型別資料間互轉有時也會造成問題

```
int           i = -1 ;
unsigned int  j = i  ;              // j = 4294567295
unsigned int  p = 3000000000 ;
int           q = p ;               // q = -1294967296
```

資料型別轉換（三）

- 強迫型別轉換：**static_cast<B>(a)** 或 **B(a)**

  將 **a** 變數資料型別強制轉成 **B** 型別

```
double foo ;
cin >> foo ;
cout << static_cast<int>(foo+0.5) << endl; // 四捨五入
cout << int(foo + 0.5) << endl ;           // C 語言方式

unsigned int a = 2 ;
int b = -3;
cout << "a + b = " << a + b << endl ; // 輸出 4294967295
cout << "b - a = " << b - a << endl ; // 輸出 4294967291
cout << "a + b = " << static_cast<int>(a+b); // 輸出 -1
cout << "b - a = " << static_cast<int>(b-a); // 輸出 -5
```

變數存在領域

- 變數存在領域：變數自定義起到消失前所存在的區塊
- 程式區塊為對等的大括號 {} 所含蓋的程式碼

局部變數 (local variable)　　變數所在區塊為局部範圍
全域變數 (global variable)　　變數定義於所有區塊之外

```
#include <iostream>
using namespace std ;
const double PI = 3.14159 ;                    // PI 為全域變數
int main() {
    int i = 5 ;                                // i 為局部變數
    {
      int j = 10 ;                             // j 為局部變數

      cout << 10 * PI + i * j << '\n' ;
    }
    cout << j << '\n' ;                        // 錯誤， j 已消失
    return 0 ;
}
```

scope

變數儲存型態

- 暫時變數 `temporary variable`

  變數離開其存在領域後隨即消失，其所擁有的記憶空間將回歸系統重新使用

  局部變數

- 永久變數 `permanent variable`

  變數所擁有的記憶空間會一直持續到程式結束才消失

  全域變數

C++基本運算子 (+ - * / %)

- 基本運算子

  +： 加　　**-**： 減

  *****： 乘　　**/**： 除　　**%**： 餘數

- 餘數運算子經常與隨機函式 **rand()** 一起使用

```
#include <iosteam>
#include <cstdlib>
using namespace std;
int main(){
    cout << (rand()%11+10) << endl ;
    return 0 ;
}
```

印出介於 [10,20] 的亂數

❖ rand() 的輸出是介於 [0,RAND_MAX] 之間的任一正整數

RAND_MAX 為預設常數，其值通常為 $2^{15} - 1$ 或是 $2^{31} - 1$

隨機函式的應用

- 兩種產生介於 [a,b] 之間的隨機亂數

```
// (1) 較差
a + rand() % (b-a+1)

// (2) 較好
a + static_cast<int>(1.*rand()*(b-a+1)/(RAND_MAX+1.))
```

- 為避免程式在每次執行的隨機數皆保持相同，通常需使用以下方式設定隨機函數的「起始值」

```
srand( static_cast<unsigned int>(time(NULL)) );
```

❖ rand() 與 srand() 在 cstdlib 標頭檔內
time() 在 ctime 標頭檔內

random

指定運算子（ +=，-=，...）

- `a += b` ⇔ `a = a + b`
  `a -= b` ⇔ `a = a - b`

- `a *= b` ⇔ `a = a * b`
  `a /= b` ⇔ `a = a / b`

- `a %= b` ⇔ `a = a % b`

- `a += b*c` ⇔ `a = a + ( b * c )`
- `a *= b-c` ⇔ `a = a * ( b - c )`

assignment operator

遞增、遞減運算子（++，--）

- ++foo 與 foo++ 結果皆為 foo = foo + 1
- --foo 與 foo-- 結果皆為 foo = foo – 1
- ++foo(--foo) 為 前置遞增(減)運算子
- foo++(foo--) 為 後置遞增(減)運算子

❖ 前置運算子先執行遞增(減)後運算
後置運算子先運算後執行遞增(減)

```
int a = 3 , b ;
b = ++a   →   ++a , b = a    // a = 4 , b = 4
b = a++   →   b = a , a++    // a = 4 , b = 3
```

increment/decrement operator

遞增、遞減運算子(二)

- 適當地使用遞增或遞減運算子可簡化程式碼

```
int a = 5 , b = 10 , c = 20 ;

c += --a + b++  ;  // --a , c += a + b , b++
```

→ a = 4 , b = 11 , c = 34

- 不要在一個式子中有太多前後置運算子交錯在一起

```
c += --a + ( b++ - a-- ) ;   // ???
```

正負運算子：+、-

- 正負運算子皆為單元運算子，分別代表數學式子的正負號

- 正負運算子與其後的變數間不可有空格存在

```
double a = 4. , b = 5. , c = 1 ;

// 印出 -4 5 -1
cout << -a << ' ' << +b << ' ' << -c
     << '\n' ;
```

左、右移位元運算子：<<，>>

- 左移位元運算子：將數字的位元資料整個左移若干位

```
11       = 00…001011
11 << 1  = 00…010110
11 << 2  = 00…101100
```

- 右移位元運算子：將數字的位元資料整個右移若干位

```
11       = 00…001011
11 >> 1  = 00…000101 = 5
11 >> 2  = 000…00010 = 2
```

- ❖ a <<= i ⇔ a = (a << i)
- ❖ a >>= i ⇔ a = (a >> i)

bitwise shift operator

位元比較運算子 (一)：~，&

■ ～：bitwise NOT 將數字的位元 0 , 1 對調

100 = 00…001100100 = 64 + 32 + 4

~100 = 11…110011011 = 4294967195

■ &：兩個同位置的位元皆為1才為 1，否則為 0

11 = 8 + 2 + 1 = 00…00001011

& 74 = 64 + 8 + 2 = 00…01001010

00…00001010

```
unsigned int i = 11 , j = 74 ;
cout << ( i & j ) << endl ;     // 印出 10
```

❖ a &= b ⇔ a = (a & b)

位元比較運算子（二）：|

- | ：bitwise or 兩同位置的位元有一個以上為 1 則為 1，否則為 0

```
  11 =  8 + 2 + 1 = 00…00001011
| 74 = 64 + 8 + 2 = 00…01001010
                    00…01001011  = 75
```

```
unsigned int i = 11 , j = 74 ;
cout << ( i | j ) << endl ;    // 印出 75
```

位元比較運算子（三）：^

■ ^：bitwise xor 同位置的位元相異為 1 ，否則為 0

```
   11 =  8 + 2 + 1 = 00…00001011
^  74 = 64 + 8 + 2 = 00…01001010
                     00…01000001 = 65
```

```
unsigned int i = 11 , j = 74 ;
cout << ( i ^ j ) << endl ;      // 印出 65
```

❖ a |= b ⇔ a = (a | b)
  a ^= b ⇔ a = (a ^ b)

單元與雙元運算子

- 單元運算子：只須要一個運算元

  ++ , -- , +(正) , -(負) ,

- 雙元運算子：須要兩個運算元

  +(加) , -(減) , * , / , % , = , +=

unary , binary operator

運算子執行優先順序

- `a = 2 * b + 4 ;`

  總共包含三個雙元運算式： `=` , `*` , `+`

  執行順序為 `*` , `+` , `=`

- `a = 2 * (b + 4) ;`

  執行順序為 `+` , `*` , `=`

operator precedence

運算子執行優先順序表(一)

運算等級	運算子
1	::
2	. -> [ ] ++(後置) --(後置) ()(函式呼叫) dynamic_cast static_cast reinterpret_cast const_cast
3	sizeof ++(前置) --(前置) ~ ! +(正) -(負) & *(參照) new delete
4	*(乘) / %
5	+(加) -(減)
6	>> <<
7	< <= > >=
8	== !=

運算子執行優先順序表（二）

運算等級	運算子
9	&
10	^
11	|
12	&&
13	||
14	= *= /= %= += -= <<= >>= &= |= ^=
15	? :
16	throw
17	,

左向與右向結合運算子（一）

- 左向結合運算子：包含 + - * / 等運算子

  同等級的運算子一起出現時，其執行順序皆是由左至右

```
int a = 4 + 5 + 6 ;
        ①
        ②
```

- 右向結合運算子：包含 = += -= *= /= 等運算子

```
int a = 1 , b = 10 , c = 20 ;
a = b = c = 1;
        ①
      ②
    ③
```

左向與右向結合運算子（二）

- 左右向結合運算子影響執行順序

```
int a = 1 , b = 10 , c = 20 ;
a += b += c ;  // a = 31 , b =  30 , c = 20
```

① ②

- 注意運算元變數要已定義才能使用

```
int a = b = 1 ;     //  錯誤 , b 未定義
```

① ②

列舉資料型別 (一)

- 列舉型別

```
enum Season {spring, summer, fall, winter};
Season foo ;
foo = winter ;
```

season 為一列舉資料型別，其值僅有 4 個，分別為

spring , summer , fall , winter

- 個別預設值為由 0 起算的整數

```
cout << foo << endl ;  // 輸出 3
```

enumerate type

列舉資料型別 (二)

- 個別數值也可以自行設定整數

```
enum Card { Jack = 11 , Queen = 12 ,
            King = 13 , Ace = 14 } ;
Card  foo , bar ;
foo = Queen     ;
bar = 12        ;          // 錯誤，不可設定成整數
cout << foo << '\n' ;  // 印出整數 12
```

- 列舉型別 enum 可間接地設定整數常數

```
enum { One = 1 , Two = 2 , ...} ;
cout << One << endl ; // 印出 1
```

結構資料型別（一）

- 結構資料型別：幾筆相關資料合併組成的新型別

```
struct  Date {
    unsigned int year     ;   // 年份
    unsigned int month    ;   // 月份
    unsigned int day      ;   // 日期
} ;
```

❖ 結構末尾的大括號要加上分號

- 使用句點運算子取用或設定結構內各筆資料

```
Date  a ;
a.year = 1912 ;  a.month = 1  ;  a.day = 1 ;
cout << a.year << "-" << a.month << endl ;
```

struct type

結構資料型別 (二)

■ 複數結構：

```
struct Complex {
    int  re , im ;
} ;
```

```
Complex  a , b , sum ;
a.re = 1 ; a.im = 2 ;     // a = 1 + 2 i
b.re = 3 ; b.im = 4 ;     // b = 3 + 4 i

sum.re = a.re + b.re ; sum.im = a.im + b.im ;
cout << "和 = " << sum.re << "+"
     << sum.im << " i" << endl ;
```

重定資料型別名稱

- **typedef** 資料型別可以使用 改變其名稱，增加程式可讀性

```
typedef  int  Index ;

typedef  double  Dollar ;

Index   i , j ;    // i , j 皆為整數

Dollar  x , y ;    // x , y 皆為 double
```

typedef