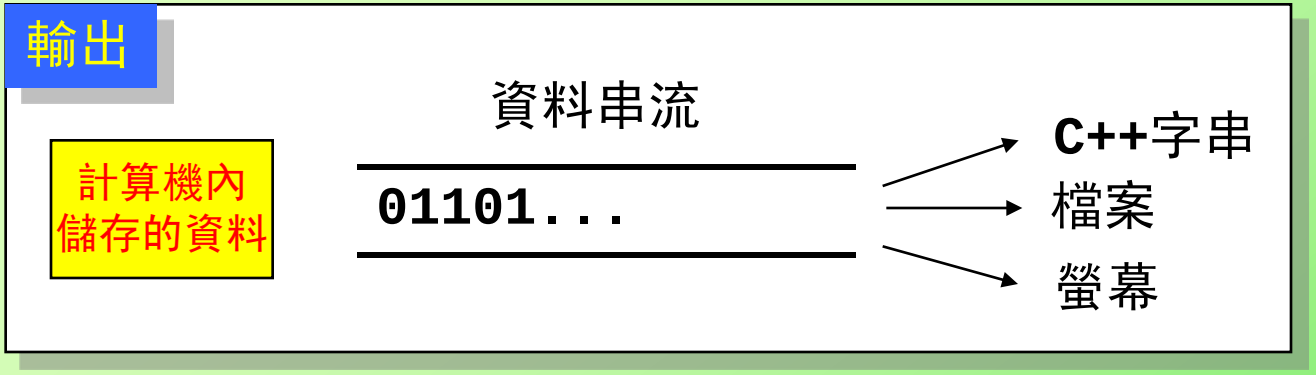
Chapter 11

輸出與輸入

輸出與輸入



- 資料串流：

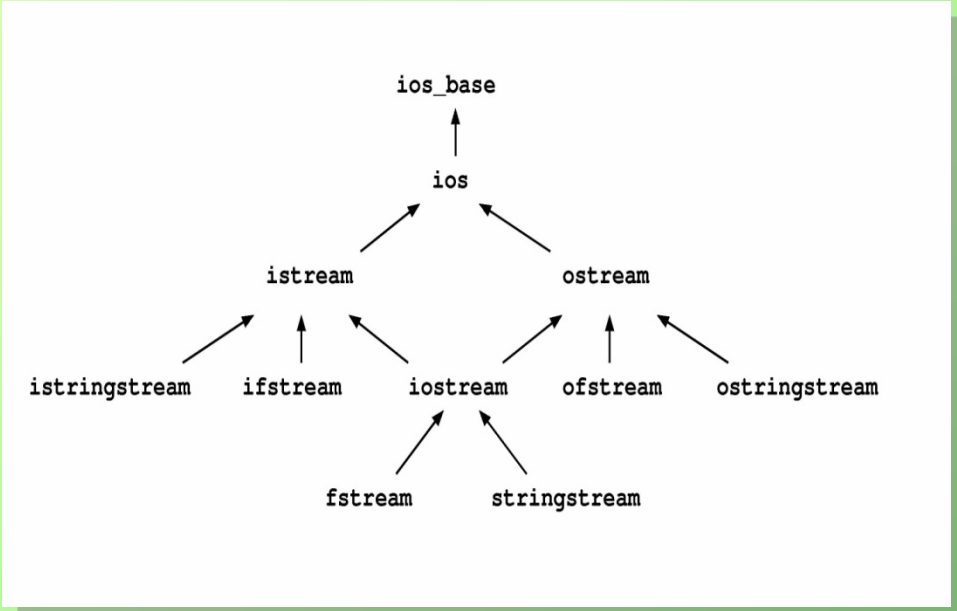
  為前後次序有別的輸入/輸出資料

- 資料串流物件：負責資料輸入/輸出的物件

data stream

輸入/輸出資料串流類別架構 (一)

■ 輸入/輸出主要類別架構



所有輸入/輸出的共同特性都被定義在上層，如 **ios_base , ios**

甲→乙：甲類別繼承自乙類別，甲類別為乙類別的衍生類別，甲類別的物件也擁有乙類別的特性，甲類別物件可以被當成乙類別物件看待

輸入/輸出資料串流類別架構 (二)

- **ostream** 類別：專門負責輸出，**cout** 為其物件

衍生類別	標頭檔	功能
ofstream	fstream	將輸出資料存入檔案內
ostringstream	sstream	將輸出資料存入C++字串內

- **istream** 類別：專門負責輸入，**cin** 為其物件

衍生類別	標頭檔	功能
ifstream	fstream	由檔案讀取資料到程式內使用
istringstream	sstream	由C++字串讀取資料到變數內使用

- **iostream** 類別：可以同時輸出與輸入

衍生類別	標頭檔	功能
fstream	fstream	可將資料存入檔案內或由之取出
stringstream	sstream	可將資料存入C++字串內或由之取出

輸入/輸出緩衝區（一）

- 緩衝區：用來暫時儲存欲輸入/輸出的字元資料

```
cout << 12 << ' ' << 345 << '\n' ;
```


```
int i , j ;
cin >> i >> j ;
```

buffer

輸入/輸出緩衝區（二）

- 所有輸入/輸出的資料串流物件各有其緩衝區
- 輸入/輸出緩衝區的資料來源或輸出目標依使用的資料串流類別有所差異

輸出

類別	來源	目標
ofstream	記憶空間	檔案
ostringstream	記憶空間	C++字串

輸入

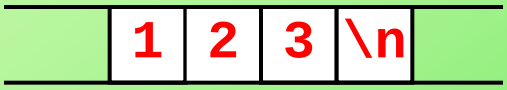
類別	來源	目標
ifstream	檔案	記憶空間
istringstream	C++字串	記憶空間

- 使用緩衝區的目的：增加程式處理輸入/輸出的效率

緩衝區的型別轉換（一）

- 資料在輸入/輸出過程中，通常都會經過型別轉換

```
int foo ;
cin >> foo ;
```

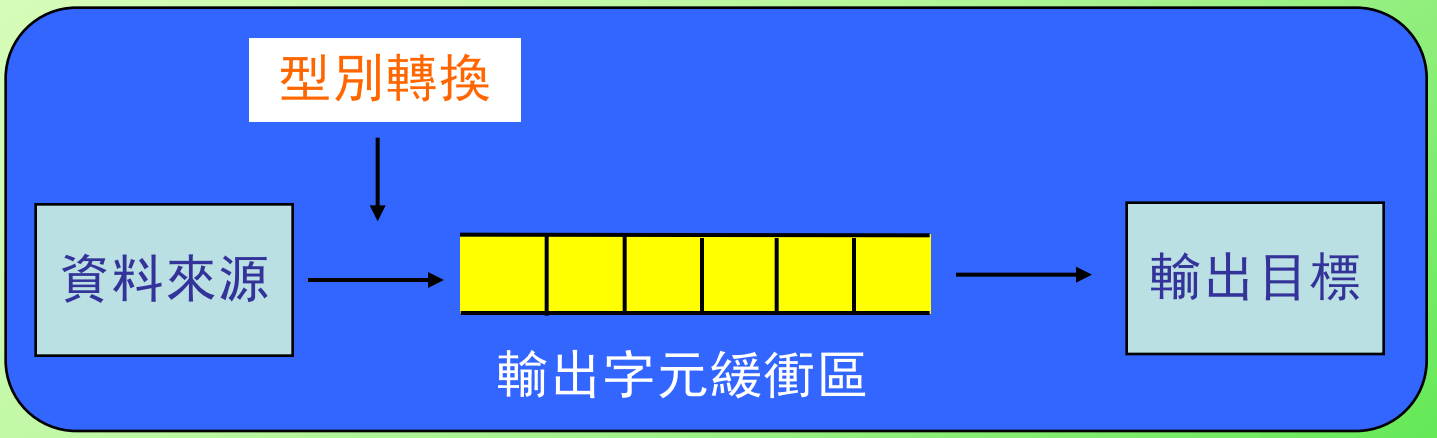


輸入緩衝區包含 4 個字元

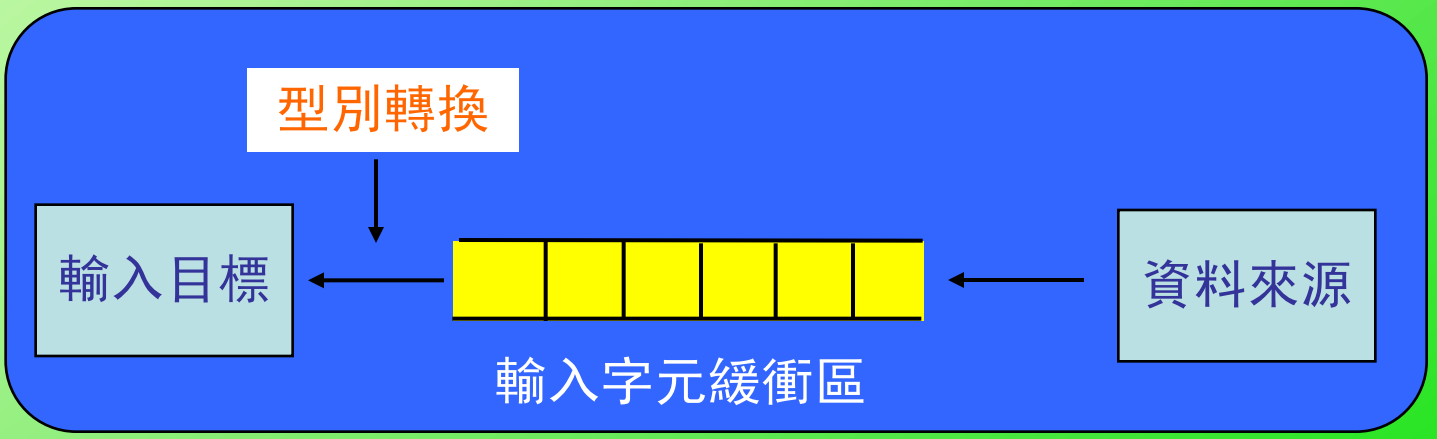
若輸入 123 後，按下 return 鍵，則輸入緩衝區內共有四個字元資料 '1'， '2' ， '3'， '\n' ，之後 C++ 會分析輸入緩衝區內字元資料，將之轉型為一整數 ， 存入 foo 變數所在的記憶空間，同時將 '1'、 '2'、 '3' 三個字元由輸入緩衝區內除去

緩衝區的型別轉換（二）

輸出



輸入



緩衝區的型別轉換（三）

- 使用者可在輸入/輸出過程中，選擇是否使用型別轉換

- 高階輸入/輸出 ： 有型別轉換的輸入/輸出
  低階輸入/輸出 ： 沒有型別轉換的輸入/輸出

- 型別轉換需要額外耗費執行時間，不利於一些需要快速輸入/輸出的應用，如影像與聲音讀取/儲存

輸入與輸出狀態旗幟（一）

- 輸入/輸出狀態旗幟：

  ios_base 類別內定義了四個輸入/輸出狀態旗幟

狀態旗幟	功用
badbit	資料串流發生問題
eofbit	已經讀到了檔案的末尾
failbit	讀取或存取的動作失敗
goodbit	讀取或存取的動作成功

使用時需加上 **ios_base**，即

```
ios_base::badbit     ios_base::eofbit
ios_base::failbit    ios_base::goodbit
```

state flag

輸入與輸出狀態旗幟（二）

- 當狀態旗幟為 **goodbit** 或 **eofbit** 時，代表程式已成功地執行之前的輸入動作
- 唯有在狀態旗幟為良好狀態時，即 **goodbit**，之後的輸入才可能成功。若是其他狀態旗幟，則之後的輸入動作皆視為無效

無效的輸入動作，不會更動變數已有的數值以及原有在輸入緩衝區內的字元資料

狀態旗幟成員函式(一)

- 狀態旗幟成員函式：定義在 **ios** 類別

成員函式	功用
`bool eof() const`	檢查是否已經讀到了檔案的末尾
`bool good() const`	檢查之前的讀取或存取動作是否成功
`bool fail() const`	檢查之前的讀取或存取動作是否失敗
`bool bad() const`	檢查是否資料串流產生問題
`void clear( iostate f = goodbit )`	重新設定狀態旗幟為 **f** ， 預設旗幟為 **goodbit**
`bool operator!() const`	與執行 **fail()** 相當

狀態旗幟成員函式（二）

- 讀取存在檔案 **data** 內的整數資料後印出

```
int      i ;
ifstream infile("data") ;
while( 1 ){                                  // 持續讀檔動作
   infile >> i ;                             // 由data檔讀取資料後，存入 i
   if ( infile.eof() ) break ;               // 如果讀到檔案末尾則跳離迴圈
   cout << i << endl ;                       // 列印 i
}
```

- 在條件式內的讀取動作會以讀取成功與否為條件式的真假值

```
int      i ;
ifstream infile("data") ;

// 重複由 data 檔讀取資料存入 i，直到讀取錯誤為止
while( infile >> i ) cout << i << endl ;
```

```
int a ;
while( cin >> a ) cout << a ;      // 列印在錯誤輸入之前的所有資料
```

狀態旗幟成員函式（三）

- 當輸入錯誤時，**failbit** 狀態旗幟會被設定，之後所有的輸入動作都被視為無效

```
int a , b ;
cin >> a  ;          // 輸入錯誤資料 "tiger"
cin >> b  ;          // 無效輸入動作
```

```
while( 1 ) {
    cin  >> a ;
    cout << a ;
}
```

左邊若輸入的過程造成錯誤，則程式會重複地列印最後一筆正確的 **a** 數值，不再接受任何輸入的資料

狀態旗幟成員函式(四)

- 程式可以透過清除動作 **clear()** 清除讀取錯誤狀態旗幟，清除後的狀態旗幟為良好 (goodbit)

```
double k ;
char s ;
while( ! ( cin >> k ) ){   // 持續讀取浮點數，直到正確為止
   cin.clear() ;           // 如果讀取錯誤，清除錯誤旗幟
   cin >> s ;              // 由緩衝區內讀取一個字元
}
cout << k << endl ;
```

❖ 以上是利用讀取字元方式去除輸入緩衝區內錯誤的型別資料

基本型別資料的輸出（一）

- C++ 會自動透過不同類型的輸出運算子來輸出基本資料型別。基本資料型別包含字元、傳統字串、整數、浮點數等等

```
char foo[] = "cat" ;
cout << foo << endl ;
```

❖ 傳統字串的列印是由起首字元起，列印到空字元為止， 即使在空字元之前的字元並不在字串內

基本型別資料的輸出（二）

- **ostream** 類別內定義兩個成員函式專門負責輸出字元與字元陣列

函式	主要用途
`ostream&  put(  char a  )`	輸出一個字元 **a**
`ostream&  write( char *p ,` `unsigned int n )`	輸出字元陣列的前 **n** 個字元 **p[0], ... , p[n-1]**

```
char foo[] = "Three cats." ;
int  i     = 0 ;
while( foo[i] != '\0' ) cout.put(foo[i++]) ;
cout << cout.write(foo,11) << endl ;      // 同上
```

基本型別資料的輸入

- 輸入運算子在讀取資料時，會跳過所有空白鍵字元
- 空白鍵字元包含 ' '、'\t'、\n'、'\f'、'\r'

```
int a , b ;
char foo[100] ;
cin >> a >> foo >> b ;   // 三筆資料以空白鍵字元分開
```

❖ 若 foo 字串內包含空白鍵字元，則須另外設定方式才能讀入正確字元資料

get 成員函式(一)

- **get** 成員函式：讀取字元與字串

函式	主要用途
`int&  get()`	讀取字元
`istream&  get( char& foo )`	讀取字元
`istream&  get( char* foo , streamsize n )`	讀取字串
`istream&  get( char* foo , streamsize n , char term )`	讀取字串

- 第一種 **get** 將取得的字元用整數方式回傳
  第二種 **get** 則是以字元參數方式回傳

- 使用 **get** 成員函式讀取字串時，函式最多讀取 **n-1** 個字元，會自動保留一個空字元當成字串的終點

get 成員函式(二)

- 使用 **get** 讀取字串時，所取得的字元數最多到終止字元 (term) 為止。當字元資料由輸入緩衝區移到字串時，終止字元仍會留在輸入緩衝區內，預設的終止字元為 **'\n'**
- 最後一個 **get** 成員函式可以自行設定終止字元
- 終止字元須明確取出，否則無法繼續使用 **get** 讀取字串

```
char line[256] ;
while( cin ){
   cin.get( line , 256 ) ;    // 讀取一行，且最多讀取 255 個字元
   cout << line << '\n'  ;
   cin.get()             ;    // 取出終止字元
}
```

getline 成員函式(一)

- **getline** 成員函式：
  讀取整行，存入傳統陣列，函式會自動除去存在輸入緩衝區內的終止字元

函式	主要用途
`istream&  getline( char* foo , streamsize n )`	讀取一行字串
`istream&  getline( char* foo , streamsize n , char term )`	自設終止字元讀取一行字串

```
char line[5] ;
while( cin ){
   cin.getline( line , 5 ) ;  // 讀取一行，但最多讀取 4 個字元
   cout << line << '\n' ;
}
```

getline 成員函式(二)

- 若所讀入行的字元數過多，則在讀入 **n-1** 個字元後會補上空字元，同時設定讀取錯誤狀態旗幟。因此若要繼續進行讀取動作，須先清除錯誤旗幟，才能接續之後的讀取

```
char line[5] ;
while( cin ){
   cin.getline( line , 5 ) ;
   cout << line << '\n' ;
   if( cin.fail() ) cin.clear() ;
}
```

getline 全域函式

- **getline** 全域函式：

  讀入整行資料後存入 **C++** 字串參數

函式	用途
`istream&  getline( istream& , string& )`	讀取一行字串
`istream&  getline( istream& , string& , char term )`	自設終止字元讀取一行字串

```
string   line ;
getline( cin , line ) ;   // 讀入一行資料於line字串內
```

❖ 此種整行讀入方式不需考慮行的字元總數，因此沒有字串長度不足的問題產生

read 成員函式

- **read** 成員函式：將讀入的字元存入字元陣列

函式	主要用途
`istream&  read( char* foo ,` `streamsize n )`	讀取資料存入字元陣列

```
char  word[5] ;
while( cin ){
   cin.read( word , 4 ) ;
   word[4] = '\0' ;
   cout << word << '\n' ;
}
```

- read 成員函式會讀入 n 個字元到字元陣列內，且末尾不會自動補上空字元

其它輸入成員函式

- **peek()** ： 觀察在輸入緩衝區內的第一個字元
- **ignore(n)** ： 去除輸入緩衝區內的前 **n** 個字元
- **gcount()** ： 回傳之前使用 **get , getline , read** 所讀入的字元個數

```
int    id ;
string name ;
cout << "> 輸入學號或姓名 :" ;

if( cin.peek() >= '0' && cin.peek() <= '9' )
   cin >> id ;
else
   cin >> name ;
```

格式化輸出

- 格式化輸出：
  資料輸出時，可以利用指定的列印型式輸出

- 輸出格式旗幟：
  定義在 **ios_base** 類別內的常整數，用來控制資料輸出的格式

❖ 若要將資料輸入程式中使用，只要將資料以空白鍵字元分開即可讀入，不須以格式方式輸入，以避免麻煩

formatted output

輸出格式旗幟

- 群組型旗幟

```
ios_base::adjustfield : 控制輸出資料的對齊方式
ios_base::basefield   : 控制整數輸出的進位方式
ios_base::floatfield  : 控制浮點數輸出的方式
```

- 獨立型旗幟

```
ios_base::showpoint   : 是否浮點數顯示末尾的零
ios_base::showpos     : 是否顯示正數的正號
ios_base::showbase    : 顯示數字的進位符號
ios_base::skipws      : 忽略輸入的空白鍵字元
...
```

群組型格式旗幟

- 三種群組型格式旗幟

旗幟參數	旗幟群組	作用
ios_base::left ios_base::right ios_base::internal	ios_base:: adjustfield	輸出的資料靠左邊對齊 輸出的資料靠右邊對齊(預設值) 輸出的資料正負號靠左對齊，數字靠右對齊
ios_base::dec ios_base::hex ios_base::oct	ios_base:: basefield	數據以十進位方式輸出(預設值) 數據以十六進位方式輸出 數據以八進位方式輸出
ios_base::fixed ios_base::scientific	ios_base:: floatfield	數據用小數點方式輸出， dd.dddd 數據以科學記號輸出， d.dddddddEdd

❖ 群組型的格式旗幟之個別旗幟功能互斥，僅能由中選擇其一設定

獨立型格式旗幟

- 五種獨立型格式旗幟

旗幟參數	作用
ios_base::showpoint	顯示末尾的零
ios_base::showbase	顯示數字的進位符號 八進位數字，其數據之前加 **0** 十六進位數字，其數據之前加 **0x**
ios_base::showpos	如果數據為正數，顯示 **+** 號
ios_base::uppercase	用大寫字母輸出的 **E** (科學記號) 與十六進位數字等
ios_base::skipws	忽略輸入的空白鍵(預設值)

獨立型格式旗幟設定

- 獨立型的格式旗幟

```
setf(flag)   : 設定 flag 旗幟
unsetf(flag) : 清除 flag 旗幟
```

```
int  i = 15 ;
cout.setf(ios_base::showpos) ;      // 顯示正號
cout << i << endl ;                 // 輸出：+15

cout.unsetf(ios_base::showpos) ;    // 不顯示正號
```

群組型格式旗幟設定

- 群組型格式旗幟

setf(flag,field)：設定或修改 **field** 群組旗幟內的 **flag** 旗幟

```
int  i = 15 ;
cout.setf( ios_base::oct ,                  // 以八進位方式輸出
           ios_base::basefield ) ;
cout.setf( ios_base::showbase )  ;  // 八進位數字之前加0
cout << i << endl ;                         // 輸出：017
```

旗幟設定注意事項

- 當旗幟設定後，其作用會持續，除非重設或者清除
- 改變群組旗幟內的旗幟只須使用第二種類型的**setf**方式來重設旗幟即可，不須使用 **unsetf** 來清除旗幟 。 但若要將非群組旗幟內的旗幟作用消除，則須使用 **unsetf** 來清除旗幟
- 群組旗幟內的旗幟不可以使用獨立型的旗幟設定或是清除方式來更改群組旗幟
- 由於旗幟設定的作用會持續，為避免影響以後的輸出格式，最好在資料輸出後隨即清除旗幟

整數的輸出（一）

- 整數輸出相關的旗幟

```
ios_base::basefield : 輸出整數的進位方式
ios_base::showpos   : 正數是否呈現正號
ios_base::showbase  : 顯示進位的代表符號
ios_base::uppercase : 進位符號是否以大寫方式呈現
```

整數的輸出（二）

- 整數輸出範例

```
cout.setf(ios_base::showbase) ;          // 顯示進位基底
cout.setf(ios_base::uppercase) ;         // 使用大寫字母
cout.setf(ios_base::hex,                 // 十六進位輸出
          ios_base::basefiled) ;
cout << "> " << 1234 << ' ' ;            // 輸出： > 0X4D2

cout.setf(ios_base::dec,                 // 以十進位輸出
          ios_base::basefiled) ;
cout << 1234 << endl ;                   // 輸出：1234
cout.unsetf(ios_base::showbase) ;        // 清除進位基底顯示
cout.unsetf(ios_base::uppercase) ;       // 清除大寫字母旗幟
```

八進位或十六進位整數的讀取

- **ios_base::basefield** 也可用在八進位或十六進位整數資料的讀取

```
int  no ;

// 輸入的數字以十六進位方式讀入
cin.setf(ios_base::hex,ios_base::basefield) ;
cin >> no ;

// 回復以十進位方式讀入資料
cin.setf(ios_base::dec,ios_base::basefield) ;
```

以上若輸入 **13** 則會被解讀為數字 **19**，即 $1x16^1+3x16^0$

❖ 此時輸入的數字之前不須加入 **0x** 字樣

浮點數輸出（一）

- 浮點數輸出由輸出格式與精度所控制

浮點數輸出格式	使用方式	作用
一般輸出形式 (general format)	setf(0, ios_base::floatfield)	dd.dddd 或 d.dddddEdd 精度 精度
小數點形式 (fixed format)	setf(ios_base::fixed, ios_base::floatfield)	dd.dddddd 精度
科學記號形式 (scientific format)	setf(ios_base::scientific, ios_base::floatfield)	d.ddddddEdd 精度

❖ 浮點數預設的精度數量為 6 位

precision

浮點數輸出（二）

- 一般輸出格式為浮點數輸出的預設格式，輸出形式可有兩種，但以能用最小的空間來顯示較多的輸出數字為選用的標準

```
double a = 1234567.8 , b = 1234.5678 ;
// 其輸出為 a = 1.234567e+06 , b = 1234.57
cout << "> " << a << ' ' << b << endl ;
```

- 浮點數預設的精度為 6 位，末尾數為四捨五入後的數字，精度可用 precision(n) 成員函式修改，設定後作用持續

```
double a = 1234.5678 ;
int    b = 12345678 ;
cout.setf(ios_base::fixed,ios_base::floatfield) ;
cout.precision(2) ;
// 其輸出為 a = 1234.57 , b = 12345678
cout << "> " << a << ' ' << b << endl ;
```

general format

輸出寬度與填滿字元（一）

- **width(n)** ： 以 **n** 格列印下一筆資料
- **fill(c)** ： 若輸出寬度較資料寬度大，以 **c** 字元填滿

➢ 輸出 **[000] [001] ... [005]**

```
cout.fill('0') ;
cout.setf(ios_base::right,ios_base::adjustfield) ;
for( int i = 0 ; i < 6 ; ++i ) {
   cout << '[' ;
   cout.width(3) ;
   cout << i << "] " ;
}
```

➢ 輸出 **100** 個星號

```
cout.fill('*')  ;
cout.width(100) ;
cout << "*"     ;     // 或者 cout << '*'
```

輸出寬度與填滿字元（二）

- **width** 成員函式僅影響下一筆輸出資料

```
int i = 123    ;
cout.fill('*') ;
cout.width(5)  ;
cout << i << ' '
     << i << endl ;
```

```
// 設定取代字元
// 設定五格寬度列印
// 輸出兩個 i，結果為
// **123 123
```

- 如果所欲輸出的資料內容較設定的列印寬度大時，則 C++ 將不理會寬度的限制

- 預設的寬度：儘可能列印最多的資料

```
cout.width(0) ;
```

忽略輸入空白鍵字元

- **ios_base::skipws**

  忽略所輸入的空白鍵字元，即輸入資料不包含空白鍵字元

```
char a[100] ;
int  i = 0  ;

// 除去忽略空白鍵設定，即所有字元皆是資料
cin.unsetf(ios_base::skipws) ;
do{
   cin >> a[i] ;                          // 讀取任何字元
} while( a[i++] != '\n' ) ;               // 直到讀到換行字元

// 回復忽略空白鍵，即空白鍵不算輸入資料
cin.setf(ios_base::skipws) ;
a[i-1] = '\0' ;                           // 註明字串結尾
cout << a << endl ;                       // 列印字串
```

輸入/輸出格式處理器（一）

- 輸入/輸出格式處理器

```
cout << hex << setfill('*') << setw(5) << 255 << endl ;
```

相當於

```
cout.setf(ios_base::hex,ios_base::basefield);
cout.fill('*')       ;
cout.width(5)        ;
cout << 255 << endl ;
```

❖ 若是使用到的格式處理器需要參數時，則要加入 **iomanip** 標頭檔

input/output manipulator

輸入/輸出格式處理器（二）

格式處理器	作用
boolalpha / noboolalpha	使用/不使用英文字 **true/false** 來代替真假
showbase / noshowbase	顯示/不顯示進位符號
showpoint / noshowpoint	顯示/不顯示小數點
showpos / noshowpos	正數顯示/不顯示正號
skipws / noskipws	忽略/不忽略空白鍵
left / right / internal	靠左/右/中對齊
dec / hex / oct	十進位/十六進位/八進位顯示
fixed / scientific	固定式小數點/科學記號顯示浮點數
flush	清除輸出緩衝區
endl	輸出換行字元後再清除輸出緩衝區
ends	輸出空字元**('\0')**後再清除輸出緩衝區
resetiosflags(flag)	清除旗幟參數，相當於 **unsetf(flag)**
setiosflags(flag)	設定旗幟參數，相當於一個參數的 **setf(flag)**
setw(int)	設定輸出寬度
setfill(char)	設定背景字元
setprecision(int)	設定浮點數輸出精度

輸入/輸出格式處理器（三）

- 範例

```
cout << showpos << left << setw(10)
                << setfill('*') << 10 << endl ;
```

輸出為

```
+10*******
```

或者為

```
cout << setiosflags(ios_base::showpos)
     << setiosflags(ios_base::left)
     << setw(10) << setfill('*')
     << 10 << resetiosflags(ios_base::left) << endl ;
```

清空輸出緩衝區（一）

- 當資料輸出時，會先存入輸出緩衝區，等到相當的資料後才會一起送出輸出的資料到目標處
- 由於緩衝區的緣故，輸出的資料並不會立即顯示於螢幕，但使用者可使用 **flush** 清空輸出緩衝區強制顯示

```
cout << "> Come to see me right away\n" << flush ;
```

或者

```
cout << "> Come to see me right away\n" ;
cout.flush() ;
```

清空輸出緩衝區（二）

- 當程式執行輸入敘述時也會先清空輸出緩衝區

```
int no ;
cout << "> 請輸入數字 :" ;
cin >> no ;
```

- **endl** 相當於換行後清空輸出緩衝區

```
cout << endl ;

cout << '\n' << flush ;     // 同上
```

自定輸出格式處理器(一)

- 使用者可以自定輸出/輸出格式處理器

```
char no = 67 ;
cout << no << endl ;                              // 輸出 C 字元
cout << static_cast<int>(no) << endl ; // 輸出 67
```

可設計輸出格式處理器使得之後的字元輸出其所對應的整數

```
cout << chint << no << endl ;                     // 輸出 67
```

- 使用者自定輸出格式處理器

```
struct  Chint {
   ostream  *optr ;   // 指標 optr 指向 ostream 物件
};
Chint    chint ;  // chint 格式處理器為 Chint 的物件
```

自定輸出格式處理器(二)

- 定義兩個輸出運算子

```
cout << chint << no << endl ;
```

```
Chint& operator<< ( ostream& out , Chint& foo ) {
   foo.optr = &out ;
   return foo ;
}
```

```
ostream& operator<< ( Chint& foo , char c ) {
   return *(foo.optr) << static_cast<int>(c) ;
}
```

❖ 要處理的資料須緊接在自定的格式處理器之後

六角形數字盤

- 利用 **setw** 與 **setfill** 來列印連續的相同字元

```
cout << setfill('*') << setw(5) << "*" ;


// 同上
for( int i = 0 ; i < 5 ; ++i ) cout << "*" ;
```

❖ 也可使用 **string(5,'*') ;**

程式

輸出

字串金字塔

- 利用關閉 **ios::skipws** 來讀取內含空白鍵字元的字串

```
cin.setf(ios::skipws) ;      // 跳開空白鍵字元

cin.unsetf(ios::skipws) ;    // 讀取空白鍵字元
```

程式

輸出

檔案資料串流類別

■ 檔案資料串流類別

類別	功用
ifstream	檔案輸入
ofstream	檔案輸出
fstream	檔案輸入與輸出

```
#include <fstream>
```

```
ifstream infile  ; // 產生一個負責由檔案讀取資料的物件 infile
ofstream outfile ; // 產生一個負責輸出資料到檔案的物件 outfile
fstream  iofile  ; // 產生一個可以同時輸入與輸出資料的物件 iofile
```

❖ 之前所有的狀態與格式旗幟都適用於檔案資料串流物件

```
outfile.setf(ios_base::hex,ios_base::basefield) ;
outfile << setw(10) << 10 << endl ;
```

開啟檔案成員函式（一）

- **open("filename",mode)：**
  以 mode 模式開啟檔案 filename

檔案開啟設定參數	功用
ios_base::in	開啟檔案準備輸入
ios_base::out	開啟檔案準備輸出
ios_base::app	將準備輸出的資料接在檔案末尾
ios_base::ate	開啟檔案且移到檔案末尾
ios_base::binary	以二進位方式輸入/輸出檔案
ios_base::trunc	將檔案內容去除成空長度檔案
ios_base::nocreate	如果檔案已經不存在，則開啟檔案動作失敗
ios_base::noreplace	如果檔案已經存在，則開啟檔案動作失敗

開啟檔案成員函式（二）

- 開啟新檔案準備輸出

```
ofstream outfile1 ;
outfile1.open(”datafile”,ios_base::out) ;
```

- 開啟檔案並將要輸出的資料接續在檔案末尾

```
ofstream outfile2 ;
outfile2.open(”datafile”,ios_base::out | ios_base::app) ;
```

- 開啟檔案準備輸入資料

```
ifstream infile(”datafile”) ;   // 開啟datafile準備輸入
```

或者是

```
ifstream infile ;
infile.open(”datafile”,ios_base::in) ;
```

開啟檔案成員函式（三）

- 可使用 **operator!** 來判斷開啟動作是否成功

```
ofstream outfile( ”datafile” , ios_base::out
                             | ios_base::trunc ) ;

if( ! outfile ){
   cout << ”> datafile 檔案開啟錯誤\n” ;
   ...
}
```

關閉檔案

- close()

```
ifstream infile(”animal”) ;
...
infile.close() ;
```

❖ 檔案關閉後，串流物件仍然存在，可以再利用其開啟其他檔案

檔案資料的存取

- 仍可使用 operator<< 與 operator>> 來處理資料的輸出與輸入

```
int i , j ;

// 將 0 到 9 整數存入 data 檔案內
ofstream outfile("data") ;
for( i = 0 ; i < 10 ; ++i ) outfile << i << '\n' ;
outfile.close() ;

// 將 data 檔案內的資料讀入程式中
ifstream infile("data") ;
for( i = 0 ; i < 10 ; ++i ){
   infile >> j ;
   cout << j << '\n';
}
infile.close() ;
```

檔案的類型

- 文字檔

  檔案僅包含為人所能辨識閱讀的 **ASCII** 交換碼字元

  例如：英文**C++**程式碼檔案、英文文字信件

- 二進位檔

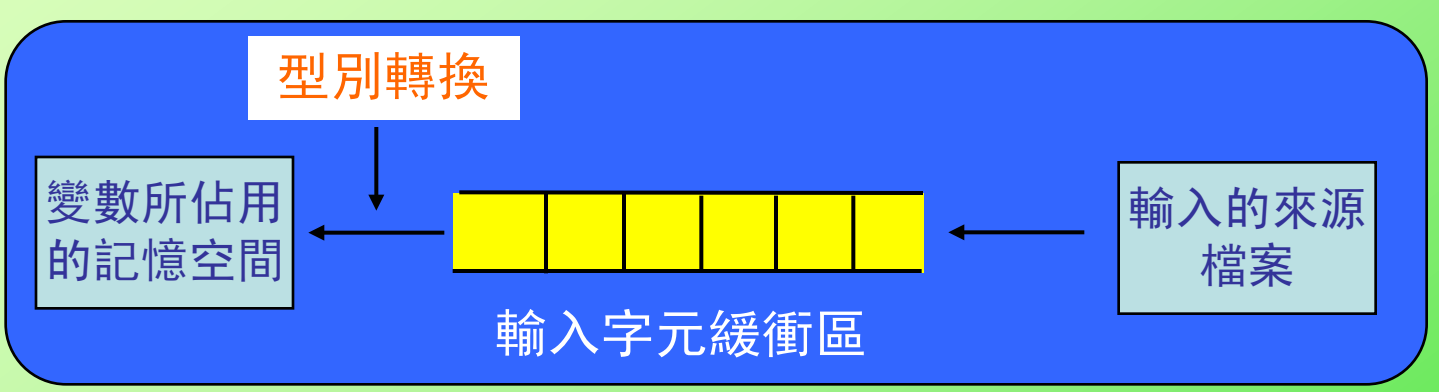
  檔案內容包含佔用整個 **8** 位元的字元

  例如：影像檔、語音檔、中文檔

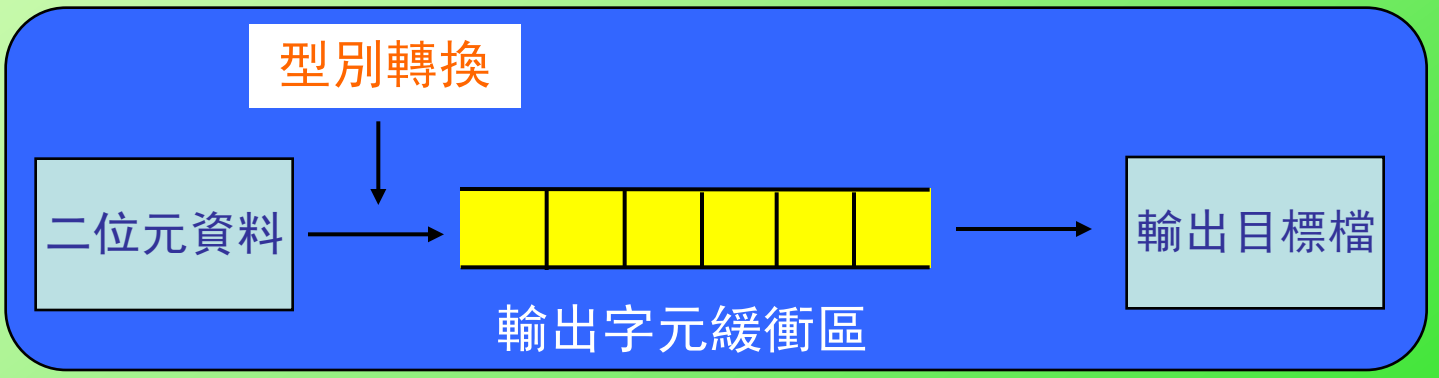
text file, binary file

文字檔與二位元檔的產生

輸入



輸出



- 若資料輸出過程無型別轉換，則輸出的資料檔為二位元檔；反之為文字檔
- 程式在讀入文字檔時須要經過型別轉換，才能轉以二位元方式存入程式佔用的記憶空間等待使用

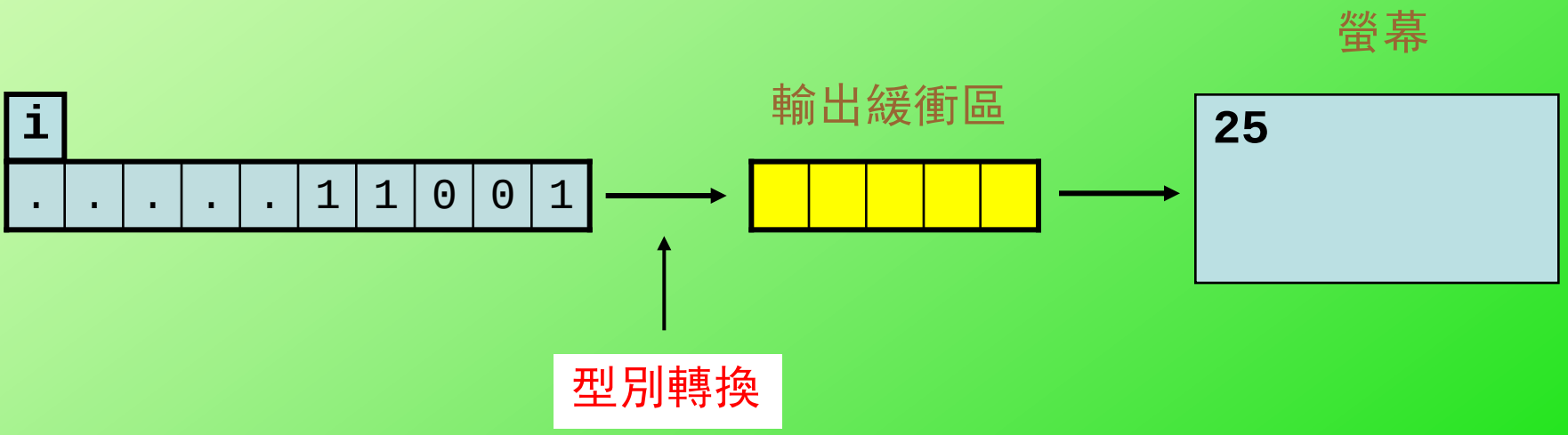
輸出資料的轉型

- 輸出資料的轉型：

  二進位數字 → 字元

```
int  i = 25 ;          // i 以四個位元組儲存資料 000...011001

cout << i << '\n' ; // 經過型別轉換後在螢幕顯示 2 個字元
                       // '2'與'5'
```



文字檔與二位元檔的優缺點

- 檔案類型的比較：

比較項目	文字檔	二位元檔
資料可讀性	容易	困難
輸入/輸出效率	差	佳
檔案空間大小	通常較大	通常較小
檔案可攜帶性	通行	差
浮點數存取誤差	有	無

二位元檔的輸出

■ 輸出二位元檔

```
// 開啟 data 檔以二位元方式寫入
ofstream foo( "data" , ios_base::out | ios_base::binary ) ;

// 將 0 到 9 共 10 個整數，以每次一個整數大小直接寫入 data 檔案內
for ( int i = 0 ; i < 10 ; ++i )
   foo.write( reinterpret_cast<char*>(&i) , sizeof(i) ) ;
foo.close() ;
```

- ios_base::binary : 代表不使用型別轉換直接複製記憶空間的資料
- ostream& ostream::write(char* p , unsigned int n)
  輸出 p 所指向的字元陣列的前 n 個字元
- reinterpret_cast<A>(B) : 將 B 型別「當成」 A 型別使用，
  A、B 皆須為位址

二位元檔的輸入

■ 讀取二位元檔

```
// 開啟 data 檔以二位元方式讀取
ifstream foo( "data" , ios_base::in | ios_base::binary ) ;

int i ;

// 持續讀取整數資料直到錯誤為止
while ( foo.read( reinterpret_cast<char*>(&i) ,
        sizeof(i) ) ) {
   cout << i << '\n' ;
}
```

❖ **istream& istream::read(char* p , unsigned int n)**

讀取 **n** 個字元到 **p** 所指向的字元陣列內

二位元檔的輸出與輸入

- 將浮點數 0.1 到 0.5 等 5 個數存入二位元檔後，再讀入

```
const int S = 5 ;

// 檔案輸出
ofstream out( "datafile" , ios_base::out | ios_base::binary ) ;
double   data1[S] = { 0.1 , 0.2 , 0.3 , 0.4 , 0.5 } ;
out.write( reinterpret_cast<char*>(data1) , S*sizeof(double) ) ;
out.close() ;


// 檔案輸入
ifstream in( "datafile" , ios_base::in | ios_base::binary ) ;
double   data2[S] ;
in.read( reinterpret_cast<char*>(data2) , S*sizeof(double) ) ;
in.close() ;
for( int i = 0 ; i < S ; ++i ) cout << data2[i] << '\n' ;
```

uuencode 編碼程式（一）

- 將二進位的資料轉成可閱讀的文字儲存

編碼策略

讀入 3 個位元組資料於整數內，然後將其切割成以 6 個位元為一單位的四等份，將每個等份的數值對應到 ASCII 碼中的可閱讀區

```
01011010 01101001 11100011 => 010110 100110 100111 100011
                                  a      b      c      d
```

uuencode 編碼程式（二）

- ASCII碼的對應區字元：64個

```
32:   33: ! 34: ” 35: # 36: $ 37: % 38: & 39: ’
40: ( 41: ) 42: * 43: + 44: , 45: - 46: . 47: /
48: 0 49: 1 50: 2 51: 3 52: 4 53: 5 54: 6 55: 7
56: 8 57: 9 58: : 59: ; 60: < 61: = 62: > 63: ?
64: @ 65: A 66: B 67: C 68: D 69: E 70: F 71: G
72: H 73: I 74: J 75: K 76: L 77: M 78: N 79: O
80: P 81: Q 82: R 83: S 84: T 85: U 86: V 87: W
88: X 89: Y 90: Z 91: [ 92: \ 93: ] 94: ^ 95: _
```

uuencode 編碼程式（三）


```
[a]  010110 = 22 ----> char(32+22) = char(54) = '6'

[b]  100110 = 38 ----> char(32+38) = char(70) = 'F'

[c]  100111 = 39 ----> char(32+39) = char(71) = 'G'

[d]  100011 = 35 ----> char(32+35) = char(67) = 'C'
```

- 3 個位元組的資料編碼後改以 4 個位元組儲存 ， 因此編碼後的資料檔約膨脹了 1/3 倍

uuencode 編碼程式（四）

- CPU 讀取資料方式
  - 高位元組優先：讀入的位元資料會由高位元組依次往低位元組儲存

    Motorola 68K CPU

  - 低位元組優先：讀入的位元資料會由低位元組依次往高位元組儲存

    Intel X86 CPU

uuencode 編碼程式（五）

- 高位元組優先 CPU

	high byte 高位元組			low byte 低位元組
位元組依次	44444444	33333333	22222222	11111111
位元依次	33322222	22222111	11111110	00000000
	21098765	43210987	65432109	87654321
a	10101101	10100100	11101011	00000000
a >>= 8	00000000	10101101	10100100	11101011

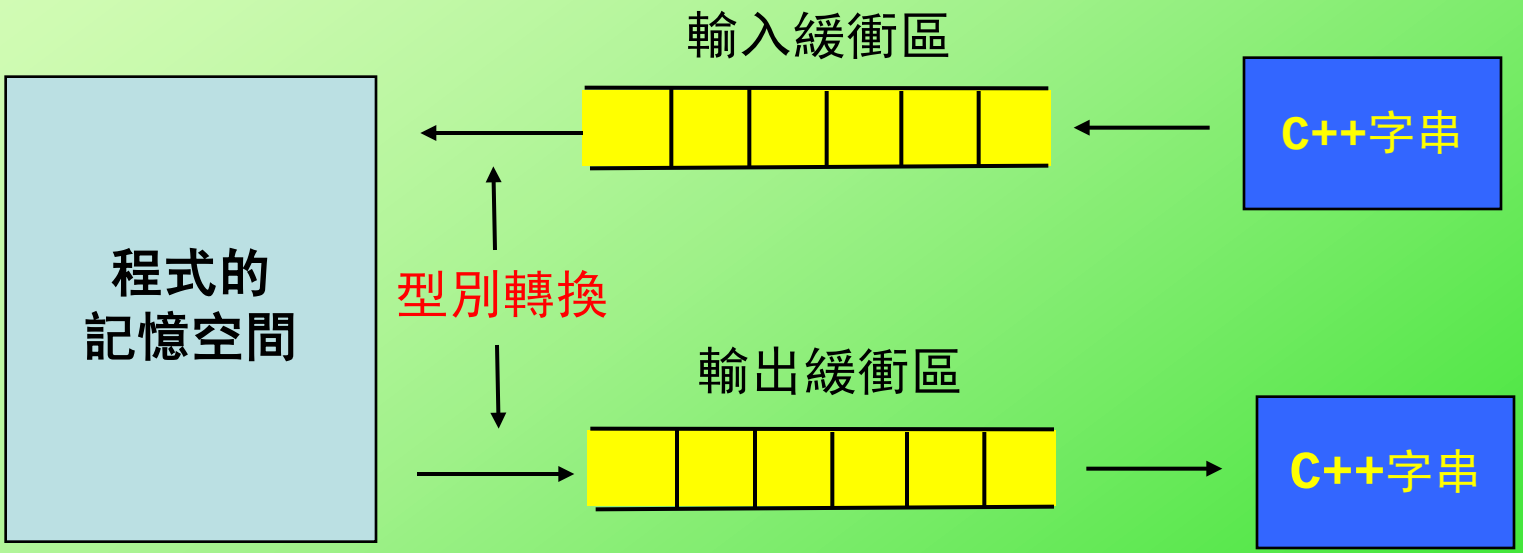
切割　　字元

```
1                 101011                    -> 43+32 -> 75 = K
2                       01 1010             -> 26+32 -> 58 = :
3                            0100 11        -> 19+32 -> 41 = )
4                                 101011    -> 27+32 -> 59 = ;
```

❖ 逆向的編碼程式稱為解碼 (decode)

C++字串資料串流的輸入/輸出

- 輸入型 C++ 字串資料串流類別：istringstream
- 輸出型 C++ 字串資料串流類別：ostringstream



❖ 須使用 sstream 標頭檔

輸出型C++字串資料串流類別（一）

- 產生串流物件

```
ostringstream foo ;
ostringstream bar1( ”Animal : ” ) ;
ostringstream bar2( string(”dog”) ) ;     // 須要標頭檔 string
```

- **str()：回傳串流物件內的資料成為一 C++ 字串**

```
int hr , min , sec ;
ostringstream time ;

cin >> hr >> min >> sec ;       // 輸入 : 18 4 15

time << ”現在時刻為 ”
     << setfill(’0’)
     << setw(2) << hr  << ’:’
     << setw(2) << min << ’:’
     << setw(2) << sec ;
cout << time.str() << endl ;   // 輸出 : 現在時刻為 18:04:15
```

輸出型C++字串資料串流類別（二）

- **str(string a) :**

  將 a 字串直接設定於串流物件內

```
ostringstream time ;
time.str(”20:12”) ;
time << ” [臺北時間]” ;

// 列印  20:12 [臺北時間]
cout << time.str() << endl ;
```

輸出型C++字串資料串流類別（三）

- 覆載 C++ 字串加法運算子

```
string operator+ ( const string& foo , double num ){
   ostringstream outstring ;
   outstring << num ;
   return foo + outstring.str() ;
}
```

如此

```
string city   = ”臺北市” ;
double degree = 25.7 ;
string temperature = city + ”氣溫為 ” + degree + ” 度” ;

// 列印 : 臺北市氣溫為 25.7 度
cout << temperature << endl ;
```

輸入型C++字串資料串流類別（一）

- 產生串流物件

```
istringstream  a ;
istringstream  b( ”海上生明月” ) ;
istringstream  c( string(5,’a’) ) ;
```

- **str()：回傳串流物件內的 C++ 字串**

```
int    i = 1 ;
string word ;
istringstream line(”海 上 生 明 月”) ;
cout << line.str() << endl ;
while( line >> word ){
   cout << i++ << ” : ” << word  // 每一個中文字分別佔用一行
        << endl ;
}
```

輸入型C++字串資料串流類別（二）

- **str(string a) :**

  將 a 字串直接設定於串流物件內

```
string   line ;
getline( cin , line ) ;          // 讀取一行資料到 line 字串
istringstream scin ;
scin.str(line) ;                 // 將 line 字串設定成 scin
                                 // 物件的資料來源

int no , sum = 0 ;
while( scin >> no ) sum += no ;   // 重複讀取資料後求和
cout << "數字和 : " << sum ;
```

❖ 以上唯有在 **scin** 串流物件讀取錯誤後才會跳離 **while** 迴圈，此時須利用 **scin.clear()** 清除錯誤旗幟，**scin** 方能繼續使用

範例：建構式數學 加減運算

- 利用字串串流物件協助讀取在一數學式中不同型別的資料

```
輸入 : 10 - 7 + 5 + 10 - 3
輸出 :
       > 10 -  7 =  3
       >  3 +  5 =  8
       >  8 + 10 = 18
       > 18 -  3 = 15
```

❖ 串流物件的錯誤旗幟要清除後才能繼續使用

範例：輸出檔案群組

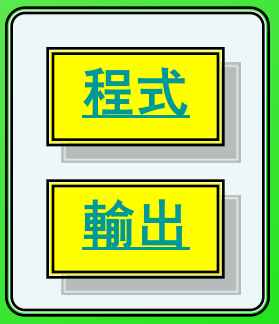
- 利用字串串流物件將不同型別的資料建構成檔案名稱

```
int   i ;
ostringstream outstring ;

...

outstring << "lottery" << setfill('0') << setw(3) << i ;

ofstream outfile( outstring.str().c_str() ) ;
```



範例：昇降梯模擬程式

- 每一層昇降梯都可按不等數量的其他樓層

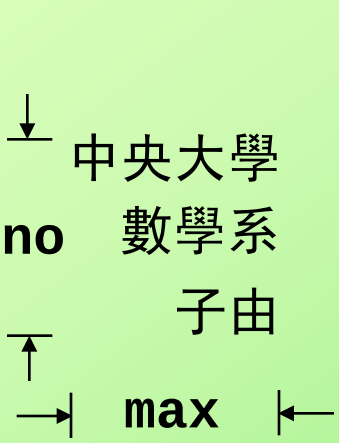
```
[1] : 3 5 10
[3] :
[5] : 8 2
...
```

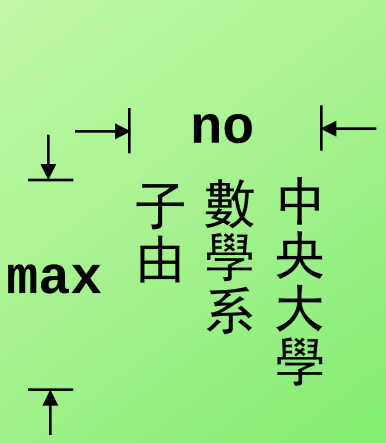
- 使用輸入字串串流物件持續讀取數字直到錯誤為止

```
int    no ;
string name ;
istringstream  stops ;
getline( cin , line ) ;          // 讀取整行
stops.str(line) ;                // 存入字串串流物件
while( stops >> no ){ ... }      // 由串流物件取出數字
```

程式 輸出

範例：橫行中文轉直行中文





- 策略：
  - 將每一橫行中文句子一一讀入字串串流物件陣列 `ostringstream lines[no]`
  - 使用一整數變數 `max` 計算所有橫行中最長的句子
  - 對調迴圈順序，以每次兩個字元方式分別列印每一個字串串流物件的內容