4. Functions in C++ and Methods in Java

4.1 Arguments Passing in C++

C/C++所提供的函式參數傳遞方式(Calling Modes)有，傳值法(call by value)，傳位址法(call by address)，與傳參考法(call by reference)三種。

4.1.1 call by value (傳值法)

格式:

函式定義/原型宣告: `void function(int x, int y);` //x, y 為 parameters
函式呼叫: `function(a, b);` //a, b 為 arguments

呼叫時，函式參數與主程式變數的關係，相當於：`int x = a;` `int y = b;`

說明:

傳值法的定義是函式的接收參數(parameter)，其型態為變數或物件，函式呼叫時的傳入值(argument)即是參數的初始值。

傳值法的函數呼叫只是將主程式的變數值，「拷貝」給函式中的對應參數，以後兩個變數間就不再有任何關係(二者有各自獨立的記憶體空間)。對函式內的變數或參數做任何運算都不會改變主程式中的變數值，此為傳值法的特點。

例:

```
int addbyone(int x )
{
    x++;
    cout<<x<<endl;          //執行後,x=101
    return x;
}

void main()
{
    int x = 100;
    int y = addbyone(x);
    cout<<x<<endl;          //函式呼叫後,x 仍保持為 100
}
```

程式範例: oop_ex29.cpp

注意要點:

如欲由函式傳回值，需使用 return 指令。但一函式只能執行 return 一次，故只能有一個傳回值，更改函式呼叫等號右邊的變數值。如欲使用函式來改變主程式中的多個變數值(傳回兩個以上的值)，則需使用傳位址法或傳參考法。

4.1.2 call by address (傳位址法)

格式:

```
函式定義/原型宣告:    void function(int *x, int *y);    //x, y 為 parameters
函式呼叫:             function(&a, &b);                 //&a, &b 為 argument
```

呼叫時，函式參數與主程式變數的關係，相當於：int* x = &a; int* y = &b;

說明:

傳位址法的定義是函式的接收參數(parameter)，其型態為變數或物件指標，函式呼叫時的傳入值(argument)是輸入變數或物件的記憶體位址。

傳位址法的做法是將傳值法中接收某輸入變數之值的拷貝，改用指標做參數來「指到」呼叫時之輸入變數，由於指標儲存的是記憶體位址，故呼叫時傳入的需是輸入變數或物件的記憶體位址(可使用「&」取址)。換句話說，函式呼叫時不拷貝給函式某變數或物件之值，而是傳其記憶體位址，告訴函式此資料之所在，相對的函式中需使用指標作為參數來接收傳入的位址(二者間相當於使用等號相連)。

如此一來，函式中的指標參數即定位到輸入變數或物件的記憶體位址，之後於函式內便可用依址取值(deference)的運算來直接存取其值。

例:

```
int addbyone(int* xptr )
{
    (*xptr)++;
    cout<<*xptr<<endl;          //執行後,*xptr=101
    return *xptr;
```

```
}

void main()
{
    int x = 100;
    int y = addbyone(&x);
    cout<<x<<endl;          //函式呼叫後,x 為 101
}
```

程式範例: oop_ex30.cpp

注意要點:

1. 使用傳位址法可使函式傳回兩個以上的值(或需用傳參考法)。其做法是可透過指標參數，將值存入呼叫時之輸入變數中，一函式可有一個以上的指標參數。
2. 如欲避免函式更改輸入變數之值，或使指標參數定址，可使用 const 關鍵字。

程式範例: oop_ex31.cpp

4.1.3 call by reference (傳參考法)

格式:

```
函式定義/原型宣告:    void function(int& x, int& y);   //x, y 為 parameters
函式呼叫:             function(a, b);                  //a, b 為 arguments

呼叫時，函式參數與主程式變數的關係，相當於： int& x = a;   int& y = b;
```

說明:

傳參考法的定義是函式的接收參數(parameter)，其型態為變數或物件的參考，函式呼叫時的傳入變數或物件(argument)即為參考參數所參照。

使用傳參考法，是將某函式之參數，宣告為參考型態，呼叫時直接輸入某變數或物件，函式之呼叫相當於二者間使用等號相連，如此一來，函式之呼叫方式

雖與傳值法相同，但其作用已不是輸入變數或物件的值之拷貝，而是使得函式中的參數成為輸入變數或物件的參考(別名)，二者均代表相同的一塊記憶體空間，改變函式中參考參數的值，也等於改變了主程式中其所對應的輸入變數或物件的值。

例:

```
int addbyone(int& xref )
{
    xref++;
    cout<<xref<<endl;        //執行後,xref=101
    return xref;
}

void main()
{
    int x = 100;
    int y = addbyone(x);
    cout<<x<<endl;           //函式呼叫後,x 為 101
}
```

程式範例: oop_ex32.cpp

注意要點:

1. 使用傳參考法與傳位址法有同樣的效果，唯形式有所不同，C++提出傳參考法主要是為了簡化傳位址法中必需要做的取址以及依址取值等運算。
2. 使用傳參考法或傳位址法來做參數傳遞，可節省拷貝傳遞參數之時間及記憶體空間，尤其是當傳遞的參數為多筆資料構成的物件。
3. 如欲避免函式更改輸入變數之值，可使用 const 關鍵字。

程式範例: oop_ex33.cpp

案例研究：使用傳值法，傳位址法及傳參考法製作函式，來對輸入其之兩變數之值做對調.

程式範例: oop_ex34.cpp

4.2 Arguments Passing in Java

Java 的參數傳遞方式只有一種 – 傳值法(call by value)。與 C++的傳值法比較，其運作機制相同，但基於 Java 的資料型態，其表現之性質與 C++中的傳值法並不相同。

格式:

函式定義: void function(int x, User y, int[] z); //x, y, z 為 parameters
函式呼叫: function(a, b, c); //a, b, c 為 arguments

Java 的資料型態，分成 primitive data type，以及 reference type。所謂傳值法，即是函式呼叫時所傳入的是這些型態變數的儲存值之拷貝。

對於 primitive data type 之變數，參數傳遞的性質與 C++相同，亦即函式呼叫時，輸入變數與函式的接收參數間只是值的拷貝，之後彼此無關。

對於物件或陣列，Java 是使用 reference 的機制對其構成資料做存取，reference 的儲存值為其所參照的物件或陣列的記憶體位址，將 reference 的儲存值傳入函式，亦就是將物件或陣列的記憶體位址傳入，故其性質與 C++中的傳位址法相同，只是不需做依址取值的動作。

程式範例: oop_ex35.java oop_ex36.java

4.3 Passing Arrays to Function in C++

於 C++程式中將陣列傳入函式，是使用傳位址法的機制，亦即是將陣列所存空間之記憶體起始位址傳入函式，而非將陣列元素逐一拷貝致函式之中。其機制與 Java 的情況類同，唯格式上稍有不同。

格式:

函式定義: void function(int arrayname[ ], int size);
or void function(int* arrayname, int size);

函式呼叫: function(arrayname, size);

說明:

呼叫函式時，所輸入的是「陣列名」，我們知道一陣列的陣列名其實為一定址指標，其儲存值固定為此陣列的起始位址，故呼叫時所傳入的位址。

對於傳入的位址值，函式之參數列之接收參數，一般是使用[ ]來宣告一接收陣列，由於[ ]中並無指定陣列長，故其只產生一名稱為陣列名指標，而無配置陣列之空間，傳入位址值即存可於此「陣列名」，另一做法是直接產生適當型態的指標來接收。

我們於函式中必須要知道陣列的長度，以便能對輸入陣列之所有元素做運算，但是 C++之陣列不像 Java 之陣列可由 length 之屬行得知其長度，故陣列之長度，常需為輸入陣列之伴隨參數。

傳遞陣列的效果相當於之前的傳址法，亦即主程式中的陣列與函式中的陣列參數對應到相同的記憶體空間，故於函式之中對所傳入的陣列作改變，亦等於對主程式中之相對陣列作改變。

例:

```
void function(int x[], int size )
{
    x[3] = …..;
}

void main()
{
    int x[] = {1,2,3,4,5};
    int size = 5;
    function(x, size);
}
```

程式範例: oop_ex37.cpp

4.4 Function Overloading (in C++ and Java)

說明:

函式覆載是指不同的函式定義可以有相同的函式名稱(函式的名稱可以重

複)，但這些同名函式必須具有不同的參數列，編譯系統可以就其彼此參數列之不同為依據，判斷選擇出其中最接近呼叫條件的一個函式來執行。

不同的參數列是指同名的函式，其後之參數列具有不同的參數個數、或不同的資料型態、或二者皆有。程式將依呼叫時所用的參數，判斷出最接近者，並執行此一函式。注意的是，不同傳回值型態，不能作為辨別同名覆載函式之依據。

函式覆載的好處在於可產生多個同名函式以用於處理相同或相近的工作，但各自針對不同的資料型態。

函式覆載適用於獨立的函式，以及類別的函式成員，例如一類別有多個建構函式。

例:

以下為合法的函式覆載：

```
double max(double x, double y);
int max(int x, int y);                  //  不同參數資料型態

int min(int x, int y);
int min(int x, int y, int z);           //  不同參數個數
```

以下的函式為不合法的函式覆載：

```
int sum(int x, int y);
double sum(int x, int y);               //不同傳回值型態
```

程式範例: oop_ex38.cpp

注意要點:

當呼叫所用的參數需作變數轉換，而造成函式條件相當而無法決定呼叫何者時，即造成錯誤。**例：** cout << max(5, 55.5);

4.5 Default Argument for C++ Functions

C++允許函式預設其輸入參數的初值，增加函式的使用彈性。

格式:

函式預設初值的位置，可在函式定義或函式原型宣告處，但同時只能有一處

有初值之設定，並於需位於函式被呼叫之前，故可能有以下兩種格式。

例:

(1)
```
int sum(int x = 0, int y = 0)   //函式定義
{
    return x+y;
}

void main()
{
    …..
    result = sum(a, b);
    ……
}
```

(2)
```
int sum(int x = 0, int y = 0);          //函式原型宣告

void main()
{
    …..
    result = sum(a, b);
    ……
}

int sum(int x , int y)          //函式定義
{
    return x+y;
}
```

注意要點:

1. 函式的預設初值，是在函式呼叫時，其相對位置的參數之輸入值未被指定時，系統才會自動引用預設值；如有輸入值，則使用輸入值為參數之值。

2. 如有不預設初值的參數，必須集中在函式參數列的右側，因為無論有無預設值，輸入值均是由參數列的最左邊開始一一設定。

例： int sum(int x=0, int y=0, int z=0);
int sum(int y, int x=0, int z=0);
int sum(int x, int y, int z=0);

以下不可：

int sum(int x=0, int y, int z=0);

程式範例: **oop_ex39.cpp**

The following function-related topics will be introduced later.

Operator Overloading in C++
Exception Handling in Java