國立屏東大學 資訊工程學系 C++程式設計入門教材

10. 函式

函式(Function)是模組化程式設計的基石，有了它以後，程式可以將各自不同的功能寫成不同的函式，除了可以用來組合出完整的系統或軟體功能外，更重要的是，這些我們所設計好的函式，以後還可以重複地在其它的程式中被使用 — 就如同我們已經使用過許多C++事先定義好的函式一樣。我們可以把函式視為一個 “ 小程式 ”，它可以接收輸入、進行特定的處理並且輸出資料。如同程式可以使用IPO模型分析，“ 小程式 ” (也就是函式)當然也可以用IPO模型加以分析，請參考figure 1。

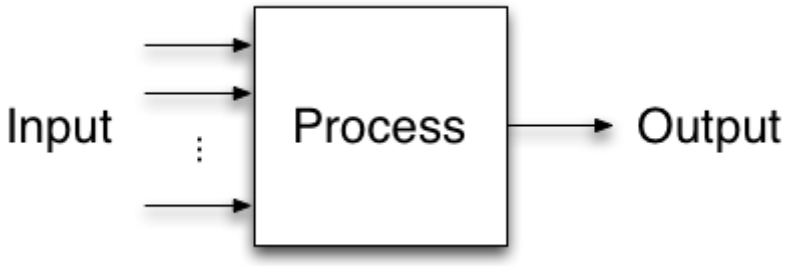

Fig. 1
要特別注意的是：「一個函式可以有0個或多個輸入，但只能有0個或一個輸出。」，我們將函式的輸入稱為「參數(parameter)」，將其輸出稱為傳回值(return value)。

C++語言的函式在某方面來說與數學的函數十分類似，且兩者的英文皆為function。本書針對在數學與C++語言，將function一詞分別譯做函數與函式，以視區別。

10.1 函式定義

函式在被使用之前必須先加以定義，其定義語法如下：

函式定義語法

傳回值資料型態 函式名稱 (參數*)
{
  敘述*
}

每個函式必須提供其傳回值的型態定義，所謂的傳回值即為IPO模型中之Output部份，這也就始是在上述語法定義裡面的「傳回值資料型態」，它有以下的規則：

- 一個函式至多只能有一個輸出，除了不能傳回陣列外，所有資料型態皆可。
- 若沒有要傳回的值(沒有輸出)，則必須寫做void表示無型態。

接著「函式名稱(Function Name)就是函式的名稱，建議依函式的功能使用較具意義的名稱，以增進程式的可讀性(Readability)函式名稱如同變數名稱一樣，在語法結構上都是「識別字」，其命名規則如下：

- 只能使用英文大小寫字母、數字與底線(_)
- C++語言是case-sensitive的語言，意即大小寫會被視為不同的字元
- 不能使用數字開頭
- 不能與<nowiki>C++</nowiki>語言的保留字相同

在函式名稱後以一組小括號「()」來定義輸入(有時亦稱為傳入)的0或多個參數，也就是IPO中的Input部份，其語法定義如下：

函式參數定義語法

資料型態 參數名稱 [, 參數]*

在上面的語法說明中，「[]」為選擇性的語法單元，其後接續「*」表示該語法單元可出現0次或多次；「?」表示出現0次或1次。另外還有「+」代表1次或多次。

- 函式可以有0個或多個輸入參數
- 若超過一個以上的參數，則任意兩個參數間必須要使用「,」隔開
- 與變數宣告相同，每個輸入參數必須定義其名稱與其所屬之資料型態。
  - 參數名稱亦為識別字，同識別字命名規則
  - 參數的資料型態並無限制
- 參數的值於使用函式(函式被呼叫)時傳入，可在函式內部的處理敘述中使用

在變數宣告時我們可以使用int i,j;這種方式，一次宣告多個整數；但在函式的參數定義時，並不可以使用這種方式。如果需要傳入多個參數，那麼你必須分別宣告，例如：

```
void foo(int i, int j)
{
}
```

最後，在一組大括號「{}」內定義函式的處理敘述，也就是IPO模型中的Process部份。由於使用了大括號「{}」，因此此部份又被稱為是函式內容區塊。這些敘述通常又可以再區分為兩部份，首先是變數宣告的部份，我們可以在此為函式的處理宣告所需要的變數，其規則與一般變數宣告一致；接著才是處理敘述的部份，你可以使用任意的C++語言敘述。相關規則如下：

- 若在函式內沒有變數宣告的需求，則可以省略
- 若函式有定義傳回值的資料型態，那麼就表示此函式有要傳回的資料，因此在處理敘述中至少須包含一行return敘述
    - return敘述語法為「return 運算式;」，此處的運算式的運算結果其型態必須與傳回值資料型態相同

以下是一個典型的函式範例：

```
int sum( int x, int y)
{
   int result;
   result = x+y;
   return result;
}

int main()
{
    int a, b;
    a=10;
    b=45;
    int x=33,y=77;

    cout << sum(a,b) << endl;
    cout << sum(x,y) << endl;
    cout << sum(a,sum(a,b)) << endl;
    cout << sum(3,5) << endl;
}
```

以這個函式為例，其函式名稱為sum，也就是加總的意思，其傳回值資料型態定義為int，傳入的參數為兩個int整數，分別命名為x與y，此函式內容先定義了所需要的變數result，然後再計算x+y的值後使用return敘述將結果傳回。

接下來，讓我們來看另一個以印出程式資訊為目的的函式定義範例：

```
void showInfo()
{
   printf("This program is written by Jun Wu.\n");
   printf("All right reserved.\n");
}
```

好了，就這麼簡單，接下來讓我們來看看一個你已經常常在使用的函式…

10.2 main()函式

打從我們一開始學習C++語言，我們就已經開始使用函式定義了:

```
C++</nowiki>程式 hello.cpp >
/* Hello, <nowiki>C++</nowiki>! */
// This is my first <nowiki>C++</nowiki> program

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, <nowiki>C++</nowiki>!" << endl;
    return 0;
}
```

在這個程式中，我們已經自己定義了一個函式 – main()不過main()函式是一個特別的函式，它是程式的進入點，同時它有以下的特別規則：

- main()函式的傳回值可以為int或void
  - 傳回值定義為int時：
    - 可使用return敘述傳回整數以說明程式執行狀態
    - return敘述可以省略，此時會傳回預設值0
  - 傳回值定義為void時：
    - 不可使用return敘述
    - main()函式還是會傳回預設的傳回值0

參考上述規則，以下的main()函式都是正確的:

```
int main()
{
}

int main()
{
   return 0;
}

void main()
{
}
```

但下面這個main()函式是錯誤的

```
void main()
{
   return 0;
}
```

每當程式被系統載入加以執行時，main()函式是程式首先也是唯一會被加以執行的程式區塊，一旦執行完main()函式的內容區塊，程式就會結束。關於main()函式的傳回值，是回傳給誰呢？由於我們是在終端機模式下執行程式，所以可以在終端機裡取得main()函式的傳回值，請參考下面的例子：

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello!" << endl;
   return 1;
}
```

當程式執行完成後，main()函式的傳回值會被存放在系統的環境變數中，你可以使用以下的指令取得其值：

```
[16:41 user@ws example] ./a.out
Hello!
[16:41 user@ws example] echo $?
1
[16:41 user@ws example]
```

另外還要注意，系統預設取回的main()函式傳回值為一個8位元的unsigned int，若傳回值為超出此範圍時，數值會有溢位或不正確的問題[1)]。

10.3 函式呼叫

先讓我們回想一下，本書到目前為止，你已經使用過(呼叫過)哪些函式了呢？

回想看看您是如何使用這些別人幫您寫好(C++語言內建)的函式呢？其實函式的使用方法很簡單，只要在程式敘述中使用該函式的名稱，並在一組大括號「()」內傳入引數(Arguments)給函式使用即可，要注意的是引數除了是值外，也可以是運算式。

Parameters vs.Arguments (到底是參數還是引數?) 所謂參數(Parameter)是在定義函式時所指定的輸入變數，至於引數(Argument)則是指在呼叫函式時所傳入的數值或運算式。

請參考下面的例子：

```
#include <iostream>
using namespace std;

int sum( int x, int y )
{
   return x+y;
}

int main()
{
   int i=2, j=4, k=6;
 //使用=將函式的傳回值指定給變數i
   i = sum(10, 30);
   cout << "10+30 = " << i << endl;

 // 直接把函式的傳回值當成一般的值，並傳給cout做為輸入
   cout << "5+6 = " << sum(5,6) << endl;
 // 引數的部份也可以為變數
   cout << "5+j = " << sum(5,j) << endl;
 // 傳回值可做為運算式的一部份
   cout << "5+j+k = " << 5+sum(j,k) << endl;

 // 引數的部份也可以為運算式
   cout << "5+j+k = " << sum(5,(j+k)) << endl;

 // 引數的部份也可以為另一個函式呼叫
   cout << "5+j+k = " << sum(5,sum(j,k)) << endl;
}
```

函式呼叫時的引數也可以為陣列，例如下面的例子：

```
void showData(int data[])
{
   int i;
   for(i=0; i<(sizeof(data)/sizeof(data[0]));i++)
      cout << data[i] << " " << endl;
}

int main()
{
   int mydata[5]={2, 4, 6, 8, 10};
   showData(mydata);
}
```

10.4 變數範圍

只可以在其所屬的程式區塊內使用的變數稱為區域變數(Local Variables)，請參考下例，我們標示了每個區塊可使用的變數：

```
int foo(int i, int j)
{
 //這裡可以使用的變數為i,j
   return i+j;
}

int main()
{
 //這裡可以使用的變數為x,y
   int x=3, y=5;

   cout << foo(x,y) << endl;
}
```

若在不同區塊內，有同樣的變數名稱時，則以該變數所在的區塊為依據，請參考下例：

```
int foo(int x, int y)
{
 // 這裡可以使用的變數為x,y
 // 可以視為 foo.x與foo.y
   if(x>y)
      return x;
   else
      return y;
}

int main()
{
 // 這裡可以使用的變數為x,y
 // 可以視為 main.x 與main.y
   int x=3, y=5;

   cout << foo(9,8) << endl;
}
```

宣告在全部函式外部的變數稱為全域變數(Global Variable)，它們可以在程式的任何地方使用。但若某區塊內有同樣名稱的變數，則以該區塊為主。請參考下面的例子：

```
// 可以視為global.x與global.y

int x,y;

int foo(int x, int y)
{
 // 這裡可以使用的變數為x,y -> foo.x, foo.y

   if(x>y)
      return x;
   else
      return y;
}

int foo2(int i, int j)
{
 // 這裡可以使用的變數為x,y,i,j -> global.x, global.y, foo2.i, foo2.j

   if((i>x)&&(j>y))
      return i+j;
   else
      return x+y;
}


int main()
{
 // 這裡可以使用的變數為x,y -> global.x與global.y
   cout << foo(9,8) << endl;
   cout << foo2(3,6) << endl;
}
```

10.5 遞迴呼叫

以下是一個計算N!(階乘)的函式定義範例：

```
int factorial( int n)
{
   int result=1, i;
   if((n==0)||(n==1))
      return 1;
   else if (n>1)
   {
      for(i=2;i<=n;i++)
      {
         result*=i;
      }
      return result;
```

```
   }
   else
      return (-1);
}
```

以這個函式為例，其函式名稱為factorial，也就是階乘的意思，其傳回值型態定義為int，傳入的參數為整數，命名為n，此函式內容先定義了所需的變數result與i，然後再計算n!並以return敘述將結果傳回。其中若參數n的值為0或1時，依階乘定義0!與1!皆為1，使用return敘述傳回1。要注意的是一旦使用了return敘述，函式就會將指定的結果傳回，剩餘未執行的程式碼不會被執行。若n的值不為0或1，則比較n是否大於等於2，使用一個迴圈來計算n!並使用return敘述將結果傳回。若n的值不為0或1且不大於等於2，那麼唯一的可能是n為負數，此時傳回-1表示錯誤。

這個計算階乘的程式，還可以改用一種叫做「遞迴(Recursive)」的方式重新設計 — 在函式的定義中呼叫自己。例如：

在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「

> 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > > 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > > > 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > > > > 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > > > > > 在一個遙遠的地方，有一座山上，山頂有一間破廟，廟裡有一個小和尚，他做了一個夢，夢到「
> > > > > > > ...」
> > > > > > 」
> > > > > 」
> > > > 」
> > > 」
> > 」
> > 」
> 」

OK這就叫做遞迴。讓我們使用遞迴重新改寫計算階乘的程式：

```
int factorial(int i)
{
   if((i==0)||(i==1))
      return 1;
   else
      return i*factorial(i-1);
}
```

以計算5的階乘為例，呼叫factorial(5)的過程如下：

factorial(5) →

> 5 * factorial(4) →
>
> > 5 * 4 * factorial(3) →
> >
> > > 5 * 4 * 3 * factorial(2) →
> > >
> > > > 5 * 4 * 3 * 2 * facotrial(1) →
> > > >
> > > > > 5 * 4 * 3 * 2 * 1

10.6 函式原型與標頭檔

如同變數的使用一樣，必須在使用前先進行過變數的宣告；函式的使用(也就是函式呼叫)，也必須在呼叫前，先提供該函式的定義。通常必須在程式碼的開頭處進行函式的定義，例如：

```
#include <iostream>
using namespace std;

void foo()
{
   cout << "This is foo." << endl;
}

int main()
{
   foo();
   cout << "This is the main function." << endl;
}
```

程式當然還是從main()函式開始執行，在main()函式的內容區塊中，我們使用了一個自行定義的函式foo()。為了讓在main()函式內容區塊中，能正確地呼叫foo函式，我們將其函式定義在main()函式之前。但這種做

法要求所有要使用到的函式都必須定義在使用之前，有時我們並不想這麼做，那麼你可以在程式開頭處先宣告函式的原型(Prototype)，然後在其它地方提供完整的函式定義，請參考下面的例子：

```
#include <iostream>
using namespace std;
void foo();

int main()
{
   foo();
   cout << "This is the main function." << endl;
}

void foo()
{
   cout << "This is foo." << endl;
}
```

在這個例子中，我們將函式定義在main()函式之後，但是為了讓main()函式內容區塊還是能呼叫foo函式，所以必須在main()函式前先提供foo()函式的原型說明。函式的原型宣告語法如下：

函式原型宣告語法

傳回值資料型態 函式名稱 (參數*);
{
  敘述*
}

其時它與函式定義的語法完全相同，只不過少了函式的內容區塊而已，並且要在結尾處加上一個「;」分號。

所謂的函式原型(Prototype)是指函式的定義，但不包含其內容區塊。從原型所提供的資訊來看，我們已經可以知道函式的名稱、輸入參數的個數與型態、還有傳回值的型態為何，這些資訊被稱為是函式的介面(Interface)，而這些資訊已經足夠讓我們呼叫這個函式。

至於函式的內容區塊，則是函式實際進行處理的地方，也就是說內容區塊決定了函式所提供的功能為何？相較於函式的介面，內容區塊被稱為函式的實作(Implement)。

如果一個或一個以上的函式，常常會被不同程式使用，那麼將它的函式定義在所有使用到的程式中，應該

是一個很糟的決定，不但麻煩而且日後很難維護。比較常見的做法是，將函式另外定義在獨立的程式檔案中，並且提供關於函式原型宣告的標頭檔案(Header File)，日後使用這些函式的人，只要在其程式碼中使用#include指令來載入你所提供的標頭檔案，然後在編譯時將實際包含有函式定義的檔案一起納入編譯，如此即可完成程式的開發。請參考以下的範例：

```
// 此處為定義sum()函式的標頭檔
int sum(int x, int y);
```

```
// 此處為sum()函式的實作
int sum(int x, int y)
{
   return x+y;
}
```

```
#include <iostream>
using namespace std;
// 此處載入了sum()的函式標頭檔
#include "sum.h"

int main()
{
   cout << "3+5=" << sum(3,5) << end;
}
```

請分別使用以下指令來編譯與執行：

```
[12:21 user@ws example] C++ -c sum.cpp -o sum.o // 編譯產生不可單獨執行的sum.o目的檔
[12:21 user@ws example] C++ test.cpp sum.o     // 結合sum.o一起編譯產生可執行檔
[12:21 user@ws example] ./a.out
3+5=8
[12:21 user@ws example]
```

如果你的工作是專責開發函式，供其它程式設計師使用，那麼有可能不想公開函式的實作細節。此時你只要提供sum.h與sum.o給其它程式設計師，他們就可以使用你所設計的函式而不知道其實作的方法。

10.7 預設引數值

承襲自C語言的C++語言，其大部份的語法與功能都與C語言相同。本章截至目前為止的內容，也都和C語言完全相同。不過做為C語言的後繼者，C++語言還是有許多創新的功能，例如本節要介紹的「預設引數值(Default Argument)」— 允許函式的引數可以有預設的數值:

```
#include <iostream>
using namespace std;

double test (double a, double b = 7)
{
   return a - b;
}

int main ()
{
   cout << test (14, 5) << endl;    // Displays 14 - 5
   cout << test (14) << endl;       // Displays 14 - 7

   return 0;
}
```

10.8 函式多載

有時候，我們會需要針對不同的資料型態設計功能相同的不同函式：

```
int sum2Int(int a, int b)
{
   return a+b;
}

double sum2Double(double a, double b)
{
   return a+b;
}
```

這種做法在使用函式時，必須視所要處理的資料之型態的差異，選擇呼叫對應的版本。

C++語言還有一項很強大的新功能 — 函式多載(Function Overloading)，它允許我們為函式設計多個版本，它們可以具備同樣的函式名稱，但依其參數的不同可被視為不同的版本。因此，我們可以使用這種方式，將適用於不同資料型態的不同版本(但通常功能相同)的函式改以「同名異式」的方式設計如下：

```
int sum2Num(int a, int b)
{
   return a+b;
}

double sum2Num(double a, double b)
{
```

```
   return a+b;
}
```

如此一來，在使用時就不必依據不同的資料型態使用不同的函式，但若是要支援更多的型態，就必須提供更多版本的實作。

10.9 函式模板

上一小節介紹的多載，解決了一部份的問題，但卻又留下了一些遺憾～～

不過C++還提供了函式模板(Function Template)的功能，可以將所有問題解決！我們可以設計一個 “ 模板 ” 讓它可以套用在任意的資料型態，如此一來，只要設計一次，就可以適用於所有的資料型態。嗯～～聽起來很棒！ 但要如何進行呢？ 請參考以下的程式碼：

```
template<class T>
T sum2Num(T a, T b)
{
   return a+b;
}
```

Function Template使用template <class T>做為前綴(template prefix)代表將T視為某種型態；未來當sum2Num()函式被呼叫時，將會視當時所傳入的參數來決定T究竟是何種型態，並將函式內容中所有出現的“T”(是型態，不是技術犯規)，以該型態進行代換。

以下是一個完整的範例：

```
#include <iostream>
using namespace std;

template<class T>
T sum2Num(T a, T b)
{
   return a+b;
}

int main()
{
   int x=4, y=6;
   double a=3.123, b=2.342;
   cout << sum2Num(x, y) << endl;
   cout << sum2Num(a, b) << endl;
}
```

好了，誰告訴你有一好就沒有兩好 C++同時滿足了你多種需求～ 嗯，很棒！

[1)] 關於此點，不同作業系統或有不同，請自行查閱相關資料。