1

Lecture 11

自訂函式
函式覆載

2

回顧

- 函式 (function)
  - 獨立的程式單元
  - 類似現實生活中的工廠，給它一些原料，製造出一些成品。
  - e.g. y = cos(x);
    - cos 是一個函式，傳入一個弧度值，並回傳一餘弦函數的值。
    - 至於如何將弧度值算成餘弦函數的值，被實現在 cos 這個函式裡面，一般而言我們不知道它怎麼辦到的，也通常不太在乎。
- C/C++ 語言有訂出標準的函式庫 (library, 收藏函式的圖書館)。而要使用這些標準的函式庫，則必需要含括入相對應的標頭檔 (header)

3

自訂函式

user-defined functions

4

自訂函式 (user-defined functions)

- 函式是一個程式的單元 (unit) 或是模組 (module)，為多行程式敍述的組合，用來**簡化**程式開發。
  - **常常**需要加總一個陣列
  - **常常**需要從一個陣列裡找最大值
  - **常常**需要計算一個函數的值
- 使用時機
  - 需要的功能不存在標準函式庫裡
  - 將重複使用到的程式碼變成獨立運作的模組
  - 增加程式的可讀性

5

11-1.cpp

執行結果
M1 M2 A1 B1 A2 M3 B1 M4

```
#include <iostream>
using namespace std;
1 void funA();
1 void funB();

int main() {
  cout << "M1 ";
  cout << "M2 ";
2 funA();
  cout << "M3 ";
2 funB();
  cout << "M4 ";

  return 0;
}
```

1 函式的原型宣告 (prototype)
2 函式的呼叫 (invocation, call)
3 函式的定義 (definition)

```
void funA() {
  cout << "A1 ";
2 funB();
  cout << "A2 ";
}
void funB() {
  cout << "B1 ";
}
```

6

11-2.cpp

```
#include <iostream>
using namespace std;
double f(double);          // 此被稱為函式的原型 (prototype)宣告
int main() {
  double a, q;
  for(q=-5;q<=5;q++) {
    a = f(q);
    cout << q << ", " << a << endl;
  }
  return 0;                // 函式呼叫 (function call, function invocation)
}
double f(double x) {       // 此被稱為函式的定義
  return x*x+3*x+2;
}
```

執行結果

```
-5, 12
-4, 6
-3, 2
-2, 0
-1, 0
0, 2
1, 6
2, 12
3, 20
4, 30
5, 42
```

7

11-2.cpp 的說明

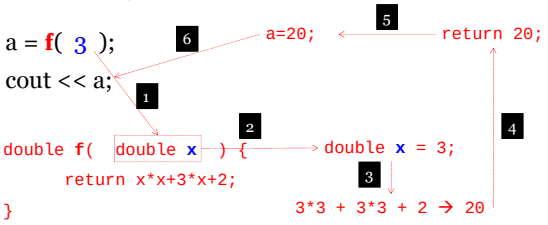
- 程式中 f(x) 即為一個函式 (類似於數學上的函數)

```
double f(double x) {
    return x*x+3*x+2;
}
```

- 函數傳入 0 或多個參數 (parameter, argument)，運作後回傳 (return) 0 或 1 個計算結果。
  - 在以上範例，傳入一個參數 x, 並傳出 x^2+3x+2 的計算結果。

8

函式呼叫時所發生的事情:



9

11-3.cpp

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

int sum(int);

int main() {
  cout << "sum(10)=" << sum(10) << endl;
  cout << "sum(100)=" << sum(100) << endl;

  return 0;
}
```

原型宣告的目的在於知會編譯器
1. 函式名稱 (sum)
2. 函式的傳入參數個數(1 個)
3. 個別參數的型別 (int)
4. 函式回傳的資料的型別 (int)

此處為函式呼叫，編譯器會根據原型宣告檢查
1. 參數個數
2. 個別傳入參數的型別是否正確

10

11-3.cpp (continued)

此處為函式定義 (function definition)

```
int sum(int x) {
  int i, ans = 0;
  for(i=1;i<=x;i++) {
     ans = ans + i;
  }
  return ans;
}
```

此行為函式的界面 (interface)
-編譯器會檢查與出現過的原型宣告是否一致
- 所有資料的「進出」都應透過此界面

此處為函式本體 (function body)
-定義實質函式的運作

注意到 sum 函式裡面一共宣告了三個變數: x, i, ans. 這三個變數被稱為**區域變數**，僅作用於此函式內而不會影響到其它函式內相同名稱的變數。

11

自訂函式

- 自訂函式可分為兩部份：
  - **原型宣告**
  - **函式定義**

12

原型宣告
prototype declaration

- 原型宣告的目的在於知會編譯器一個函式的：
  1. 函式名稱
  2. 函式的傳入參數個數
  3. 個別參數的型別
  4. 函式回傳的資料的型別
- 原型宣告必需在函式第一次被使用之前出現
- 原型宣告時之語法 (詳細說明見函式定義)
  回傳資料型別 函式名稱 (*參數列*) ;

```
double f(double);   int sum(int);   void do(int);
```

- 原型宣告之參數列個別參數的名稱不重要，可有可無。

13

原型宣告
prototype declaration

- 以下各框框內的原型宣告皆代表同一函式，不同框框代表不同函式的原型宣告。注意到原型宣告以分號作結尾。
- 在宣告函式時，可在最前面加上 inline ，此關鍵字提示編譯器此函式可考慮將其內容作「行內展開」(inline expansion) 在呼叫函式處，以期加速程式的執行。
- 注意到相同名稱、傳入參數型別不同時，視為不同函式。

```
inline int sum(int);
inline int sum(int i);
inline int sum(int j);
inline int sum(int k);
```

```
double f(double);
double f(double x);
double f(double xyz);
double f(double qq);
```

```
double f(float);
double f(float x);
double f(float xyz);
double f(float qq);
```

14

函式定義 - overview
function definition

函式定義 (function definition) 語法如下：

回傳資料型別 函式名稱 (*參數列*) {
  函式本體
}

```
int sum(int x) {
  int i, ans = 0;
  for(i=1;i<=x;i++) {
    ans = ans + i;
  }
  return ans;
}
```

```
int plus(int x, int y) {
  return x+y;
}
```

```
void show(int x, int y) {
  cout << x+y;
}
```

15

函式定義 – return data type
function definition

- **回傳資料型別** (return datatype)
  - 定義了函式回傳的資料型別 (double, int, int*, float*, char, ...)
  - 亦可宣告為 void, 代表無回傳資料

```
int plusOne(int x) {
    int y = x + 1;
    return y;
}
```

```
void printTwoNumbers(int x, int y) {
    cout << x << "+" << y << "=" << x+y << endl;
}
```

16

函式定義 – function name / function identifier
function definition

- **函式名稱** (function identifier)
  - 函式名稱的限制同變數名稱的限制
  - 可使用英文字母、底線符號、與數字
  - 第一個字必需是英文字母或是底線符號

```
void printTwoNumbers(int x, int y) {
    cout << x << "+" << y << "=" << x+y << endl;
}
```

```
int plusOne(int x) {
    int y = x + 1;
    return y;
}
```

17

函式定義 - arguments
function definition

- **參數列 (argument list / parameters)**
  - 定義了函式執行時所需要傳入的資料數量與每個資料的型別。
  - 亦可宣告為 void 或省略, 代表無傳入的資料。
  - 參數列所列之變數可視為函式內的部份**變數宣告**
  - **亦可為參數設定內定值／預設值**

```
void helloWorld(void) {
    cout << "Hello World";
}
```

```
void printTwoNumbers(int x, int y=5) {
    cout << x << "+" << y << "=" << x+y << endl;
}
```

18

函式定義 – function body
function definition

- **函式本體 (function body)**
  - 定義了函式被呼叫時的執行動作
  - 可宣告自己所需使用的變數，稱為區域變數 (local variable)
  - 透過 return 指令將資料回傳給呼叫的程式 (caller)，並繼續執行呼叫者下一行的程式
  - 若函式無回傳值，則無需有 return 指令出現。

```
int sum(int x) {
  int i, sum = 0;
  for(i=1;i<=x;i++) {
    sum += i;
  }
  return sum;
}
```

```
void hi(char *x) {
  cout << x << endl;
}
```

```
void hi(char *x) {
  cout << x << endl;
  return
}
```

19

自訂函式

- 自訂函式分為兩部份：**原型宣告**與**函式定義**。
- 若函式定義出現在第一次使用前，則原型宣告可省略
  - i.e. 函式定義也可作為原型宣告。
- 函式原型宣告與函式定義應一致 (參數個數、回傳型別 ...)
  - 原型宣告時參數名稱不重要，可以和函式定義的參數名稱不同。
  - 一個程式裡可使用多個函式，函式裡也還可以再呼叫其它函式
  - main () 事實上也是一個函式，只是它代表了程式主要開始的地方。

```
int sum(int);
int sum(int i);
int sum(int j);
int sum(int k);
```

```
int sum(int x) {
      int i, sum = 0;
      for(i=1;i<=x;i++) {
            sum += i;
      }
      return sum;
}
```

20

11-4.cpp

```
#include <iostream>
using namespace std;
void nStar(char symbol, int x=20) {
  for(int i=0;i<x;i++)
    cout << symbol;
  cout << endl;
}

int main() {
  for(int i=1;i<=5;i++)
    nStar('+', i);
  cout << "\n";
  for(int j=0;j<5;j++)
    nStar('*');
  return 0;
}
```

執行結果

```
+
++
+++
++++
+++++

********************
********************
********************
********************
********************
```

在此程式中，函式 nStar 定義在第一次使用之前，故可以省略函式的原型宣告，並**以函式定義作為函式的原型宣告**。

在函式被執行時，傳入的變數名稱不重要、重要的是位置。第一個位置的資料被存入函式內的 symbol 變數；第二個位置的資料被存入函式內的 x 變數

21

參數預設值

- 在前範例中，我們給了第二個參數一個預設值 20，因此當呼叫此函式而沒有給第二個參數時，nStar函式執行時，會令 x 為 20 來執行函式的內容。
  - nStar('*');
- 定義函式時，一個參數若有預設值，則其之後的所有參數都必需有預設值!!
- 在函式呼叫時，任意兩逗點間或逗號與括號間都必需有值，因此有預設值的參數必需集中於參數列的後面
  - nStar('+');

```
void nStar(int x=20, char symbol) {
  for(int i=0;i<x;i++)
    cout << symbol;
  cout << endl;
}
```

22

11-5.cpp

```
#include <iostream>
using namespace std;
void test(int a=1, int b=2, int c=3, int d=4) {
  cout << a << ", " << b << ", ";
  cout << c << ", " << d << endl;
}
int main() {
  test();
  test(100);
  test(100,200);
  test(100,200,300);
  test(100,200,300,400);
  return 0;
}
```

執行結果

```
1, 2, 3, 4
100, 2, 3, 4
100, 200, 3, 4
100, 200, 300, 4
100, 200, 300, 400
```

錯誤宣告

```
void test(int a=1, int b, int c=2, int d) {...}
void test(int a=1, int b=2, int c, int d) {...}
```

錯誤呼叫

```
test(,,,100)
test(100,, 300, 400)
test(100, , , 400)
test(100, 200, 300, )
```

23

函式內之變數

- 函式內所宣告的變數稱為區域變數 (local variable) ，不同函式的區域變數各自獨立，可使用同樣名稱而不互相干擾。
- 因此，當在程式設計初期在分析時，即可將不同的工作步驟寫為不同函式並交由不同人去將函式實作出來，而不用耽心程式會互相衝突。

```
double power(double x, int n) {
      int i;
      double ans = 1;
      for(i=0;i<n;i++) {
            ans = ans * x;
      }
      return ans;
}
```

```
int doSum(int n) {
      int i;
      int ans = 0;
      for(i=1;i<=n;i++) {
            ans = ans + i;
      }
      return ans;
}
```

24

使用函式的好處

- 將具有特定功能的敍述組合獨立為函式，可提高程式的可讀性。
- 將程式模組化，讓程式可重複使用 (code reuse)，可提升程式寫作的效率。
- 將程式分解為多個獨立函式，當有錯誤時，可較容易找出問題在那一個函式，提高除錯的效率。
- 各函式間互相獨立,可由不同程式設計人員完成。

25

11-6.cpp

```
#include <iostream>
using namespace std;
bool isTriangle(double, double, double);
int main() {
    double a, b, c;
    cout << "請輸入三邊長: ";
    cin >> a >> b >> c;
    cout << a << ", " << b << ", " << c;
    if( isTriangle(a, b, c) )
        cout << " 可";
    else
        cout << " 不可";
    cout << "構成一個三角形。";

    return 0;
}
```

26

11-6.cpp (continuted)

```
bool isTriangle(double a, double b, double c) {
    if(c >= (a+b)) return false;
    if(a >= (b+c)) return false;
    if(b >= (a+c)) return false;
    if((a-b) >= c) return false;
    if((b-c) >= a) return false;
    if((a-c) >= b) return false;

    return true;
}
```

雖然函式只能回傳一個值，但可以有多個 return 出現在函式中。

27

函式覆載

Function Overloading

28

函式覆載
Function overloading

- 函式的介面 (Interface) 由三部份組成
  - 傳回值型別 函式名稱 (參數 1 型別 參數1名稱, 參數 2 型別參數3名稱, 參數 3 型別 參數3名稱, 參數 4 型別 ...)

函式的原型宣告

```
void nStar(char symbol, int x=20);
```

函式的定義

```
void nStar(char symbol, int x){
  for(int i=0;i<x;i++) {
    cout << symbol;
  }
  cout << endl;
}
```

29

函式覆載
Function overloading

- 在 C 語言與大部份的程式語言，函式名稱不可以一樣。
  - double **sum**(const int N, double A);
  - × int **sum**(const int N, int A, int B);
- 在 C++ 中，函式名稱可以重複，只要**函式名稱**與**參數個數及型別**的組合唯一即可。
  - ✓ double **sum**(**const int N, double A**);
  - ✓ int **sum**(**const int N, int A**);
  - ✓ int **sum**(**int N, int A, int B, int C, int D**);
- 不可以使用回傳資料的型別來區分不同的函式。

30

11-7.cpp (1/2)

此例中， area 函式被即一被覆載函式 (function overloading)，有三個不同版本的 area 函式，使用不同的參數個數來選擇不同的版本。

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

double area(double r) {
  return r * r * 3.1415926535897932385;
}
double area(double width, double height) {
  return width * height;
}
double area(double up, double down, double height) {
  return (up + down) * height * 0.5;
}
```

31

11-7.cpp (2/2)

在此範例中，有三個同名的函式 area，但它們的參數個數不一樣，所以編譯器不會弄不清楚應該執行那一個函式。

```
int main() {
  cout << "r=5 的圓面積為" << area(5) << endl;
  cout << "10x20 的矩形面積為" <<area(10, 20)<< endl;
  cout << "上底2, 下底4, 高為3 的梯形面積為" <<
          area(2, 4, 3) << endl;

  return 0;
}
```

程式輸出：
r=5 的圓面積為78.5398
10x20 的矩形面積為200
上底2, 下底4, 高為3 的梯形面積為9

32

函式覆載
Function overloading

- **由前例可以看出，在 C++ 裡函式名稱可以不是唯一。其限制：**
  - 函式名稱不同時，沒有問題。
  - 函式名稱相同時，**沒有預設值**的參數個數不同時，沒有問題。
  - 函式名稱相同、參數個數相同、任一參數或以上之型別不同時，沒有問題。
  - 函式名稱相同、參數個數相同、且型別皆相同時則不行。
  - 亦即：函式名稱與傳入參數個數與到別之組合必需唯一。
  - **總而言之，要讓編譯器不會無法決定應該要呼叫那一個函式！**
- **注意，如果不同函式之間僅有回傳參數之型別不同時，不可共存於同一個程式中。編譯器無法僅靠回傳形別區分不同函式。**
  - `void myFunc(int, int, int);`
  - `int myFunc(int, int, int);`

33

範例

```
void myFunc();
void myFunc(int);
void myFunc(int, int, int);
void myFunc(double, int, int);
void myFunc(double, double, double);
int myFunc(float, double, int);
void myFunc(int, int, int, int);
void myFunc(double, double, double, double);

int myFunc();
double myFunc(double, double, double);
```

此兩個函式原型宣告不合法，會和前面的衝突。

34

11-8.cpp

此例中，編譯器無法判斷是要呼叫
`area(5.0)`
或是
`area(5.0, 1.0)`

```
#include <iostream>
using namespace std;
double area(double r) {
  return r * r * 3.1415926535897932385;
}
double area(double width, double height=1.0) {
  return width * height;
}
int main() {
  cout << "area(5) = " << area(5) << endl;
  cout << "area(10, 20) = " << area(10, 20) << endl;
  return 0;
}
```

35

pow 函式

- 在標準函式庫中的 pow 函式，亦為一個多重覆載的函式，它有以下的原型宣告：

```
long double pow ( long double base, long double exponent );
long double pow ( long double base, int exponent );
double pow ( double base, double exponent );
double pow ( double base, int exponent );
float pow ( float base, float exponent );
```

```
pow(3.0, 4.0)
pow(3.0, 4)
pow(3, 4.0)
pow(3, 4)
pow(3.0f, 4)
pow(3.0f, 4.0f)
```

此函式呼叫會有問題，會無法判斷要型別轉換為
a) pow(double, double)
b) pow(double, int)
c) pow(float, float)

36

05/04 隨堂練習

5.4.2012 隨堂練習

- 請寫一程式，讓使用者輸入 1) 有幾顆骰子 k (<=15顆)、2) 總共丟幾次 n 。並模擬丟完 n 次後，輸出各點數出次的次數以及機率。
  - 請撰寫一函式 nDice,傳入骰子顆數 n，並以亂數模擬並回傳一次丟 n 顆骰子所出現的值。 (e.g. n=1 時，此函數可能回傳 1, 2, 3, … 6; 當n=3，此函數可能回傳 3, 4, 5, … 18)
  - 利用陣列來記錄不同的點數的出現次數
  - 下頁有參考輸出範例
  - 注意到各點數的出現機率會近似於常態分佈。

```
請輸入有幾個骰子: 3
請輸入要丟幾次: 2000000
     3       9202       0.46%
     4      27423       1.37%
     5      55640       2.78%
     6      93170       4.66%
     7     138331       6.92%
     8     194218       9.71%
     9     231797      11.59%
    10     250816      12.54%
    11     249790      12.49%
    12     231026      11.55%
    13     194786       9.74%
    14     138625       6.93%
    15      92630       4.63%
    16      55476       2.77%
    17      27816       1.39%
    18       9254       0.46%
```

```
請輸入有幾個骰子: 5
請輸入要丟幾次: 2000000
     5        249       0.01%
     6       1258       0.06%
     7       3867       0.19%
     8       8942       0.45%
     9      17815       0.89%
    10      32604       1.63%
    11      52667       2.63%
    12      78611       3.93%
    13     108309       5.42%
    14     138716       6.94%
    15     167338       8.37%
    16     189102       9.46%
    17     200663      10.03%
    18     200368      10.02%
    19     189160       9.46%
    20     167743       8.39%
    21     138918       6.95%
    22     108159       5.41%
    23      78288       3.91%
    24      52653       2.63%
    25      32280       1.61%
    26      17804       0.89%
    27       9052       0.45%
    28       3818       0.19%
    29       1383       0.07%
    30        233       0.01%
```