1

## Two weeks ago...

- 標準函式庫函式
  - cmath
  - cstdlib
  - cstring
  - iostream
  - iomanip
  - string
- 使用者自訂函式
  - 函式的原型宣告
  - 函式的呼叫
  - 函式的定義

```
bool isPrime(unsigned long p);
```

```
if(isPrime(7)) {
    cout << "7是質數\n";
}
```

```
bool isPrime(unsigned long p) {
    if(p==2) return true;
    if(p%2==0) return false;
    for(unsigned long i=3;i<p;i+=2) {
      if(p%i==0) return false;
    }
    return true;
}
```

2

## 9-11.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)

至於個別參數的名字不是很重要，可省略。但寫的好處是別人看到變數的名稱就大概知道要傳什麼資料進去。

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

3

## 9-11.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;
}
```

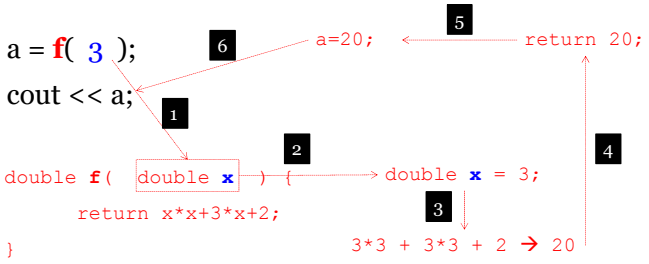
此行為函式的界面 (interface)
- 編譯器會檢查與出現過的原型宣告是否一致
- 所有資料的「進出」都應透過此界面
- 注意到參數應給一變數名稱，以供函式內執行時使用這個變數的名字可以和原型宣告時使用的名字不同。

此處為函式本體 (function body)
- 定義實質函式的運作 (由傳入的參數到傳出的結果之流程)

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。

4

## 函式呼叫時所發生的事情:



5

# 自訂函式

- 自訂函式
  - 原型宣告 (prototype declaration)
  - 函式定義 (function definition)
  - 函式呼叫 (function invocation)

```
int sum(int);
```

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

```
cout << sum(100);
```

6

# 原型宣告
## prototype declaration

- 原型宣告的目的在於告知編譯器一個函式的：
  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 (被呼叫) 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。

7

# 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);
```

8

# 函式定義 - overview
## function definition / implementation

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 (function body)**
}

```
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;
}
```

9

## 函式定義 – return data type
## function definition / implementation

- 回傳資料型別 (return datatype)
  - 定義了函式回傳的資料型別 (double, 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;
}
```

10

## 函式定義 - function name / function identifier
## function definition / implementation

- 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;
}
```

11

## 函式定義 - arguments
## function definition / implementation

- **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;
}
```

12

## 函式定義 - function body
## function definition / implementation

- **函式本體 (function body)**
  - 定義了函式被呼叫時的執行動作、或是由輸入參數計算成回傳結果中間的流程
  - 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
}
```

13

## 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. 函式定義直接作為原型宣告。
- 函式原型宣告與函式定義應一致 (名稱、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;
}
```

14

# Lecture 10

函式 (II)

函式的參數預設值
函式覆載
遞迴函式

15

## 大綱

- 參數預設值
- 函式覆載
- 遞迴函式
- 變數視野/範疇

16

## 參數預設值

default parameters

17

## 10-1.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

18

## 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;
}
```

19

## 10-2.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, )
```

20

## 函式 (user-defined function)

- 函式可以有預設/內定參數值

```
int APlusB (int A, int B=4);
```

- 函式內宣告的變數為區域變數，不同函式之間的區域變數互相獨立，即使它們同名也沒有關係。

```
int APlusB(int A, int B) {
  return A+B;
}
int AMinusB(int A, int B) {
  return A-B;
}
```

21

## 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;
}
```

22

## 10-3.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;
}
```

23

## 10-3.cpp (continued)

```
bool isTriangle(double p, double q, double r) {
    if(r >= (p+q)) return false;
    if(p >= (q+r)) return false;
    if(q >= (p+r)) 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。

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。
- 也可把函式想成是自己在創造程式語言的指令。
  - e.g. nStar, isTriangle
- 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。
- 以由上而下 Top-Down 的設計方式完成程式
  - 將主要流程想清楚，並撰寫在主程式裡面。
  - 分析每一個流程步驟所需的輸入輸出資料， 並撰寫成函式的原型宣告
  - 每一個流程的實作細節，則撰寫在函式定義裡。

25

10-3.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;
}
```

1. 先想好主要的程式流程
2. 找出可或應獨立出來的函式 (isTriangle)
3. 定義每一函式的介面。(e.g. 函式的原型宣告 – 函式需要的傳入資料與回傳資料)
4. 分別、分工地實現不同的程式單元。(e.g. 函式定義)

```
bool isTriangle(double p, double q, double r) {
    if(r >= (p+q)) return false;
    if(p >= (q+r)) return false;
    if(q >= (p+r)) return false;

    return true;
}
```

此乃所謂的 **top-down design** (由上而下的程式設計方法)

26

# 回顧 - 函式

函式的目的主要主於簡化程式發展的難度、並使多人容易互相合作以發展大型程式

- 將一個大問題分割成為很多的小問題 (較為簡單處理或解決的問題)
- 將個別的小問題逐一解決，或將部份小問題丟出去請別人幫忙解決。
- 待個別問題被解決後，原來的大問題應該就要可以被解決。
- 各個擊破法 (divide and concur)

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＋參數個數＋參數型別**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!
  - × double **sum**(**const int N, int A**);
  - × int **sum**(**const int N, int A**);

31

## 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);`

## 10-4.cpp

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;
}
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

## 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。

33

## 10-5.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 * 3.14159265358979323385;
}
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;
}
```

34

# 遞迴函式

## Recursive Function

35

# 遞迴函式
# Recursive Function

- 何謂遞迴函式?
  - 一個函式呼叫自己，即構成遞迴函式。
    `int fun(int a) { return fun(a-1); }`
  - 有些數學上的定義本來就以遞迴定義。
    等差級數、 等比級數、 Fibonacci 級數、 ...

$a_i = a_{i-1} + b$　　$a_i = b \times a_{i-1}$　　$a_i = a_{i-1} + a_{i-2}$

- 範例：
  - 10-6.cpp: 計算階乘
    - 5! =
  - 10-7.cpp: Fibonacci 數列
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

36

## 10-6.cpp

**也可以這樣實現**

```
int fact(int n) {
  int ans = 1;
  for(;n>0;n--) ans *= n;
  return ans;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

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

int fact(int n) {
  if(n==1) return 1;
  return n * fact(n-1);
}

int main() {
  for(int i=1;i<=10;i++) {
    cout << i << "! = " << fact(i) << endl;
  }
  return 0;
}
```

此為遞迴的終止條件！

此為階乘的遞迴定義！

37

## 10-6.cpp

```
int fact(int n) {
  if(n==1) return 1;
  return n * fact(n-1);
}
```

以 fact(5) 為例

```
fact(5):
  return 5 * fact(4);
```

函式 fact(5) 呼叫的生命週期

```
fact(4):
  return 4 * fact(3);
```

函式 fact(4) 呼叫的生命週期

```
fact(3):
  return 3 * fact(2);
fact(2):
  return 2 * fact(1);
fact(1):
  return 1;
fact(2):
  return 2 * 1;
fact(3):
  return 3 * 2;
fact(4):
  return 4 * 6;
fact(5):
  return 5 * 24;
```

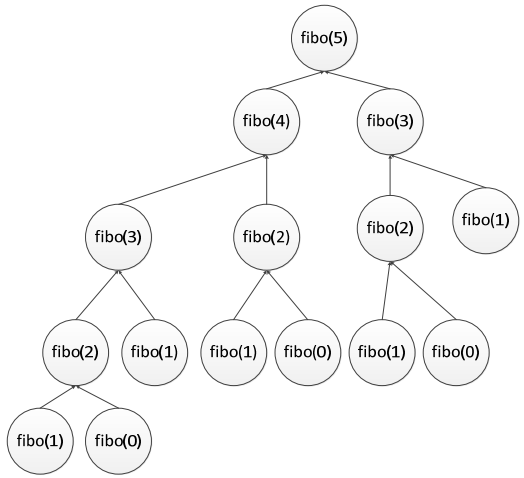
38

## 10-7.cpp

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

unsigned int fibo(unsigned int n) {
  if(n<2) return n;
  else return fibo(n-1) + fibo(n-2);
}

int main() {
  cout << fibo(5) << endl;

  return 0;
}
```

5

39




40

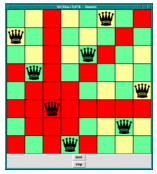
## 撰寫遞迴函式的注意事項

- 遞迴函式必需有「終止」條件，電腦不能無窮遞迴下去!
  - if(n==1) return 1; return ...
  - if(n<2) return n; else return ...
- 遞迴的「深度」不能太深，否則程式執行會發生錯誤
  - stack overflow! (堆疊溢位)
- 遞迴的寫法會讓程式看起來比較簡短，但執行效率不見得比較高；遞迴常常可使用迴圈取代。

41

## 遞迴的應用

- 除了計算各種使用遞迴定義的數列外，遞迴還有許多應用 。
  - 走迷宮 (使用回溯法/back-tracking)
  - 八后問題 (n-queen problem)
  - 樹狀結構的搜尋 (three traversal)
  - 快速排序法 (quick sort)
  - 合併排序法 (merge sort)



42

## 視野 / 範疇

## Scope

43

## 變數視野 / 變數範疇
## scope

- 變數視野／變數範疇 (scope) 指的是一個被宣告出來的變數的可見度 (visibility)，在考慮到變數視野時，主要有以下兩種分類
  - 全域變數 (global variable) – 變數宣告於所有函式之外。全域變數可被其宣告之後的所有函式看到與使用。
  - 區域變數 (local variable) – 變數宣告於函式之內，僅宣告該變數的函式可以看到並使用它。
  - 當區域變數與全域變數同名時，區域變數會遮蔽 (shadowing) 掉同名之全域變數，此時可使用「::」範疇解析運算子/視野解析運算子來暫時存取全域變數。

44

## 10-8.cpp

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

int a;

void func2() {
  int a = 8;
  int b = ::a + 7;
  a++;
  cout<<a<<","<<b<<endl;
}
void func1() {
  int b = a*2;
  a++;
  cout<<a<<","<<b<<endl;
}
```

此 a 為一全域變數

```
int main() {
  int b=5;
  a=4;
  func1();
  func2();
  func1();
  cout<<a<<","<<b<<endl;
  return 0;
}
```

```
5,8
9,12
6,10
6,5
```

45

## 10-8.cpp 說明

- 以上範例中，全域視野裡 (宣告不在任何一個函式中) 有一變數 a。
- fun1() 函式內有宣告一變數 a，此為區域變數 a。編譯器在解析變數時，會以區域變數優先。故 fun1() 中的 a 變數皆為區域變數 a。
- fun2() 中也有宣告一變數 a，故大部份使用變數 a 時，乃使用區域變 a。但是其中的 ::a用來指定要存取的 a 為全域的 a 變數而非區域的 a 變數。其中::稱為 scoping operator (範疇解析運算子)。
- 在 main() 裡，由於 main 裡沒有宣告任何一個叫作 a 的變數。故在 main 裡使用的 a 全部都是使用全域變數 a。

46

## 10-9.cpp

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

void func1() {
  int b = a*2;
  a++;
  cout<<a<<","<<b<<endl;
}

int a;
```

此 a 為一全域變數，僅能被 main 使用，因為只有 main 定義在它之後。所以此程式會發生編譯錯誤。

```
int main() {
  int b=5;
  a=4;
  func1();
  cout<<a<<","<<b<<endl;
  return 0;
}
```

47

## 10-9.cpp 說明

- 在 C/C++ 中，無論是變數宣告或是函式的宣告，都只能在其宣告之後的程式所使用。
- 由於全域變數 a 宣告在 func1 之後，故func1 無法看到全域變數 a ，即使使用 ::a 也無法存取到全域變數 a。

```
Func1() {
    ….
}
Func2() {
    ….
}
```

Func1無法呼叫Func2()，除非Func1之前有 Func2的原型宣告。

```
int main() {
   cout << a << endl;
   int a = 4;
   return 0;
}
```

同樣的，以上程式會發生編譯錯誤，因為在 cout 時 a 還沒有被宣告出來。(除非 main 前面有宣告一全域變數 a。

48

## 隨堂練習

49

## 隨堂練習

- **請「不使用迴圈」完成以下程式**
- 請撰寫一函式 forwardPrint，其傳入一正整數n，並會列印出 1, 2, 3, … n等 n 個數字。
- 請撰寫一函式 backwardPrint，其傳入一正整數n，並會列印出 n, n-1, n-2, …, 3, 2, 1 等 n 個數字。
- 請撰寫主程式，讓使用者輸入一正整數 k，並應用以上兩個函式，正向地從1,2,3列印到k、再反向地由 k, k-1, k-2, 列印到 1。

```
請輸入一正整數 k: 5
1 2 3 4 5
5 4 3 2 1
```

```
請輸入一正整數 k:20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

50

## 作業

- 請撰寫一函式min，傳入三個float的資料，回傳其中最小值。
- 請撰寫一函式max，傳入三個float的資料，回傳其中最大值。
- 請撰寫一函式mid，傳入三個float的資料，利用以上兩函式，回傳三個值中非最小也非最大的值。
- 請撰寫一主程式，讓使用者輸入任意三個數字，並利用以上函式，將使用者所輸入數字由小到大輸出一次、再由大到小輸出一次。

```
請輸入三個數字:3.3 1.1 5.5

由小到大： 1.1, 3.3, 5.5
由大到小： 5.5, 3.3, 1.1
```

```
請輸入三個數字:-1.1 -11.11 -111.111

由小到大： -111.111, -11.11, -1.1
由大到小： -1.1, -11.11, -111.111
```