1

Last time

參數預設值 (default parameters)

```
void test(int a, int b, int c=3, int d=4) {
       cout << a << ", " << b << ", ";
       cout << c << ", " << d << endl;
}


test(10,20);                10, 20, 3, 4
test(100, 200, 300);        100, 200, 300, 4
test(100, 200, 300, 400);   100, 200, 300, 400
```

2

Last time

函式覆載 (function overloading)

```
double area(double r) {
       return r * r * 3.1415926535897932385;
}
double area(double width, double height) {
       return width * height;
}

cout << "r=5 的圓面積為" << area(5) << endl;
cout << "10x20 的矩形面積為" <<area(10, 20)<< endl;
```

3

Last time

遞迴函式 (recursive functions)

```
forwardPrint(int n) {
       if(n>1) forwardPrint(n-1);
       cout << n << " ";
}

backwardPrint(int n) {
       cout << n << " ";
       if(n>1) backwardPrint(n-1);
}

forwardPrint(20);
backwardPrint(20);
```

4

```
forwardPrint(int n) {
  if(n>1) forwardPrint(n-1);
  cout << n << " ";
}

forwardPrint(5) {
  forwardPrint(4)
    forwardPrint(3)
      forwardPrint(2)
        forwardPrint(1)
          cout << 1 << " ";
        cout << 2 << " ";
      cout << 3 << " ";
    cout << 4 << " ";
  cout << 5 << " ";
```

```
backwardPrint(int n) {
  cout << n << " ";
  if(n>1) backwardPrint(n-1);
}


backwardPrint(5) {
  cout << 5 << " ";
  backwardPrint(4);
    cout << 4 << " ";
    backwardPrint(3);
      cout << 3 << " ";
      backwardPrint(2);
        cout << 2 << " ";
        backwardPrint(1);
          cout << 1 << " ";
```

5

Lecture 11

變數範疇
函式呼叫時的資料傳遞 (I) – 傳值
參考型別
函式呼叫時的資料傳遞 (II) – 傳參考
基本檔案操作

6

視野 / 範疇

Scope

7

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。

8

11-1.cpp

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

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

int a;          此 a 為一全域變數

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;
}
```

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

9

11-1.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。

10

11-2.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;
}
```

11

11-2.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。

12

函式呼叫時的資料傳遞 (I)

傳值呼叫 (call by value)

13

函式定義

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

函式定義的介面決定了資料如何**進入**此函式、以及運算結果回傳給呼叫者 (caller)。

一般參數列可以將資料傳入進行運算，也可用來將資料傳出！
(這個範例僅將資料傳入，而沒有資料能從參數列傳出！)

此為函式傳出資料的途徑一，使用 **=** 將回傳的資料存入一指定變數或是把它直接給 cout 列印出來。
e.g. q = sum(200); cout << sum(400);

14

11-3.cpp

```
#include <iostream>
using namespace std;
int test(int b) {
  b = b * 2;
  return b+1;
}
int main() {
  int a = 3;
  int b;
  b = test(a);
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;
  return 0;
}
```

int b = a;

- 在 C/C++ 裡, 參數的傳遞內定的方法是**傳值呼叫 (call by value)**。
- 在 test 函式被呼叫時, 主程式傳入的 a 變數的值被**複製**一份到函式裡區域變數 b ，然後才開始執行 test 函式的內容。
- 所傳入的參數 (e.g. a)，是用來**初始化**在函式裡的區域變數 (b)。
- test 函式結束時將 b+1 的值透過 return 的功能傳回主程式中，並在此範例中存入主程式中的 b 變數，然後test函式中的 b 變數即被消滅。
- 因為是**傳值呼叫**，無論test函式裡的 b 變數如何改變，都不會影響主程式裡 a 變數的值。

a=3
b=7

15

11-3.cpp 說明

以下的 main::、test:: 用來強調該變數所在的區域或是說該變數所在的範疇 (scope)，並不是正確的語法 。

```
#include <iostream>
using namespace std;
int test(int b) {
  b = b * 2;
  return b+1;
}
int main() {
  int a = 3;
  int b;
  b = test(a);
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;
  return 0;
}
```

a
b
_
1
2
3
4
5

int b = a;

1: int **main::**a = 3;
2: int **main::**b;
3:
  _: int **test::**b = **main::**a; (test::b = 3)
  a: **test::**b = **test::**b * 2; (test::b = 6)
  b: return **test::**b + 1; (return 7)
  main::b = 回傳值; (main::b = 7)
4: cout << "a=" << **main::**a << endl;(印出 main::a)
5: cout << "b=" << **main::**b << endl;(印出 main::b)

16

11-4.cpp

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

void swap(int a, int b);

int main() {
  int a = 3;
  int b = 7;

  cout << "a=" << a << ", b=" << b << endl;
  swap(a, b);
  cout << "a=" << a << ", b=" << b << endl;

  return 0;
}
```

17

11-4.cpp (cont'd)

```
void swap(int a, int b) {
    cout << a << ", " << b << endl;
    int tmp = a;
    a = b;
    b = tmp;
    cout << a << ", " << b << endl;
}
```

```
a=3, b=7
3, 7
7, 3
a=3, b=7
```

此範例亦為傳值呼叫 (call by value)，所以雖然在 swap 函式裡成功將傳入的兩個變數交換了，在主程式中的兩個變數並沒有交換。

18

11-4.cpp (cont'd)

以上程式中，真正重要且有執行的敍述與順序有：

```
main:: int main::a = 3;
main:: int main::b = 7;
main:: cout << "a=" << main::a << ", b=" << main::b << endl;
main:: swap(a, b);
  swap:: int swap::a = main::a;
  swap:: int swap::b = main::b;
  swap:: cout << swap::a << ", " << swap::b << endl;
  swap:: int swap::tmp = swap::a;
  swap:: swap::a = swap::b;
  swap:: swap::b = swap::tmp;
  swap:: cout << swap::a << ", " << swap::b << endl;
main:: cout << "a=" << main::a << ", b=" << main::b << endl;
```

19

參考型別

20

11-5.cpp

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

int main() {
  int a = 3;
  int &b = a;

  cout << a << endl;
  cout << b << endl;
  b = 4;
  cout << a << endl;
  return 0;
}
```

變數 b 參考到為變數 a
也可說變數 b 為變數 a 的別名、分身

```
3
3

4
```

21

11-5.cpp 的說明

- 宣告 b 為一參考型別的變數，其參考的變數為 a
- 參考型別在宣告時，「**一定**」要指定參考那一個變數!
  (有點類似在宣告常數變數(const)時，一定要指定常數值)
- 宣告出來的參考型別變數 b, 為原來變數 a 的分身。
- 有時, 亦可說 b 為 a 的別名 (alias)
- 所以，改變參考型別變數 b 的值，被參考到的變數 a 的值也會一起跟著改變。

22

11-6.cpp

```
#include <iostream>
using namespace std;
int main() {
  int a = 3;
  int &b = a;
  int &c = b;
  int d = c;

  cout << a << ", " << b << ", " << c << ", " << d << endl;
  d = 10;
  cout << a << ", " << b << ", " << c << ", " << d << endl;
  c = 5;
  cout << a << ", " << b << ", " << c << ", " << d << endl;

  return 0;
}
```

```
3, 3, 3, 3
3, 3, 3, 10
5, 5, 5, 10
```

23

函式呼叫時的資料傳遞 (II)

傳參考呼叫 (call by reference)

24

11-7.cpp

```
#include <iostream>
using namespace std;
int test(int &b) {
  b = b * 2;
  return b+1;
}
int main() {
  int a = 3;
  int b;
  b = test(a);
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;
  return 0;
}
```

int &b = a;

- 此範例為**傳參考**的函式呼叫。
- 在 test 函式被呼叫時, 主程式傳入的 a 變數成為 b 這個參考型別的變數初始值 ，因此 b 變數在 test 函式中，為主程式中的 a 變數的分身。
- test 函式結束時將 b+1 的值透過 return 的功能傳回主程式中，並在此範例中存入主程式中的 b 變數。
- 因為是傳參考呼叫，所以在 test 函式中對於 b 變數的改變，實際上就是對主程式中的 a 變數作改變。

```
a=6
b=7
```

11-7.cpp 說明

以下的 main::、test:: 用來強調該變數所在的區域或是說該變數所在的範疇 (scope)、: 前面則提示所對應的程式敍述所在位置，並不是正確的語法。

```
#include <iostream>
using namespace std;
int test(int &b) {
a   b = b * 2;
b   return b+1;
}
_   int &b = a;
int main() {
1   int a = 3;
2   int b;
3   b = test(a);
4   cout << "a=" << a << endl;
5   cout << "b=" << b << endl;
    return 0;
}
```

1: int main::a = 3;
2: int main::b;
3:
_: int test::&b = main::a; (test::b = 3)
a: test::b = test::b * 2; (test::b = 6)
b: return test::b + 1; (return 7)
main::b = retval; (main::b = 7)
4: cout << "a=" << main::a << endl;(印出 main::a)
5: cout << "b=" << main::b << endl;(印出 main::b)

參數傳遞的方式有三種

- **傳值 (call by value)**
  - 此為預設方式, 將傳入的變數值**拷貝**一份到函式內的區域變數 (local variable).
- **傳參考 (call by reference)**
  - 使用參考型別的變數, 將函式的區域參考變數參考到呼叫者的變數
- ***傳址 (call by address)***
  - *使用指標型別的變數, 將變數的記憶體位址拷貝一份到函式內的區域指標變數*

11-8.cpp

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

void swap(int &a, int &b);

int main() {
  int a = 3;
  int b = 7;

  cout << "a=" << a << ", b=" << b << endl;
  swap(a, b);
  cout << "a=" << a << ", b=" << b << endl;

  return 0;
}
```


11-8.cpp (cont'd)

```
void swap(int &a, int &b) {
    cout << a << ", " << b << endl;
    int tmp = a;
    a = b;
    b = tmp;
    cout << a << ", " << b << endl;
}
```

a=3, b=7
3, 7
7, 3
a=7, b=3

此範例亦為傳參考呼叫 (call by reference)，所以在 swap 函式裡成功將傳入的兩個變數交換了，在主程式中的兩個變數也成功地交換，因為函式裡的 a, b 為主程式中 a, b 的分身。

29

11-8.cpp (cont'd)

以上程式中，真正重要且有執行的敍述與順序有：

```
main:: int main::a = 3;
main:: int main::b = 7;
main:: cout << "a=" << main::a << ", b=" << main::b << endl;
main:: swap(a, b);
  swap:: int swap::&a = main::a;
  swap:: int swap::&b = main::b;
  swap:: cout << swap::a << ", " << swap::b << endl;
  swap:: int swap::tmp = swap::a;
  swap:: swap::a = swap::b;
  swap:: swap::b = swap::tmp;
  swap:: cout << swap::a << ", " << swap::b << endl;

main:: cout << "a=" << main::a << ", b=" << main::b << endl;
```

30

11-9.cpp

A. 看起來，area 不管值是多少，函式呼叫完成後都會是圓面積。
B. 看起來，不管 perimeter 是多少，函式呼叫完成後都會是圓周長。
C. 所以，它們都應該使用傳參考呼叫。
D. 看起來，circle函式沒有回傳值。

```
int main() {
  double r, area, perimeter;
  cout << "請輸入半徑:";
  cin >> r;

  circle(r, area, perimeter); // 如何寫這個函式？

  cout << "面積 = " << area << endl;
  cout << "圓周 = " << perimeter << endl;

  return 0;
}
```

31

先想原型宣告

A. 看起來，area 不管值是多少，函式呼叫完成後都會是圓面積。
B. 看起來，不管 perimeter 是多少，函式呼叫完成後都會是圓周長。
C. 所以，它們都應該使用傳參考呼叫。
D. 看起來，circle函式沒有回傳值。

```
double r, area, perimeter;
circle(r, area, perimeter);  // 如何寫這個函式？
```

回傳值型別
函式名稱(參數1宣告, 參數2宣告, …)

宣告 = 變數型別 變數名稱
e.g. int a;

```
void
circle(
      double r,
      double &area,
      double &perimeter
);
```

32

11-9.cpp (cont'd)

```
void circle(double r, double &area, double &perimeter)
{
    const double pi = 3.141592654;
    area = pi*r*r;
    perimeter = 2*pi*r;
}
```

33

小結

- 函式呼叫 僅使用傳值法 (call by value/copy，C/C++ 的內定方法) 時，我們只能傳回一個運算結果 (透過 return)。
  - e.g.

```
double f(double x, double y) {
   x+=3; y+=2;
   return x*x+y*y;
}
…
double x=0, y=0;
double z = f(x, y);
cout << x << “, “ << y << “, “ << z;  // 0, 0, 13
```

- 傳值呼叫**不可能**意外改到呼叫者 (e.g. main) 內的變數值。

34

小結

- 傳參考的函式呼叫 (call by reference) 時，乃將呼叫者的變數 (e.g. a)，在被呼叫函式中創造出一個別名 (也許相同)，所以在函式裡改變變數值，則呼叫者的變數也會被改變。
- 傳參考呼叫方法則可能可以傳回一個以上的運算結果，並可以改變呼叫者內的變數值。

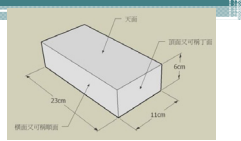
e.g.

```
double f(double &x, double y) {
   x+=3; y+=2;
   return x*x+y*y;
}
…
double x=0, y=0;
double z = f(x, y);
cout << x << “, “ << y << “, “ << z;  // 3, 0, 13
```

- 傳值呼叫，該參數為函式的輸入；傳參考呼叫，該參數可能為輸入值也可能為輸出值。11-8.cpp 中的 swap 函式 a b 兩變數是傳入值、也是傳出值。11-9.cpp中的circle，r為傳入值、area、perimeter為傳出值。

35

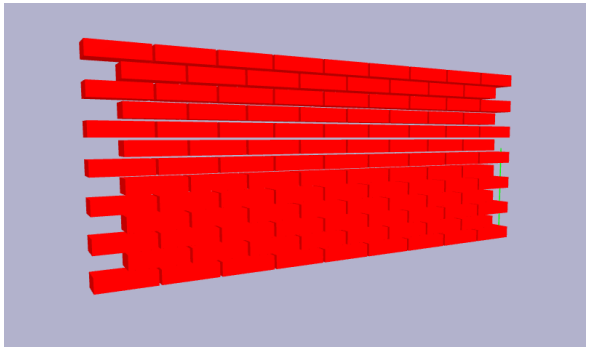
5.19隨堂練習



- 本隨堂練習會利用到第五次上課的簡單3D繪圖功能 ...
- 請撰寫一函式 wallType1，其原型宣告如下：
  int wallType1(creator& world, int& width, int& height);
  此函式會以「順式砌法」蓋/繪出磚牆、回傳總共用了多少磚塊、並將width與height設定為最終的牆寬與牆高。
- 在主程式中，讓使用者輸入欲蓋磚牆的牆寬、牆高，並呼叫以上函式 wallType1，傳入寬度與高度，繪出磚牆，並印出最終的牆寬、牆高、以及使用的磚塊數。
- p.s. 磚塊與磚塊隨會有10mm的灰縫

http://sanchien.blogspot.tw/2015/08/blog-post.html
http://www.artbrick.com.tw/about_05.htm

36



作業十

- 延伸隨堂練習，以自己學號的末兩碼除以5的餘數+2決定你要砌的磚牆種類：
  - 2: 美式砌法
  - 3: 法式砌法
  - 4: 英式砌法
  - 5: 丁砌
  - 6: 荷式砌法
- 撰寫 wallType2,wallType3, wallType4, wallType5或是wallType6 中的一個函式，繪出以該砌法的磚牆，並回傳使用磚頭數目，並設定最終牆的寬度與高度。該函式可能需要額外傳入參數以提供不同砌磚方式的參數。
- 在主程式中，讓使用者輸入欲蓋磚牆的牆寬、牆高，並呼叫你所撰寫的函式 wallType？，傳入寬度與高度與其它需要的參數，繪出磚牆，並印出最終的牆寬、牆高、以及使用的磚塊數。

http://sanchien.blogspot.tw/2015/08/blog-post.html