1

Lecture 14

變數視野
呼叫函式時的資料傳遞路徑
檔案操作

2

回顧

- 變數的生命週期 / 儲存等級
  - 自動、靜態、動態
- 動態記憶體管理
  - 動態配置變數

```
int *a = new int(3); *a = 7; delete a;
double *b = new double(3.14); cout << *b; delete b;
```

  - 動態配置陣列

```
int *c = new int[20];
for(int i=0;i<20;i++) c[i] = i;
delete []c;
```

  - 檢查配置是否成功

```
int *e = new (nothrow) int[50];
if(e==0) cout << "失敗";
```

3

回顧

- 觀念回顧
  - 一般變數 ➔ 一個名字，用來儲存與運算資料，為獨立的個體。
  - 參考型別變數➔ 一個一般變數的分身、別名，不為獨立的個體 (不具有自己獨立的記憶體空間)。➔主要用來在不同函式之間傳遞並修改資料 (傳參考)。
  - 指標型別變數 ➔ 特別的變數，用來記錄一個地址。此地址可以指到一般變數的位置 (住別人的房子)、可以指向一動態配置出來的地址 (租自己的房子)。可用來在不同函式之間傳遞變數、陣列。

4

上週隨堂練習

- 主程式已貼在 BB 的 5/27 隨堂練習，請完成以下兩個函式的定義：

```
void showData(int n, int *data);
int* getEven(int &nEven, int n, int *data);
```

- 其中，showData 會將傳入 data 指標位置上的前 n 個整數印出來。
  - int a[] = {5,4,3,2,1};
  - showData(5, a); // 會印出 5 4 3 2 1 後換行
- 而 getEven 比較複雜，它會將傳入 data 指標位置上的前 n 個整數中所有的偶數找出來存入一新的大小剛剛好的陣列裡 (動態配置出來的記憶體位置)，並將偶數個數放在 nEven 變數裡回傳回來。
  - int nEven;
  - int *evenData = getEven(nEven, 5, a);
  - showData(nEven, evenData); // 會印出 4 2 後換行

5

```
.
.
.

  cout << "\n請輸入你要輸入的整數資料個數: ";
  cin >> n;
  data = new int [n];
  for(int i=0;i<n;i++) {
    cout << "請輸入第 " << i+1 << " 個資料: ";
    cin >> data[i];
  }

  cout << "\n你輸入的資料為: ";
  showData(n, data);
  evenData = getEven(nEven, n, data);

  cout << "\n其中共有 " << nEven << " 個偶數。他們為: ";
  showData(nEven, evenData);

  delete []data; delete []evenData;
  return 0;
}
```

6

今日內容

- 變數視野
  - 區域變數、全域變數
- 函式之間的資料傳遞途徑
  - 參數列 – 傳值、傳址、傳參考
  - 回傳值 – 傳值、傳址、傳參考
  - 變數的保護 - const
  - 全域變數
- 檔案操作

7

視野

Scopes

8

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。

9

14-1.cpp

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

int a;

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

此 a 為一全域變數

```
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;
}
```

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

10

14-1 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 (視野運算子)。
- 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。
- 全域變數的儲存等級為靜態的 (static variable)，亦即其在程式開始執行時即被創造出來，直至程式結束時才被消滅掉。若沒有給初始值時會被賦予初始值 0 。

11

14-2.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;
}
```

12

14-2 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。
- 在 14-2 中，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。

13

函式呼叫時的資料傳遞途徑

14

函式呼叫時的參數列資料傳遞

- 呼叫函式時，可以傳值、傳址、或傳參考的方式，透過參數列將變數資料由呼叫者傳遞給被呼叫的函式，供函式運算或回傳資料。
  - 傳值的原型宣告範例：void func1(int a, double b, float c);
  - 傳址的原型宣告範例：void func2(int *a, double *b, float *c);
  - 傳參考的原型宣告範例： void func3(int &a, double &b, double &c);
- 用途：
  - 傳值：參數列的資料只進不出
  - 傳址：參數列的資料可進可出，較適合用來傳遞陣列資料
  - 傳參考：參數列的資料可進可出，較適合用來傳遞一般非陣列資料

15

14-3.cpp 參數列傳值呼叫

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

int max(int a, int b) {
  if(a>b) return a;
  return b;
  //在此函式中無論怎麼改 a 或 b 的值,都不會影響主程式中的 c 或是 d
}

int main() {
  int c=5, d=7, q;
  q = max(c, d);
  cout << "q=" << q << ", c=" << c << ", d=" << d << endl;
  return 0;
}
```

此時，若函式有運算結果要傳出來，必需透過 return 的方式回傳。

16

14-3.cpp 說明

- 參數列可視為函式內的區域變數宣告，在函式被呼叫時會將變數生出來，待函式結束時被消滅。
- 以 14-3.cpp 為例，當 main 呼叫 max 時，參數列可視為宣告變數 a、b並分別給予初始值 (根據位置)
  - int a = c;
  - int b = d;

17

14-4.cpp 參數列傳參考呼叫

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

int max(int &a, int &b) {
  a++;
  b++;
  if(a>b) return a;
  return b;
}

int main() {
  int c=5, d=7, q;
  q = max(c, d);
  cout << "q=" << q << ", c=" << c << ", d=" << d << endl;
  return 0;
}
```

q=8, c=6, d=8

18

14-4.cpp 說明

- 參數列可視為函式內的區域變數宣告，在函式被呼叫時會將變數生出來，待函式結束時被消滅。
- 以 14-4.cpp 為例，當 main 呼叫 max 時，參數列可視為宣告變數 a、b並分別給予初始值 (根據位置)
  - int &a = c;
  - int &b = d;
- 因此，a 成為主程式中 c 的分身、b 成為主程式中 d 的別名，因此在 max 函式中對 a, b 作修改，即對主程式中的 c, d 作修改。
- 透過傳參考，一個函式可以回傳多個運算結果。

19

14-5.cpp 參數列傳址呼叫

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

int max(int *a, int *b) {
  (*a)++;    // 等同於 a[0]++;
  b[0]++;    // 等同於 (*b)++;
  if(a[0]>*b) return *a;
  return *b;
}

int main() {
  int c=5, d=7, q;
  q = max(&c, &d);
  cout << "q=" << q << ", c=" << c << ", d=" << d << endl;
  return 0;
}
```

q=8, c=6, d=8

20

14-5.cpp 說明

- 參數列可視為函式內的區域變數宣告，在函式被呼叫時會將變數生出來，待函式結束時被消滅。
- 以 14-5.cpp 為例，當 main 呼叫 max 時，參數列可視為宣告變數 a、b並分別給予初始值 (根據位置)
  - int *a = &c;
  - int *b = &d;
- 此時，指標 a 記得的是 c 變數住的位址、指標 b 記得的是 d 變數住的位址。因此可透過取值運算子 (dereference) 對主程式中的 c、d 修改、亦可將 a、b 視為僅有一個元素的陣列作存取。
- 透過傳址，一個函式可以回傳多個運算結果。
- 可以注意到 14-4.cpp 比 14-5.cpp 易讀，故我個人建議在傳一般非陣列變數時，使用傳參考的方式。傳址比較適合用來傳陣列。

21

14-6.cpp (1/2)

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

int max(int n, int *array);
int main() {
  int a[] = {5,4,3,2,1};
  int b[] = {1,2,3,4,5,6,7,8};
  int c[] = {10,20,30};
  cout << max(5, a) << endl;
  cout << max(8, b) << endl;
  cout << max(3, c) << endl;

  return 0;
}
```

```
5
8
30
```

22

14-6.cpp (2/2)

```
int max(int n, int *array) {
  int ans = array[0];
  for(int i=1;i<n;i++) {
      if(array[i] > ans) ans = array[i];
  }
  return ans;
}
```

23

14-6.cpp 說明

- 在此範例中，傳遞一陣列至 max 函式找尋陣列中最大的元素。
- 雖然說是傳遞陣列，其實我們傳的是陣列中的第一個元素的記憶體位置 (指標指向陣列的第一個元素)。我們也必需將陣列中的元素個數傳至函式中，不然函式裡不知道陣列有幾個元素、或是此函式將只能應用於某一固定大小的陣列。
- 此範例中，原型宣告或是函式定義的介面，亦可採以下語法：
  - `int max(int n, int *array);` // 指標
  - `int max(int n, int array[]);` // 陣列
  - `int max(int n, const int *array);` // 唯讀指標
  - `int max(int n, const int array[]);` // 唯讀陣列
- 使用唯讀指標，可使函式只能讀出陣列的內容而無法對其作修改，可達到保護陣列資料的目的，確保其不會被函式內的程式所修改。

24

函式回傳值

- 函式呼叫時，參數列可將參數以傳值、傳址、傳參考的方式傳入函式內。
- 函式在回傳運算結果時，亦可回傳值、回傳位址、回傳參考。
  - 回傳值：用來回傳一個運算結果，類似於數學函數。
  - 回傳址：用來回傳一個指標，此指標需指向合法的記憶體位址 (動態記憶體配置、傳回陣列中的某個元素、 ...) (如上次的隨堂練習)
  - 回傳參考：回傳一個合法資料的參考 (e.g. 靜態變數、傳入陣列中的某個元素 ...)
- 注意回傳址或參考時，不可參照到區域變數 (不回傳區域變數的位址、不回傳區域變數的參考)

```
int max(int n, int *array) {
  int ans = array[0];
  for(int i=1;i<n;i++) {
    if(array[i] > ans) ans = array[i];
  }
  return ans;
}
```

25

14-7.cpp

```
#include <iostream>
using namespace std;
int *even(int *it) {
  do {
    ++it;
  } while(it[0]!=0 && it[0]%2!=0);
  return it;
}
int main() {
  int a[] = {1,2,3,4,5,6,0};
  int *it = a;
  do {
    it = even(it);
    cout << *it << " ";
  } while(*it != 0);
  return 0;
}
```

此函式傳入一指標，並回傳下一個出現偶數或是 0 的指標 ...

```
2 4 6 0
```

26

錯誤範例

```
int *fun(int a) {
  a *= 2;
  return &a;
}
```

```
int &fun(int a) {
  a *= 2;
  return a;
}
```

在此會錯誤的原因是，此兩範例的 a 皆為儲存等級為自動的區域變數，a 只有在此函式在執行時才擁有一記憶空間儲存資料，等函式結束後即被消滅。故回傳之 a 的位置或參考皆無法正常使用。
語法正確、可編譯無誤，但執行會錯！

正確範例

```
int fun(int a) {
  a *= 2;
  return a;
}
```

```
int &fun(int &a) {
  int b=0;
  a *= 2;
  return a;
}
```

```
int *fun(int a) {
  static int b=0;
  a *= 2;
  return &b;
}
```

27

非正式傳遞路徑 – 全域變數

- 全域變數可被所有定義在它「下面」的函式所使用。
- 缺點如果全域變數的值不正確，我們無法很快找到那一個函式對全域變數作了修改。
- 所以，雖然可以使用全域變數讓不同函式之間共享一個資料，僅量不要！

28

14-8.cpp

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

int a;

void fun1() {
    a = 3;
    cout << a << " ";
}
void fun2() {
    a *= 2;
    cout << a << " ";
}
void fun3() {
    a += 10;
    cout << a << " ";
}
```

```
int main() {
    fun1();
    fun3();
    fun2();

    return 0;
}
```

```
3 13 26
```

29

檔案操作

30

檔案輸入與輸出
file I/O (input/output)

- 為什麼需要電腦檔案？
  - 所有我們目前學到的，都是在利用記憶體 (變數) 來進行運算，而運算的資料來源為：
    - 1) 內建於程式內的變數 (data = 3;)
    - 2) 由使用者輸入 (cin >> data);
  - 運算得到的結果常常是輸出至螢幕上
    - cout << result;
- 問題
  - 運算結果無法長期保存下來、無法檢查
  - 使用者會忘記要輸入什麼資料
  - 重複輸入資料很煩、會出錯。
- 在 C/C++ 來說，電腦檔案是一種串流 (stream) 或資料流。

31

檔案串流
file stream

- 檔案串流，即為其來源或目的為電腦檔案
  - cin: 來源為鍵盤輸入的串流物件
  - cout: 來源為螢幕輸出的串流物件
  - 檔案串流需有相關的檔案名稱！
- 檔案輸入輸出串流
  - **ifstream** (input file stream)
  - **ofstream** (output file stream)
  - **fstream**(file stream) – 可設定作輸入、輸出的串流

32

14-9.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ofstream file;
  file.open("c:\\test.dat");
  if(!file) {
      cout << "檔案開啟錯誤!";
      return 255;
  }
  file << "Hello File!";
  file.close();
  return 0;
}
```

含括入檔案串流的相關原型宣告

宣告 file 為一**輸出**檔案串流物件
開啟 c:\test.dat
檢查檔案串流開啟狀態，一個好的程式應該要檢查檔案是否有開啟成功，以免後面的程式出錯。

將 “Hello File!” 插入至檔案串流
關閉檔案串流

33

檔案串流物件

- cin, cout, ofstream, ifstream 都是一種串流物件
- 之前 cin, cout 的操作 <<, >> 都可直接套用在檔案操作上。
  - `file << "Hello File!";`
  - `file << "\nA=" << A << ", " << "B=" << B;`
  - << 和>>稱為串流的插入與抽取運算子
- 檔案串流只增加了
  - #include <**fstream**>
  - ifstream abc; 或是 ofstream abc; ← 用來產生輸入/輸出串流物件
  - abc.open(“c:\\test.dat”); ← 將串流物件關連於一個電腦檔案
  - abc.close(); ← 關閉檔案
  - 其中 abc 為任意合法的變數名稱

34

14-10.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ifstream file;
  char a;
  file.open("c:\\test.dat");
  if(!file) {
      cout << "檔案開啟錯誤!";
      return 255;
  }
  while(file.good()) {
      file >> a;
      cout << a;
  }
  file.close();
  return 0;
}
```

宣告 file 為一輸入檔案串流物件

嘗試開啟 c:\test.dat
檢查檔案開啟是否成功
失敗時輸出一訊息
結束程式執行，回傳一非零值乃一習慣以代表程式異常結束
當檔案串流的「狀態」良好時作：
  由檔案讀入一個字元
  透過 cout 輸出至螢幕上。

檔案關閉
程式正常結束

35

串流的狀態
stream state

- 串流有以下的狀態，通常是經過輸入或輸出的操作後會被設定。
  - **eof** (end of file): 串流已至檔案結尾，表所有資料都已經流過了，不會再有更多資料進來了。
  - **fail**: 失敗，表示有輸入或操作的動作失敗。例如：
    - cin >> a; (其中 a 為整數，但輸入的資料為 “$AC”，無法轉換)
  - **bad**: 壞掉，表串流本身之完整性 (integrity) 有問題，通常代表即使將狀態清除後換一種操作仍然會發生問題。
- 可透以下相關函式檢查或設定
  - **clear**(): 將以上三種不好的狀態清除。
  - **good**(): 回傳 true 表示以上三種狀態都沒有發生, 可以進行操作。
  - **bad**(): 回傳 true 時表示 bad 的狀態發生了。
  - **fail**(): 回傳 true 時表示 bad 或 fail 的狀態發生了。

36

14-11.cpp

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

int main() {
  int a;

  // 插入 A)
  // 插入 B)

  cout << "請輸入 a:";
  cin >> a;
  cout << "a=" << a << endl;

  // 插入 C)
  return 0;
}
```

A)
```
ifstream cin("e:\\test.dat");
if(!cin) {
  cout << "e:\\test.dat 開啟失敗";
  return 255;
}
```

B)
```
ofstream cout("e:\\test1.dat");
if(!cout) {
  cout << "e:\\test1.dat 開啟失敗";
  return 255;
}
```

C)
```
cin.close();
cout.close();
```

37

14-11.cpp 的說明

- cin, cout 為 C++ 幫我們宣告好的全域變數！
- 所以，我們可以透過遮蔽的方式，利用區域變數來把 cin、cout 變成檔案串流，所以可以很容易地把原來輸出在螢幕上或是由鍵盤輸入的方式，轉變成使用檔案作輸入輸出。
- 三步驟
  - 開啟檔案
  - 檢查是否開啟成功
  - 進行串流輸入輸出
  - 關開檔案串流
- 以下兩範例程式說明透過 iomanip 對輸出進行美化，僅供參考！

38

格式化的輸入輸出

- 格式化輸入輸出 (formatted input/output)
  - <<: 將右手邊變數的值經過**轉換**成為文字後插入左手邊之串流
  - >>: 將由串流輸入的資料以空格 (white space) 或換行將輸入的文字資料分割開來**轉換**成為適當格式後存入右手邊的變數。
- 若欲使輸出的檔案更加美觀 (對人來說!) ➔ 使用串流操控器 iomanip
  - #include <iomanip>
  - 使用 << 插入格式控制

39

串流操控器	效　果
setw(n)	將欄位寬度設定為 n。
setfill('*')	設定欄位內空白處填入的字元。
left	在欄位內靠左對齊。
right	在欄位內靠右對齊。

	串流操控器	效　果	範例
int	showpos	永遠顯示正負號，即正數之前加上+號。	+1
	noshowpos	不強制顯示正號。（預設值）	1
	dec	以10進位法表示。（預設值）	17
	oct	以8進位法表示。	21
	hex	以16進位法表示。	11
	noshowbase	不顯示數字的基底格式（預設值）	11
	showbase	顯示數字的基底格式（預設值）	0x11

40

14-12.cpp

```
const int i=100;

cout << "|12345678901234567890" << endl;
cout << "|" << setw(10) << i << "|" << endl;
cout << "|" << showpos  << i << "|" << endl;
cout << "|" << setw(5)  << hex << i << "|" << endl;
cout << "|" << setw(10) << showbase << i << "|" << endl;
cout << "|" << dec << left << i << "|" << endl;
cout << right;
cout << "|" << setw(7) << noshowpos << i << "|" << endl;
cout << "|" << setw(8) << oct << i << "|" << endl;
```

41

14-12.cpp 執行結果

```
|12345678901234567890
|       100|
|+100|
|   64|
|      0x64|
|+100|
|   _100|
|    0144|
```

42

	串流操控器	效　果	範例
float, double	setprecision(n)	將浮點精度設為 n 位數。(內定為6)	12.300000
	fixed	使用一般的浮點數表示法。	
	scientific	使用科學記號表示法。	1.230000e+001
	showpoint	永遠顯示小數點	1.0
	noshowpoint	不強制顯示小數點（預設值）	1
	showpos	永遠顯示正負號，即正數之前加上+號。	+1
	noshowpos	不強制顯示正號	1

	串流操控器	效　果	範例
bool	boolalpha	以true、false字串表示bool值。	true
	noboolalpha	以1、0表示bool值（預設值）。	1

43

14-13.cpp

```
double pi=1;

cout << showpoint << pi << endl;
cout << noshowpoint  << pi << endl;

pi = 314.159265358979323846;
cout << pi << endl;
cout << scientific << pi << endl;
cout << fixed << pi << endl;
cout << setprecision(10);
cout << setw(6) << scientific << pi << endl;
cout << setw(6) << fixed << pi << endl;
```

44

14-13.cpp 執行結果

```
1.00000
1
314.159
3.141593e+002
314.159265
3.1415926536e+002
314.1592653590
```

45

隨堂練習

46

隨堂練習

- 假設一物體之初始位置為 (0, 0)，請寫一程式：
  - 讓使用者輸入一物體之水平與垂直方向的初始速度
  - 輸出每隔 0.1 移該物體的時與與位置 (t, x, y並以逗號隔開)，直到物體的 y 小於 0 為止。
  - 並將計算結果輸出至「 fly.csv 」檔案內。
- 找到輸出之「fly.csv」 檔案，點兩下後可由 Excel 自動開啟。請利用 x y 散佈圖的方式畫出該物體的飛行軌跡。

47

參考輸出

$$x = x_0 + v_x \times t$$

$$y = y_0 + v_y \times t - \frac{g}{2} \times t^2$$

```
請輸入初速度 Vx 與 Vy: 5 5
0, 0, 0
0.1, 0.5, 0.45095
0.2, 1, 0.8038
0.3, 1.5, 1.05855
0.4, 2, 1.2152
0.5, 2.5, 1.27375
0.6, 3, 1.2342
0.7, 3.5, 1.09655
0.8, 4, 0.8608
0.9, 4.5, 0.52695
1, 5, 0.095
1.1, 5.5, -0.43505
```