1

回顧 – 指標與其算術運算

- 指標可類比於變數住的房間號碼
- 指標可以當陣列使用，也可說指標可用來當陣列的別名
- 陣列的名稱本身可視為指標

```
int a[] = {1,2,3,4,5};
int *b = a;           // 此時 b 記得 1 所住的房間號碼
cout << *b << b[0];   // 印出 11
cout << b[1];         // 印出 2
cout << b[3];         // 印出 4
```

- 對指標作 + 、- 、++ 、-- 的算術運算，代表將改變所記得的房間號碼

```
b++;                  // 現在 b 記得 2 的房號
cout << b[2];         // 印出 4
b += 2;               // 現在 b 記得 4 的房號
cout << *b;           // 印出 4
```

2

運算子優先順序 (level) 與運算方向 (grouping)

Level	Operator	Description	Grouping
2	() ++ --	postfix	Left-to-right
3	++ -- ! * & + -	unary (prefix) indirection and reference unary sign operator	Right-to-left
4	(type)	type casting	Right-to-left
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
17	= *= /= %= += -=	assignment	Right-to-left

http://www.cplusplus.com/doc/tutorial/operators/

3

```
    int a[10] = {2,4,6,8,10,12,14,16,18,20};
    int *pa = a, *pb = a+9, *pc = &a[5];

2   cout << *pa++;
12  cout << *pc;
19  cout << --*pb;
    pa += 5;
14  cout << *pa--;
    pb -= 2;
16  cout << *pb;
12  cout << *pc++;
13  cout << --*pc;
18  pa += 3; cout << *pa;
6   pb -= 5; cout << *pb;
6   pc -= 4; cout << *pc;
```

* 與 ++ 都是第三階的運算子，結合是由右向左，所以 *pa++ → (pa++) → *pa

--*pb，因為是由右向左運算，所以 --*pa → (*pa) → --(*pa)

4

```
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
    int *pa = a, *pb = a+9, *pc = &a[5];

1   cout << *pa++;
6   cout << *pc;
9   cout << --*pb;
    pa += 5;
7   cout << *pa--;
    pb -= 2;
8   cout << *pb;
6   cout << *pc++;
6   cout << --*pc;
9   pa += 3; cout << *pa;
3   pb -= 5; cout << *pb;
3   pc -= 4; cout << *pc;
```

5

回顧 - 動態記憶體管理

- 利用 new 來取得一個新的房間裝資料
- 利用指標來記得新的房間號碼
- 利用 delete 來將租來的房間歸還 (給作業系統)

```
int *a = new int(5);
*a += 20;
delete a;
```

6

回顧 - 動態記憶體管理 – 單一資料

- 利用 new 來取得一個新的房間裝資料
- 利用指標來記得新的房間號碼
- 利用 delete 來將租來的房間歸還 (給作業系統)

```
int *a = new int(5);
*a += 20;
cout << *a;
delete a;
```

- 這種用法在未來大家學到物件導向比較有用

7

回顧 - 動態記憶體管理 – 陣列

- 利用 new [] 來取得一連串新的房間裝資料
- 利用指標來記得第一間新的房間號碼
- 利用 delete [] 來將租來的房間歸還 (給作業系統)

```
int *b = new int[20];
for(int i=0;i<20;++i) b[i] = i+1;
cout << b[5];
delete[] b;
```

- 這種用法比較有用 ，可動態地決定所需陣列大小

8

Lecture 15

多維陣列之動態記憶體管理與傳遞
變數的儲存等級/生命週期
變數視野

9

多維陣列之動態記憶體管理

10

多維動態記憶體管理的三種方式

- 使用**不夠彈性**的多維動態記憶體管理

```
int (*a)[5]= new int [10][5];
delete [] a;
```

- 使用指標陣列 (pointer array)、或稱為指標的指標 (pointer to pointers)

```
int **a = new int*[10];
for(int i=0;i<10;i++) a[i] = new int[5];
for(int i=0;i<10;i++) delete []a[i];
delete []a;
```

- 將多維陣列轉換為一維陣列 (透過數學上的簡單計算，將多維度的索引轉換成一維的索引)

```
int *a = new int[10*5];
delete [] a;
```

11

不夠彈性的多維動態記憶體管理

15-1.cpp

此方式無法完全動態地配置多維陣列
其中**僅有第一個維度**可以變數指定，
其它維度必需為常數 ... Orz

```
#include <iostream>
using namespace std;
int main() {
  int (*a)[3] = new int[5][3];

  for(int i=0;i<5;i++) {
    for(int j=0;j<3;j++) {
      a[i][j] = i*3 + j;
    }
  }
```

```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```

```
  for(int i=0;i<5;i++) {
    for(int j=0;j<3;j++) {
      cout << a[i][j] << " ";
    }
    cout << endl;
  }
  delete []a;
  return 0;
}
```

12

指標陣列

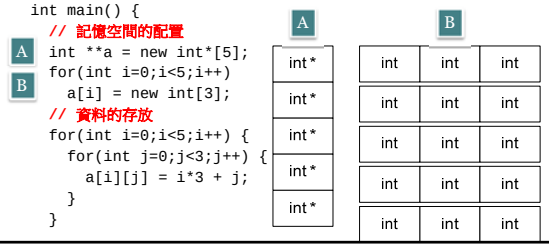
15-2.cpp

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

int main() {
  // 記憶空間的配置
  int **a = new int*[5];
  for(int i=0;i<5;i++)
    a[i] = new int[3];
  // 資料的存放
  for(int i=0;i<5;i++) {
    for(int j=0;j<3;j++) {
      a[i][j] = i*3 + j;
    }
  }
```

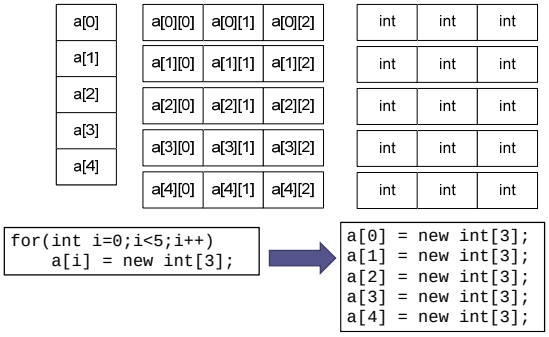
int* *a;
- 和 a 粘在一起的 * 表示 a 為一指標變數(使用起來 ~= 一維陣列)
- int* 代表一維陣列中所存放的資料為整數的指標 (~= 整數的一維陣列)

new int *[5]; ← 配置一個可以存放 5 個 int* 的記憶空間



13

```
int* *a = new int*[5];
```




```
for(int i=0;i<5;i++)
    a[i] = new int[3];
```

```
a[0] = new int[3];
a[1] = new int[3];
a[2] = new int[3];
a[3] = new int[3];
a[4] = new int[3];
```

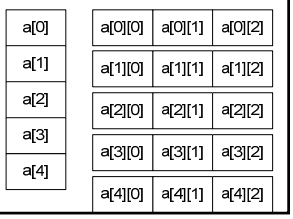
14

15-2.cpp

```
// 資料的輸出
for(int i=0;i<5;i++) {
  for(int j=0;j<3;j++) {
    cout << a[i][j] << " ";
  }
  cout << endl;
}

// 記憶空間的釋放
for(int i=0;i<5;i++)
  delete []a[i];
delete []a;
return 0;
}
```

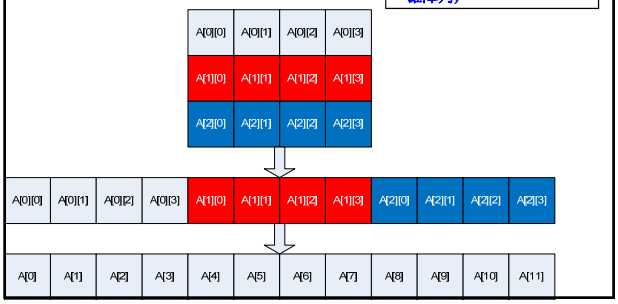
```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```



15

將多維陣列轉為一維陣列

實際上，C/C++ 在內部在處理多維陣列時，也是將多維陣列轉為一維陣，而轉換方式也是如此！(recall: 記憶體可視為龐大的一維陣列)



16

多維陣列轉一維陣列之映射

- 二維陣列中，第一個索引決定是第幾列、第二個索引決定第幾行。
- 每一列有四個元素
- 所以每一行的開頭之一維陣列的索引:第一個索引 * 每行元素個數
- 二維陣列的第二個索引決定了是第幾行，亦即與該列開頭之偏移個數。
  - A[2][2] → A[2][0+2] (該列開頭) → A[8+2] → A[10]
  - A[1][3] → A[1][0+3] → A[4+3] → A[7]
- 二維轉一維的公式: 列索引 * 一列有幾個元素 (陣列行數) + 行索引

17

多維陣列轉一維陣列之反映射

- 先算出應該在第幾列:
  (一維索引 / 一列之元素個數) 採無條件捨去
- 再算出在該列裡面與該列第一個元素之間的偏移:
  一維索引 % 一行元素個數
  10 / 4 → 2, **10** % 4 → 2: A[**10**] → A[2][2]
  7/4 → 1, **7**%4 → 3: A[**7**] → A[1][3]

18

15-3.cpp

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

int main() {
  int *a = new int[5*3];

  for(int i=0;i<5;i++) {
      for(int j=0;j<3;j++) {
          a[i*3+j] = i*3+j;
      }
  }
```

5 列 3 行 – 共需存放 15 個元素

i 控制第幾列、j 控制第幾行
每一列有 3 個元素
i*3+j 即為轉換後一維陣列的索引值

19

15-3.cpp

```
for(int i=0;i<5;i++) {
    for(int j=0;j<3;j++) {
        cout << a[i*3+j] << " ";
    }
    cout << endl;
}

delete []a;

return 0;
}
```

```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```

20

摘要 – 三種多維陣列的動態記憶體管理

- 假設 nRow 為列數、nCol為行數。
- 不夠彈性的多維動態記憶體管理 (p.s. nCol 必需為常數變數)

```
int (*a)[nCol] = new int[nRow][nCol];
delete []a;
```

- 使用指標陣列 (p.s. **配置**與**刪除**都有點麻煩)

```
int **a = new int*[nRow];
for(i=0;i<nRow;i++) a[i] = new int[nCol];
for(i=0;i<nRow;i++) delete []a[i];
delete []a;
```

- 以一維陣列模擬 (p.s. 使用上有點麻煩)

```
int *a = new int[nRow * nCol];
delete []a;
```

21

多維陣列之資料傳遞

22

多維陣列的傳遞

- 目前我們一共有四種多維陣列
  - 固定大小的多維陣列

    `int a[5][3];`
  - 動態配置、可變動大小的陣列三種，但都是指標
    - 陣列指標：int (*a)[3] = new int[5][3];
    - 指標的指標: int **a = new int*[5]; for(int ...
    - 以一維陣列模擬：int *a = new int[5*3];
  - 陣列在傳遞時，其實都是以指標的方式傳遞

23

15-4.cpp – 固定大小陣列或陣列指標

```
#include <iostream>
using namespace std;
void array2D(int a[2][3]) {
  for(int i=0;i<2;i++) {
    for(int j=0;j<3;j++) {
      cout << a[i][j] << ", ";
    }
    cout << "\n";
  }
}

int main() {
  int p[2][3] = { {1,2,3}, {4,5,6}};
  int q[2][3] = { {11,12,13}, {14,15,16}};
  array2D(p);
  array2D(q);
  return 0;
}
```

也可寫為：
`void array2D(int (*a)[3]) {`

陣列在傳遞時，其實它不乎第一個維度的大小。

```
1, 2, 3,
4, 5, 6,
11, 12, 13,
14, 15, 16,
```

24

15-5.cpp – 指標的指標

```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```

```
void pArray(int **, int, int);
int main() {
  int **a = new int*[5];
  for(int i=0;i<5;i++)
    a[i] = new int[3];
  for(int i=0;i<5;i++) {
    for(int j=0;j<3;j++) {
      a[i][j] = i*3 + j;
    }
  }
  pArray(a, 5, 3);
  for(int i=0;i<5;i++)
    delete []a[i];
  delete []a;
  return 0;
}
```

```
void pArray(int **a, int m, int n)
{
  for(int i=0;i<m;i++) {
    for(int j=0;j<n;j++) {
      cout << a[i][j] << " ";
    }
    cout << endl;
  }
}
```

25

15-6.cpp – 一維陣列模擬二維陣列

```
#include <iostream>
using namespace std;
void pArray(int *, int, int);
void sArray(int *, int, int);

int main() {
  int *a = new int[5*3];
  sArray(a, 5, 3);
  pArray(a, 5, 3);

  delete []a;
  return 0;
}
```

```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```

```
void sArray(int *a, int nRow,
int nCol) {
  for(int i=0;i<nRow;i++) {
    for(int j=0;j<nCol;j++) {
      a[i*3+j] = i*3+j;
    }
  }
}
```

```
void pArray(int *a, int nRow,
int nCol) {
  for(int i=0;i<nRow; i++) {
    for(int j=0;j<nCol;j++) {
      cout << a[i*3+j] << " ";
    }
    cout << endl;
  }
}
```

26

變數的儲存等級/生命週期

27

變數的儲存等級/生命週期

- 變數為運算電腦記憶體 (RAM) 的工具
- 變數儲存等級(storage class) / 生命週期代表了
  - 變數何時開始被創造出來 (並開始佔據記憶體)
  - 變數何時被消滅 (歸還其佔據的記憶體給作業系統)
- 在 C/C++ 裡，變數的儲存等級主要有三種：
  - **自動 (automatic)**：變數在該變數所屬的函式/區塊開始執行時創造出來，在函式/區塊結束執行後被消滅。此為預設/內定之儲存等級，到目前為止我們大部份都是使用這種儲存等級的變數。
  - **動態 (dynamic)**：由程式作者決定一變數何時被創造出來、何時被消(new, delete, new[], delete[])。
  - **靜態 (static)**：變數在程式開始執行即被創造出來，在程式結束執行後被消滅。

28

15-7.cpp

在此範例中，變數 a, c 的儲存等級皆為自動，簡稱為自動變數 (auto variables)

```
#include <iostream>
using namespace std;
void func() {
  int c = 5;
  cout << ++c << " ";
}

int main() {
  int a=3;

  cout << a << " ";
  for(int i=0;i<5;i++)
    func();

  return 0;
}
```

func 函式 開始執行：
- **自動**配置一個 4 bytes 記憶體給變數 c，並給初始值 5
- 將 c 加一後，列印出來
- 函式結束執行，**自動**將變數 c 消滅掉，將 4 bytes 的記憶體歸還給作業系統

main 函式 / 主程式開始執行：
- **自動**配置一個 4 bytes 記憶體給變數 a，並給初始值 3
- 將 a 變數的值列印出來
- 重複呼叫 func() 函式 5 次
- 程式結束執行，**自動**將 變數 a 消滅並將 4 bytes 記憶歸還給作業系統

程式執行結果
3 6 6 6 6 6

29

15-8.cpp

在此範例中，變數 a 為自動變數；變數 c 的儲存等級則為靜態，簡稱為靜態變數 (static variables)

```
#include <iostream>
using namespace std;
void func() {
  static int c = 5;
  cout << ++c << " ";
}

int main() {
  int a=3;

  cout << a << " ";
  for(int i=0;i<5;i++)
    func();

  return 0;
}
```

func 函式 開始執行：
- 若此函式為第一次被呼叫，配置4 bytes 記憶體給變數 c。
- 將 c 加一後，列印出來。

main 函式 / 主程式開始執行：
- **自動**配置一個 4 bytes 記憶體給變數 a，並給初始值 3
- 將 a 變數的值列印出來
- 重複呼叫 func() 函式 5 次
- 程式結束執行，**自動**將 變數 a 消滅並將 4 bytes 記憶歸還給作業系統
- 將靜態變數 c 消滅，將 4 bytes 記憶體歸還

程式執行結果
3 6 7 8 9 10

30

靜態變數

- 在函式中的靜態變數在該函式第一次被呼叫時，被配置記憶體並進行初始化的動作，而待整個程式結束執行時，靜態變數的記憶體才會被釋放掉。
- 雖然靜態變數宣告在函式內，並不會隨著函式開始執行而被重新創造出來，也不會因為函式結束執行而被消滅掉。此與自動變數不同。
- 由於變數的初始化 (int c=5;) 是在變數被創造出來佔據記憶體時所執行的動作，因此在函式被重複呼叫時，變數的初始化是不會再被執行的。
- 靜態變數若沒有初始化時，會被初始化為 0。

31

視野 / 範疇

Scope

32

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。

33

15-9.cpp

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

```
#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;
}
```

34

15-9 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)，但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 。

35

15-10.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;
}
```

36

15-10 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 15-10 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。

37

作業九 (6/8 due)

作業十 (6/15 due，不可遲交)

38

隨堂練習

請寫出 pArray 這個函式的定義

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

void pArray(int (*array)[4], int nRow);

int main() {
    int a[5][4] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12},
        {13,14,15,16},
        {17,18,19,20}
    };
    pArray(a, 5);
    return 0;
}
```

程式輸出結果

```
1 5 9 13 17
2 6 10 14 18
3 7 11 15 19
4 8 12 16 20
```