

# C/C++ 進階班 資料結構

## 陣列 (Array)

李耕銘

# 課程大綱

- 指標複習
- 陣列與記憶體位置
- 多維陣列
- 陣列複雜度分析&使用時機

## 指標複習

# 指標複習

## 指標是一種變數

- 變數負責儲存資料，指標儲存的是記憶體位置
- 變數需要符合的規範指標都要遵守

## 當資料沒有名稱時，透過指標所儲存的記憶體位置來使用資料

- 動態記憶體配置
- 函式之間傳遞資料
- 資料的存放與管理(資料結構與演算法)

# 指標複習

## 使用指標的方式

- 1 宣告：幫指標取個變數名稱
- 2 取址：取出目標變數在記憶體的位置
- 3 取值：透過該記憶體位置間接存取資料的值

int	資料型別
50	資料內容
名稱：v	資料名稱
位置：0x01	資料位置

取址

v → 0x01

取值

0x01 → 50

# 指標複習


宣告方式

資料型態 <sup>指標</sup> \* 指標名稱;  
指到的資料型別      宣告指標的變數名稱

宣告一個  
指向整數的指標

int \*p;

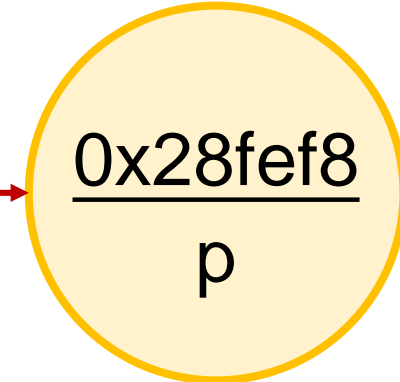
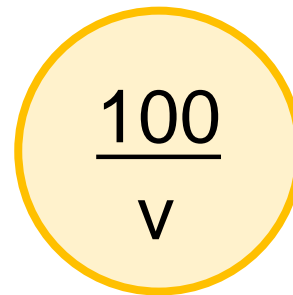
宣告好之後，p就可以  
儲存整數的**記憶體位置**

int\*  
 ← 記憶體位置  
名稱：p

# 指標複習

**&**：取出變數的記憶體位置

```
int *p;  
int v = 100;  
p = &v;  
cout << p;
```



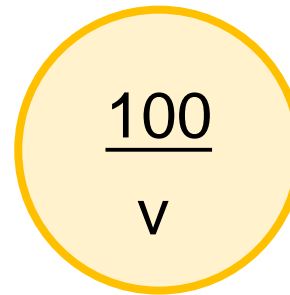
&v : 0x28fef8

The screenshot shows a Windows command prompt window with the title bar "E:\Dropbox\LKM\Desktop\資工訓練班\Code\Lecture1\test\bin\Debug\test.exe". The command prompt displays the output "0x28fef8", followed by "Process returned 0 (0x0) execution time : 0.062 s" and "Press any key to continue.".

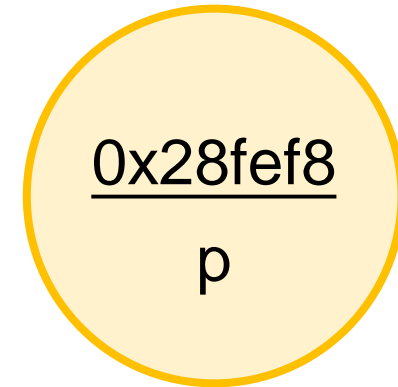
# 指標複習

\* : 取得指標所指位置上存放的值

```
int *p;  
int v = 100;  
p = &v;  
cout << *p;
```



0x28fef8



\*(0x28fef8) : 100

\*p : 100

C:\Users\Test\Desktop\Test\bin\Debug\Test.exe

```
100  
Process returned 0 (0x0)   execution time : 0.021 s  
Press any key to continue.
```



# 動態記憶體配置

- 程式執行時，臨時需要空間來存放資料
- 向作業系統索取一塊記憶體來儲存資料
- 通常都是等待使用者告知需要的空間大小
- 當此記憶體空間不需要後，可將之釋放

# 指標複習

1. 準備好指標
2. 跟作業系統要記憶體空間，把位置給指標



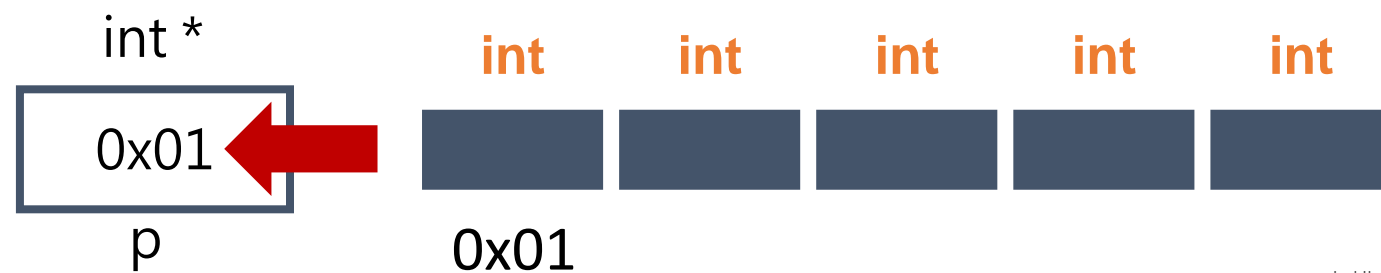
- 2 請作業系統挖出空間



我這裡需要5個int存放資料



- 3 把記憶體位置指派給指標



# 指標複習

- 1 `#include <stdlib.h>`
- 2 宣告出指標備用
- 3 使用`malloc()`使作業系統挖出一塊空間

`Pointer = (資料型態 *) malloc(sizeof(資料型態) * 個數);`

顯性資料轉型

要挖的空間大小

- 4 結束後用`free(指標名稱)` 釋放記憶體。

# 指標複習

**malloc**

配置空間大小後不歸零

```
Pointer = (資料型態 *) malloc(sizeof(資料型態) * 個數);
```

**calloc**

配置空間大小後歸零

```
Pointer = (資料型態 *) calloc(個數, sizeof(資料型態));
```

**realloc**

重新配置空間大小


```
int *p2 = (int*) realloc(p1, sizeof(資料型態) * 個數);
```

# 指標複習

指標是存取該記憶體位置唯一的媒介

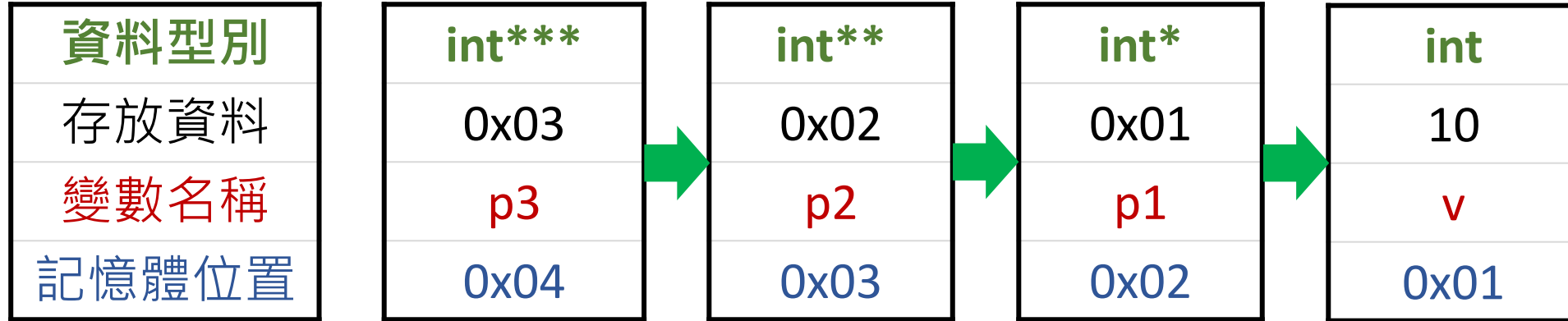
遺失指標→該記憶體位置永遠無法存取

永遠無法存取



```
int len;  
cin >> len;  
int *p = (int *) malloc(sizeof(int)*len);  
p = (int *) malloc(sizeof(int)*len);
```

# 指標複習



- v = ?
- p1 = ?
- p2 = ?
- p3 = ?

- &v = ?
- &p1 = ?
- &p2 = ?
- &p3 = ?

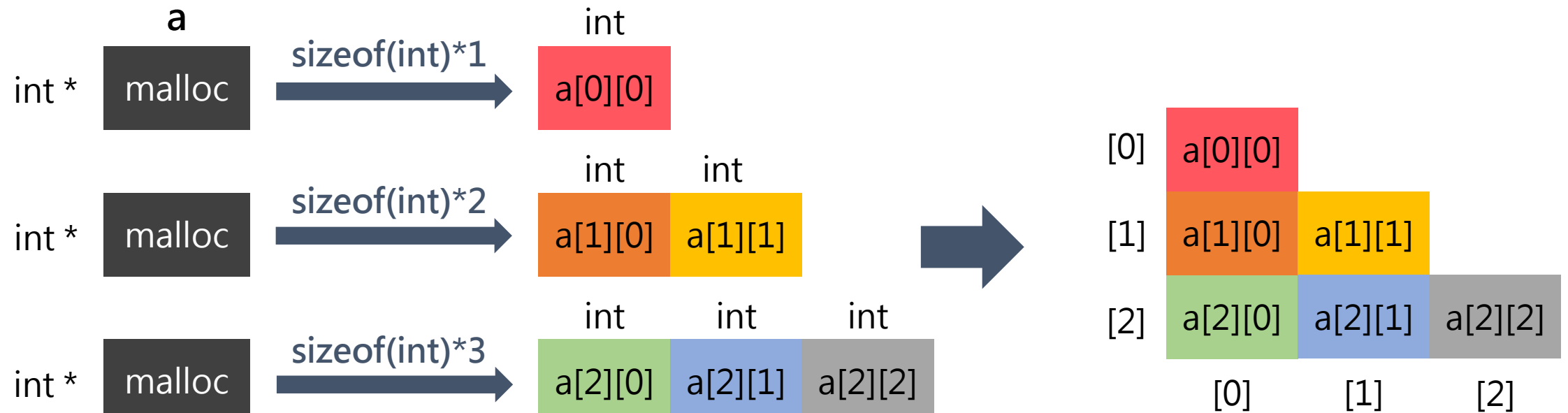
- \*p1 = ?
- \*p2 = ?
- \*\*p2 = ?
- \*p3 = ?
- \*\*p3 = ?
- \*\*\*p3 = ?

# 指標複習

陣列中儲存的是指標

```
int *a[3];
```

指標又指到另一個陣列→二維陣列



# 指標複習

`int**`



`int*`



`int`



`int`



`int`



`int`



`int`



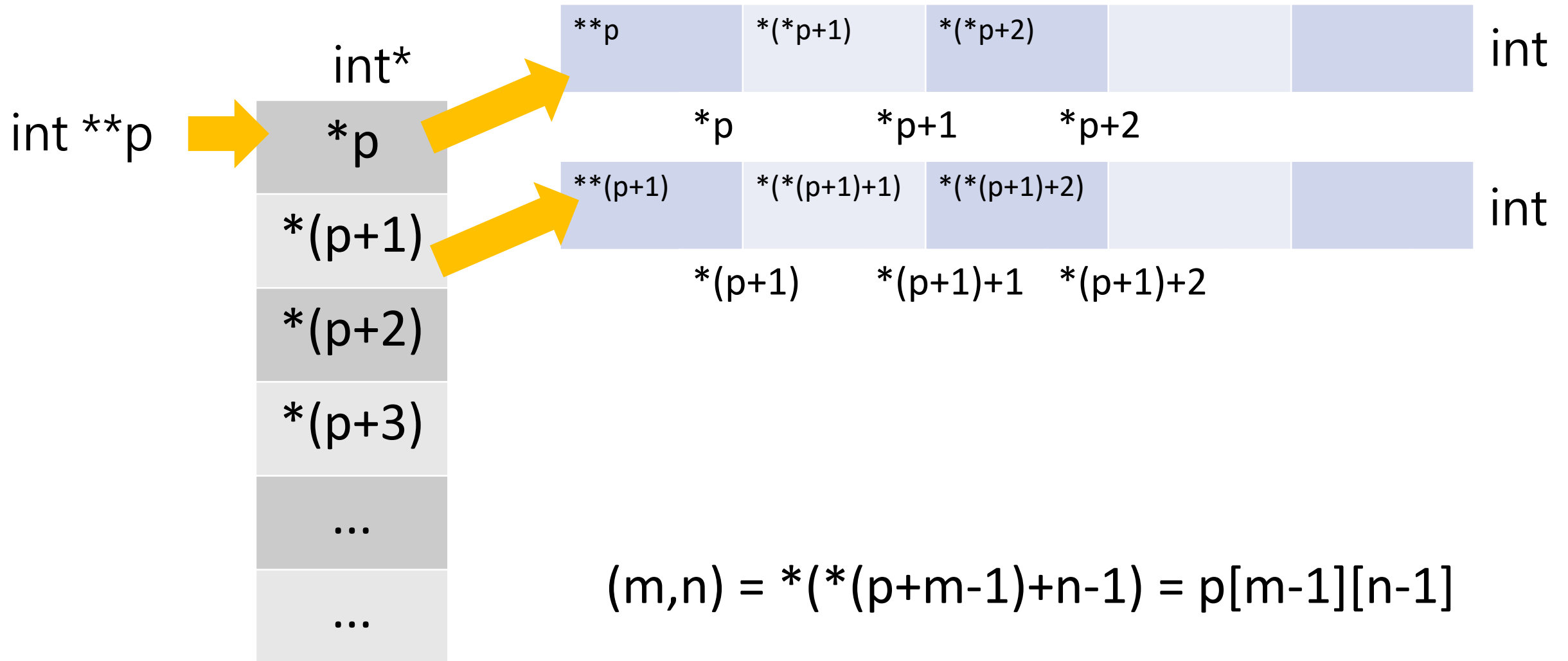
`int`



`int`



# 指標複習



# 陣列與記憶體位置

# 指標複習

## 陣列名稱

表示陣列所在記憶體空間的起始位址

## 索引值

表示該元素距離陣列開頭有多遠

索引值代表距離開頭有多遠

score[0]

score[1]

score[2]

score[3]

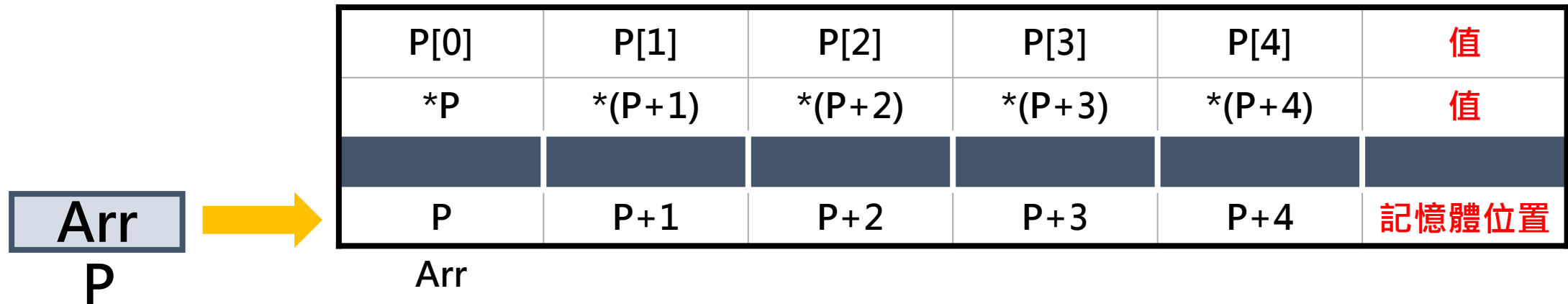
score[4]

score代表開頭的記憶體位置

# 陣列與記憶體位置

- 陣列名稱代表陣列的起始位置
- 把陣列名稱指派給指標，指標指向陣列最開頭

```
int Arr[5];  
int *p = Arr;
```



# 陣列與記憶體位置

```
int Arr[5]={1,2,3,4,5};  
int *p=Arr;  
  
for(int i=0; i<5; i++)  
{  
    cout << *(p+i) <<" ";  
}
```

```
int Arr[5]={1,2,3,4,5};  
int *p=Arr;  
  
for(int i=0; i<5; i++)  
{  
    cout << p[i] <<" ";  
}
```

# 陣列與記憶體位置

```
bool Arr[5]={0,0,1,1,1};
```

```
for(int i=0; i<5; i++)  
{  
    cout << &Arr[i] <<" ";  
}
```

Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]	值
*Arr	*(Arr+1)	*(Arr+2)	*(Arr+3)	*(Arr+4)	值
0x6dfee7	0x6dfee8	0x6dfee9	0x6dfeea	0x6dfeeb	記憶體位置
&Arr[0]	&Arr[1]	&Arr[2]	&Arr[3]	&Arr[4]	記憶體位置

Arr

0x6dfee7 0x6dfee8 0x6dfee9 0x6dfeea 0x6dfeeb

# 陣列與記憶體位置

```
int Arr[5]={1,2,3,4,5};

for(int i=0; i<5; i++)
{
    cout << &Arr[i] <<" ";
}
```

Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]	值
*Arr	*(Arr+1)	*(Arr+2)	*(Arr+3)	*(Arr+4)	值
0x6dfed8	0x6dfedc	0x6dfee0	0x6dfee4	0x6dfee8	記憶體位置
&Arr[0]	&Arr[1]	&Arr[2]	&Arr[3]	&Arr[4]	記憶體位置

Arr

0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4 0x6dfee8

# 陣列與記憶體位置

```
char Arr[5]={'1','2','3','4','5'};
```

```
for(int i=0; i<5; i++)
```

```
{
```

```
    cout << &Arr[i] << " ";
```

```
}
```

12345 2345 345 45 5

?



# 陣列與記憶體位置

```
char Arr[5]={'1','2','3','4','5'};  
  
for(int i=0; i<5; i++)  
{  
    cout << &Arr[i] <<" ";  
}
```

12345 2345 345 45 5

The << operator is overloaded to print the whole cstring when you provide a char\* argument.

# 陣列與記憶體位置

```
char Arr[5]={'1','2','3','4','5'};

for(int i=0; i<5; i++)
{
    cout << (void*) &Arr[i] <<" ";
}
```

Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]	值
*Arr	*(Arr+1)	*(Arr+2)	*(Arr+3)	*(Arr+4)	值
0x6dfee7	0x6dfee8	0x6dfee9	0x6dfeea	0x6dfeeb	記憶體位置
&Arr[0]	&Arr[1]	&Arr[2]	&Arr[3]	&Arr[4]	記憶體位置

Arr

0x6dfee7 0x6dfee8 0x6dfee9 0x6dfeea 0x6dfeeb

# 陣列與記憶體位置

```
bool Arr_Bool[5]={0,0,1,1,1};
char Arr_Char[5]='1','2','3','4','5';
int Arr_Int[5]={1,2,3,4,5};
float Arr_Float[5]={1.5,2.5,3.5,4.5,5.5};
double Arr_Double[5]={1.5,2.5,3.5,4.5,5.5};
cout << "Size of bool array:\t" << sizeof(Arr_Bool) << endl;
cout << "Size of char array:\t" << sizeof(Arr_Char) << endl;
cout << "Size of int array:\t" << sizeof(Arr_Int) << endl;
cout << "Size of float array:\t" << sizeof(Arr_Float) << endl;
cout << "Size of double array:\t" << sizeof(Arr_Double) << endl;
```

```
Size of bool array:      5
Size of char array:     5
Size of int array:      20
Size of float array:    20
Size of double array:   40
```

# 陣列與記憶體位置

若變數大小為  $S$  個位元組，陣列的起始位置為  $P$ ，則陣列中

1. 第  $i$  個索引值、 $i+1$ 個元素的記憶體位置：

- $P + S \times i$

2. 陣列長度為  $len$ ，則陣列的總長度為：

- $len \times S$



# 多維陣列

# 多維陣列

```
int arr[2][3];
```

利用**線性記憶體**空間，換算成二維

arr[0][0] arr[0][1] arr[0][2] arr[1][0] arr[1][1] arr[1][2]



第一個Row

第二個Row

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]

# 多維陣列

```
int arr[2][3] = {1,2,3,4,5,6};

for(int i=0;i<2;i++){
    for(int j=0;j<3;j++){
        cout << &arr[i][j] << " ";
    }
}
```

```
0x6dfed0 0x6dfed4 0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4
Size of this array: 24
```

## 多維陣列

`arr[0][0]`   `arr[0][1]`   `arr[0][2]`   `arr[1][0]`   `arr[1][1]`   `arr[1][2]`



## 第一個Row

## 第二個Row

Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[1][0]	Arr[1][1]	Arr[1][2]	值
0x6dfed0	0x6dfed4	0x6dfed8	0x6dfedc	0x6dfee0	0x6dfee4	記憶體位置
&Arr[0][0]	&Arr[0][1]	&Arr[0][2]	&Arr[1][0]	&Arr[1][1]	&Arr[1][2]	記憶體位置

Arr

```
0x6dfed0 0x6dfed4 0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4
Size of this array: 24
```



# 多維陣列

若變數大小為  $S$ ，陣列起始位置為  $L$ ，陣列長寬為  $m \times n$ ，則該二維陣列中第  $(i, j)$  個元素(1開始計數)的記憶體位置：

- $L + (i - 1) \times n \times S + (j - 1) \times S$

Row-First

m		n						
		L	$L + 1 \times S$	$L + 2 \times S$	...	...	...	$L + (n - 1) \times S$
		$L + n \times S$	$L + (n + 1) \times S$	$L + (n + 2) \times S$	...	...	...	...
		$L + 2n \times S$	$L + (2n + 1) \times S$	$L + (2n + 2) \times S$	...	...	...	...
		...	...	...	...	...	...	...
		...	...	...	...	...	...	...
		$L + (i - 1) \times n \times S$	...	...	...	(i,j)	...	...
		...	...	...	...	...	...	...

# 多維陣列

若變數大小為  $S$ ，陣列起始位置為  $L$ ，陣列長寬為  $m \times n$ ，則該二維陣列中第  $(i, j)$  個元素(1開始計數)的記憶體位置：

- $L + (j - 1) \times m \times S + (i - 1) \times S$

Column-First

n							
m	L	$L+m \times S$	$L+(2m) \times S$	...	$L + S \times (j - 1) \times m$	...	...
	$L+1 \times S$	$L+(m+1) \times S$	$L+(2m+1) \times S$	...	...	...	...
	$L+2 \times S$	$L+(m+2) \times S$	$L+(2m+2) \times S$	...	...	...	...
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...
	...	...	...	...	(i,j)	...	...
	...	...	...	...	...	...	...

# 多維陣列

```
int arr[2][3] = {1,2,3,4,5,6};  
  
for(int i=0;i<2;i++){  
    cout << arr[i] << " ";  
}  
cout << endl;
```



這是甚麼？

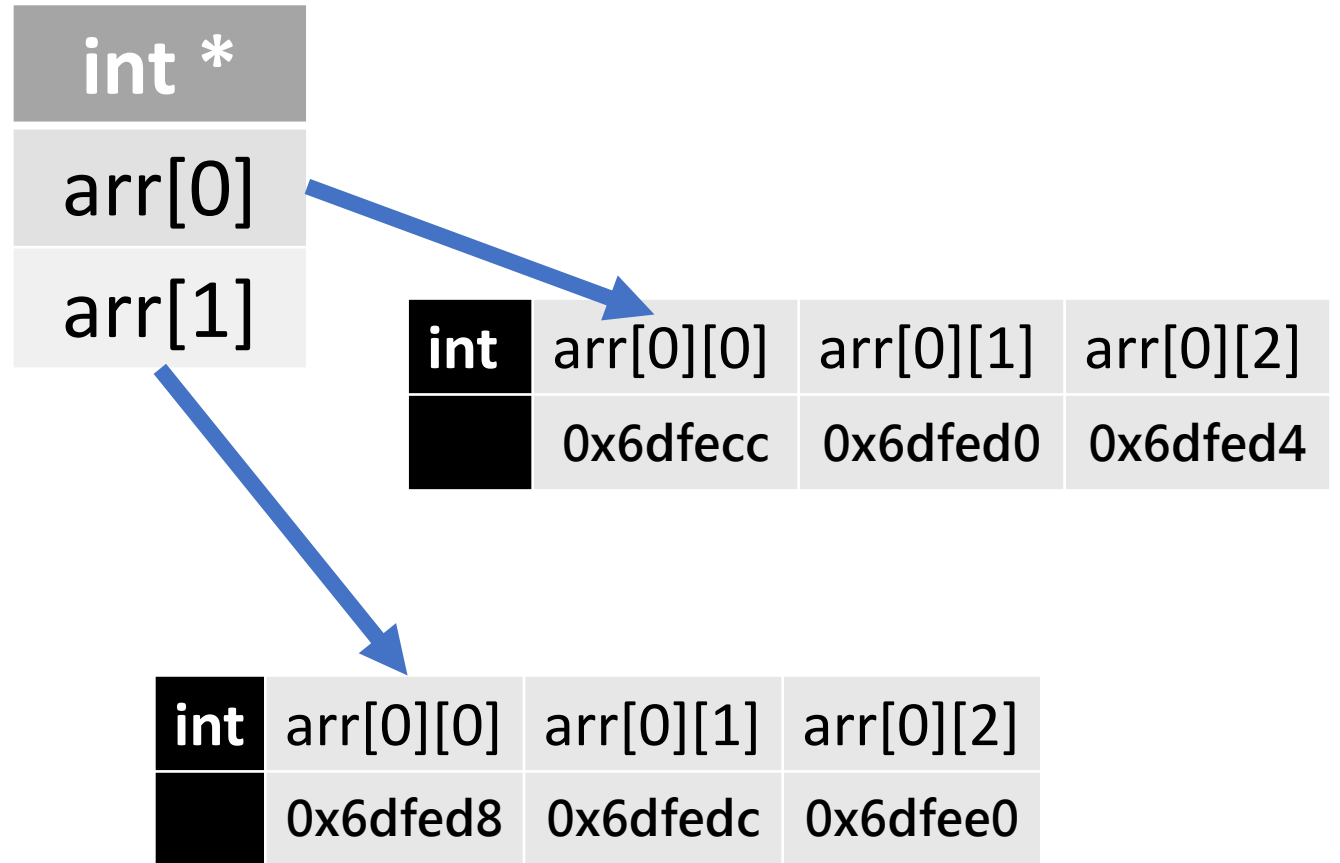
0x6dfecc 0x6dfed8

# 多維陣列

```
int arr[2][3] = {1,2,3,4,5,6};
cout << "arr = " << arr << endl;

for(int i=0;i<2;i++){
    cout << arr[i] << " ";
}
cout << endl;

for(int i=0;i<2;i++){
    for(int j=0;j<3;j++){
        cout << &arr[i][j] << " ";
    }
}
cout << endl;
```



```
arr = 0x6dfecc
0x6dfecc 0x6dfed8
0x6dfecc 0x6dfed0 0x6dfed4 0x6dfed8 0x6dfedc 0x6dfee0
```

# 多維陣列

```
int arr[2][3] = {1,2,3,4,5,6};  
  
for(int i=0;i<2;i++){  
    for(int j=0;j<3;j++){  
        cout << (*(arr+i)+j) << " ";  
    }  
}
```

**$*(*(arr+i)+j)=arr[i][j]$**

1 2 3 4 5 6

# 多維陣列

```
int arr[2][3] = {1,2,3,4,5,6};  
cout << "arr = " << arr << endl;  
  
for(int i=0;i<6;i++){  
    cout << *(arr+i) << " ";  
}  
cout << endl;
```

```
arr = 0x6dfed4  
0x6dfed4 0x6dfee0 0x6dfeec 0x6dfef8 0x6dff04 0x6dff10
```

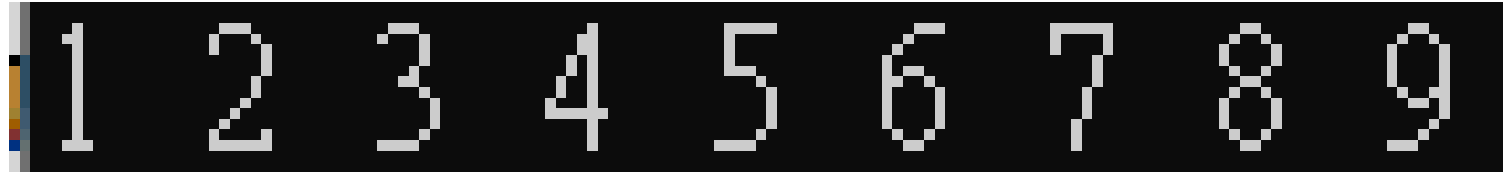
Q : Why ?

A : arr 被轉成雙重指標

# 多維陣列

```
int arr[3][3] = {1,2,3,4,5,6,7,8,9};
```

```
int *p = (int*)arr;  
for(int i=0;i<9;i++)  
    cout << *(p+i) << " ";
```



```
1 2 3 4 5 6 7 8 9
```

# 多維陣列

```
clock_t s,f;
const int n = 600;
auto arr_3d = new int[n][n][n];
int counts = 0;

s = clock();
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        for(int k=0;k<n;k++){
            arr_3d[i][j][k] = counts;
            counts++;
        }
    }
}
f = clock();
cout << "Time consumed:" << (f-s)/((double)CLOCKS_PER_SEC << "s" <<endl;

s = clock();
counts = 0;
int *p = (int*) arr_3d;
for(int i=0;i<n*n*n;i++){
    *p = counts;
    counts++;
}
f = clock();
cout << "Time consumed:" << (f-s)/((double)CLOCKS_PER_SEC << "s" <<endl;
```

為什麼要學這個？

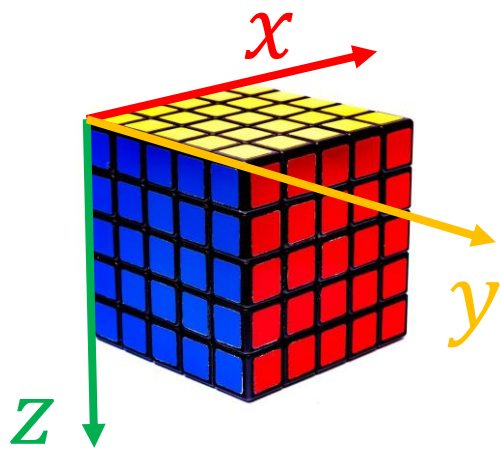
```
Time consumed:1.064s
Time consumed:0.629s
```



# Practice

若變數大小為  $S$ ，陣列起始位置為  $L$ ，陣列長寬高為  $x \times y \times z$ ，則該三維陣列中，第  $(i, j, k)$  個元素(1開始計數)的記憶體位置為何(Row-First)？

請用  $L$ 、 $i$ 、 $j$ 、 $k$ 、 $x$ 、 $y$ 、 $z$ 、 $S$  表示。



$$\begin{aligned} & \&data[i][j][k] \\ &= L \\ &+ (i - 1) \times y \times z \times S \\ &+ (j - 1) \times z \times S \\ &+ (k - 1) \times S \end{aligned}$$

# Practice

## Mission

宣告一 3x3x3 陣列，並試著在單一 **for 迴圈** 內把值初始化成 1~27

## 陣列複雜度分析&使用時機

# 陣列的效能分析



score\_1 score\_2 score\_3 score\_4 score\_5 score\_6 score\_7 score\_8

- 搜尋
  - Upper bound :  $N$
  - Average bound :  $(N+1)/2$
  - Lower bound :  $1$
  - $O(N)$

# 陣列的效能分析



score\_1 | score\_2 | score\_3 | score\_4 | score\_5 | score\_6 | score\_7 | score\_8

- 新增/刪除第一個元素
  - $O(N)$
- 新增/刪除最後一個元素
  - $O(1)$
- 新增/刪除特定引數
  - $O(N)$

- 搜尋特定元素
  - $O(N)$
- 走訪
  - $O(N)$
- 記憶體需求
  - 只需儲存資料本身

# 陣列的效能分析

- Pros

- 利用記憶體의連續位置來記錄次序資訊
- 利用 index 即可在  $O(1)$  時間對Array的資料做存取。
- 最節省記憶體空間的方式

- Cons

- 只能儲存順序、無法儲存其餘資料間的對應關係
- 搜尋需要 $O(N)$ 的時間
- 新增/刪除特定位置，需要 $O(N)$ 時間做搬移資料
- 若長度改變，也會花費 $O(N)$ 的時間在遷移資料

# 陣列的效能分析

- 陣列使用時機
  - 希望透過索引值快速存取資料
  - 資料數量是**固定**的，執行後不會改變長度
  - 希望使用的記憶體空間越少越好。

score\_1

score\_2

score\_3

score\_4

score\_5



4 Bytes



4 Bytes



4 Bytes



4 Bytes



4 Bytes

# 常見的資料操作

- access 取出第  $i$  個資料： $O(1)$
- search 搜尋(未事先排序)： $O(n)$
- delete 刪除特定元素： $O(n)$
- insert 插入特定元素： $O(n)$
- 無法對長度做修改，每次插入都會有資料遺失
- 最節省空間的資料結構！



score_1	score_2	score_3	score_4	score_5
---------	---------	---------	---------	---------



# 陣列與編譯

- 陣列長度必須在編譯時就決定好
  - C99後支援 VLA(Variable Length Array)
  - 但 C++11中為非必要功能
  - 盡量讓陣列長度為定值 (define、const)
- 程式開始執行後，無法改變陣列長度。

# static vs. dynamic array

- Static array
  - 直接宣告出的陣列：`int data[5];`
  - 可用 `sizeof` 來取得陣列大小
  - 不能修改大小
- Dynamic array
  - 使用 `malloc` 或 `new` 開出的陣列
  - 無法得知陣列大小，需另外紀錄
  - 可以修改大小

# 陣列使用時機

- 陣列長度固定無法彈性擴充
  - 少增修移動、多查找
- 可以直接透過索引值查詢資料
  - 索引值查找快速： $O(1)$
- 耗費的記憶體空間最小
- C++ 中應盡量以 **vector** 取代 **array** 的使用
  - 傳統指標沒有自動化的垃圾回收機制

# 陣列使用時機

```
#include <iostream>

using namespace std;

int data[100000];

int main() {

    return 0;

}
```

- 解題技巧：如果題目有給測資的大小上限
  - 直接把陣列開好開滿到上限
    - ✓ 評分看執行時間，記憶體消耗量不影響
    - ✓ 記得宣告成全域變數
      1. 執行前就配置好空間
      2. 能容納的空間也較大
    - ✓ 實務上別這麼做 (通常競試限定)

**評分說明：**輸入包含若干筆測試資料，每一筆測試資料的執行時間限制(time limit)均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 10 分， $N = 2$ ，且取用次數  $f(1)=f(2)=1$ 。

第 2 子題組 20 分， $N = 3$ 。

第 3 子題組 45 分， $N < 1,000$ ，且每一個物品  $i$  的取用次數  $f(i)=1$ 。

第 4 子題組 25 分  $N \leq 100,000$