資料結構

C 演算法

結 構

Structure

結構 Structure

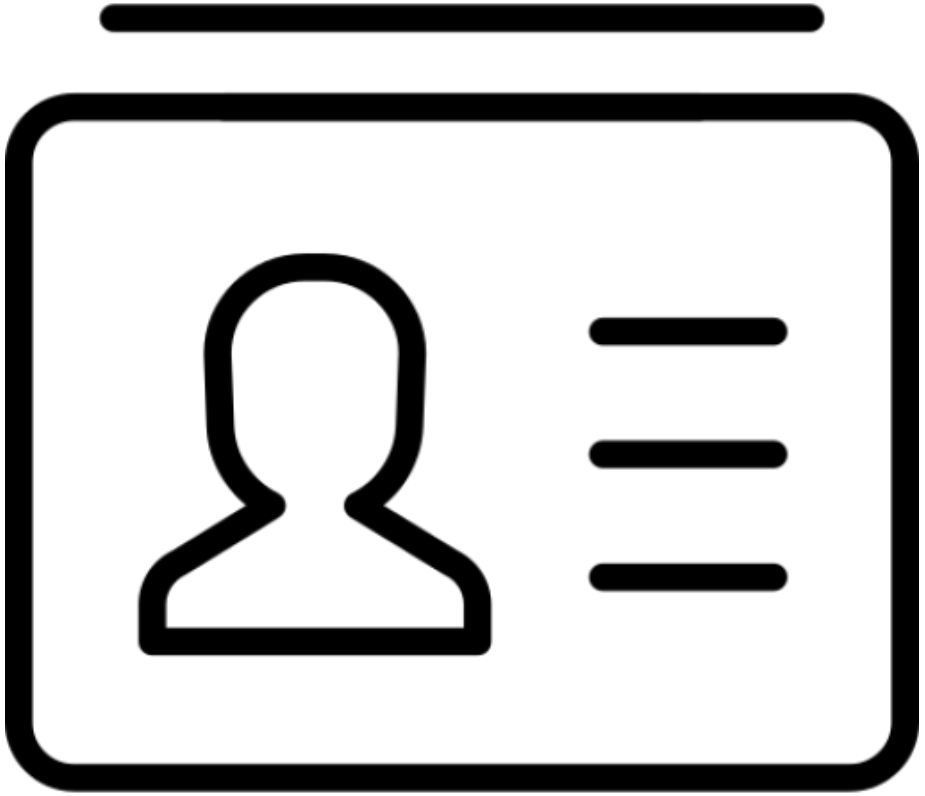

什麼是結構？

什麼是結構?

- 結構是一種由使用者自訂的資料型態
  - 系統預設的資料型態: int, float, double, char, ...
  - 使用者自訂的資料型態: struct

- 結構是**各種不同資料型態的集合**
  - 可以組合多個系統預設的資料型態(int, float, double, char, 陣列, 指標, ...)，產生新的資料型態

為什麼要用結構 (1) ？



- 如何記錄一個學生的個人資料？

```
char name[] = "John";
int age = 12;
char gender[] = '男';
char phone[] = '21567234';
```

為什麼要用結構 (2) ？

如何記錄多個學生的個人資料？

- 第 1 種方法: 每個資料都用一個變數

```
char name1[] = "John";
int age1 = 12;
char gender1[] = '男';
char phone1[] = '21567234';

char name2[] = “May";
int age2 = 11;
char gender2[] = ‘女';
char phone2[] = ‘13248945';
                …
```

問題: 變數太多，會崩潰的！

為什麼要用結構 (3) ？

如何記錄多個學生的個人資料？

- 第 2 種方法: 使用陣列

```
char name[][]=["John", "May“, …];
int age=[12, 11, …];
char gender[][]=['男', '女‘, …];
char phone[][]=['21567234', '13248945‘, …];
```

問題: 如果其中一種資料少打了一個值的話，那後面會全亂掉！

例: 第 2 個學生的 age 沒輸入的話，那第 3 個學生的資料，會是 name[2], age[1], gender[2], phone[2]，容易讓索引值混亂。

為什麼要用結構 (3) ？

如何記錄多個學生的個人資料？

- 第 3 種方法: 使用**結構**

```
struct studentStruct {
   char name[10];
   int age;
   char gender[4];
   char phone[10];
};
```

每個學生的資料使用
一個型態單位記錄，
彼此不會相互影響，
存取資料也更直覺！


定義與宣告
結構(變數)

定義結構

```
struct 結構標籤{
    資料型態 變數名稱;
    資料型態 變數名稱;
      …
};
```

定義結構 – 拆解說明 1

1

用來標示後面會定義一個結構型態。

如函式前面會加上 function 標示一樣

結構(struct)是結構(structure)的縮寫

```
struct 結構標籤{
    資料型態 變數名稱;
    資料型態 變數名稱;
      …
};
```

定義結構 – 拆解說明 2

2

要建立的**結構型態標籤**。

如系統預設的整數型態為 int

```
struct 結構標籤{
    資料型態 變數名稱;
    資料型態 變數名稱;
       …
};
```

定義結構 – 拆解說明 3

```
struct 結構標籤{
    資料型態 變數名稱;
    資料型態 變數名稱;
    …
};
```

在結構內的變數稱為成員(member)

3

結構內包含的**成員(member)**資料。

如: char name[10];

定義結構 – 拆解說明 4

```
struct 結構標籤{
    資料型態 變數名稱;
    資料型態 變數名稱;
    …
};
```

4 大括號後面一定要加上**分號(;)**

宣告結構變數

- 方法1: 定義結構時宣告變數
- 方法2: 先定義結構，再另外宣告變數
- 方法3: 定義結構，搭配 typedef 產生新的資料型態名稱，再使用新的資料型態宣告變數

方法1: 定義結構時宣告變數

使用方法 1 宣告變數時，可以將結構名稱省略。

```
struct 結構標籤{
   資料型態  變數名稱;
   資料型態  變數名稱;
   …
}變數名稱1, 變數名稱2;
```

```
struct {
   資料型態  變數名稱;
   資料型態  變數名稱;
   …
}變數名稱1, 變數名稱2;
```

- 缺點：此種方式的缺點是之後不能在宣告此種型態的變數。
- 優點：不需要定義結構名稱

宣告結構變數

方法1: 定義結構時宣告變數

```c
#include <stdio.h>

struct pointStruct {
  int x;
  int y;
} point1, point2;

int main()
{
  return 0;
}
```

```c
#include <stdio.h>

struct {
  int x;
  int y;
} point1, point2;

int main()
{
  return 0;
}
```

方法2: 先定義結構，再另外宣告變數

當變數另外宣告時，結構名稱絕對不可以省略！

```
struct 結構標籤{
   資料型態  變數名稱;
   資料型態  變數名稱;
     …
};

struct 結構名稱  變數名稱1,
  變數名稱2;
```

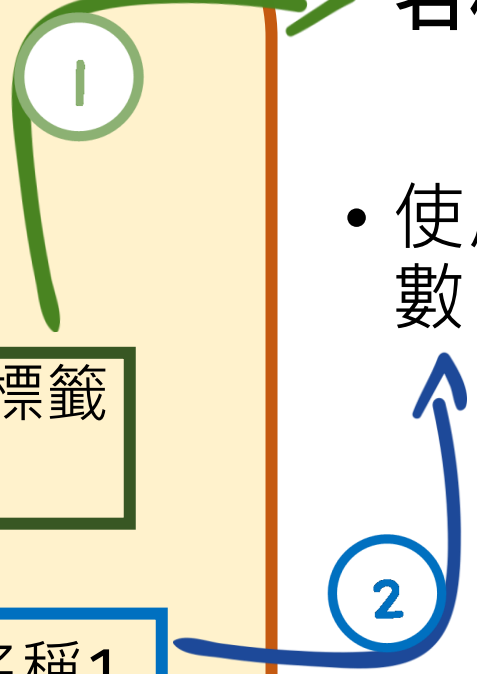
```c
#include <stdio.h>

struct pointStruct {
   int x;
   int y;
};

int main()
{
   struct pointStruct point1;
   struct pointStruct point2;
   return 0;
}
```

方法3: typedef 定義新的資料型態(1)

```
struct 結構標籤{
   資料型態 變數名稱;
   資料型態 變數名稱;
     …
};
```

```
typedef struct 結構標籤
  新的資料型態名稱;
```

```
新的資料型態名稱 變數名稱1,
  變數名稱2;
```



- 用 typedef 替 **struct 結構名稱** 取新別名
- 使用 typedef 取的別名宣告變數

方法3: typedef 定義新的資料型態(2)

```
typedef struct 結構標籤{
    資料型態  變數名稱;
    資料型態  變數名稱;
    …
}新的資料型態名稱;

新的資料型態名稱  變數名稱1,
    變數名稱2;
```

- 定義結構時，在前面加上 typedef，就可直接在大括號後面加上新的資料型態。
- 使用 typedef 取的別名宣告變數

更多關於 typedef 的說明可參考後面『**延伸概念 4: typedef**』

宣告結構變數

注意: 方法 1 與 方法 3 別搞混！

```
struct 結構標籤{
   資料型態  變數名稱;
   資料型態  變數名稱;
   …
}變數名稱1, 變數名稱2;
```

一般的 struct 大括號後面是接要宣告此型態的**變數名稱**

```
typedef struct 結構標籤{
   資料型態  變數名稱;
   資料型態  變數名稱;
   …
}新的資料型態名稱;
```

前面有加上 typedef 的 struct 大括號後面是接要此**結構的別名**

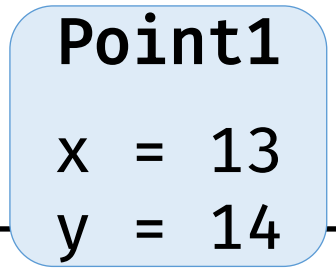
結構變數初始化

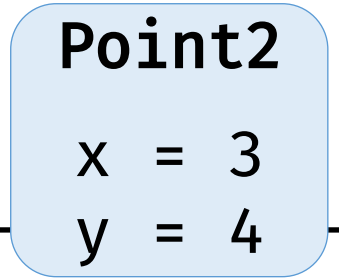
- 方法1：不指名，依序對應結構內的成員變數
  - 根據結構定義中的成員順序對應給值
  - 必須注意對應順序，否則可能會造成錯誤的型態資料對應。
    - 如：整數資料存到指標字元變數內。

- 方法2：指名道姓，明確指出哪個值對應哪個成員變數

方法1：不指名的對應給值　方法２：指名道姓設定值

```
struct pointStruct { point1={13,14};      point2={x:3,y:3};
   int x;
   int y;
} point1, point2;
```





結構變數初始化 – 範例

方法2：明確指出結構內的成員 x 與 y 分別要設為多少。

```c
#include <stdio.h>

struct pointStruct {
   int x;
   int y;
}point1, point2;

typedef struct pointStruct pointType;

int main()
{
   pointType point1 = {13, 14};
   pointType point2 = {x: 3, y: 4};
   return 0;
}
```

1

2

方法1：依照結構成員的定義順序，會依序將 13 存在 x 變數，14 存在 y 變數。

指標型態的結構

```
struct pointStruct {
   int x;
   int y;
};


struct pointStruct point1;
struct pointStruct *ptrPoint;

ptrPoint = &point1;
```

結構變數同樣是使用 * 符號代表是指標型態

同樣使用取址(&)符號設定指標型態的結構變數值。

結構陣列

```
struct pointStruct {
   int x;
   int y;
};


struct pointStruct point1[10];
```

用同樣的陣列宣告方式宣告結構陣列

結構陣列的初始化

若要對結構陣列內每筆陣列元素(結構變數[0],結構變數[1]…)各別設定值，就與一般結構變數設定值的方式一樣(point[0] = {3,5,'c'})，若要對整個陣列設定值就用此方式。

取得結構陣列指定索引值的結構成員

```c
#include <stdio.h>
struct pointStruct {
  int x;
  int y;
  char z;
}point;

int main()
{
  struct pointStruct point[3] = {
    {3, 5, 'c'},
    {4, 1, 'a'},
    {1, 6, 'd'}
  };
  printf("p0.x: %d, p0.y: %d, p0.z: %c\n", point[0].x, point[0].y, point[0].z);
  printf("p1.x: %d, p1.y: %d, p1.z: %c\n", point[1].x, point[1].y, point[1].z);
  printf("p2.x: %d, p2.y: %d, p2.z: %c\n", point[2].x, point[2].y, point[2].z);
  return 0;
}
```

存取結構變數成員 – 一般結構變數

- 使用「.」運算符號可以存取一般結構內的成員變數

結構變數名稱.成員變數名稱

```
struct pointStruct point1;
point1.x = 13;
point1.y = 14;
```

存取結構變數成員 – 指標型態

- 使用「`->`」運算符號可以存取指標型態的成員變數

結構變數名稱`->`成員變數名稱

```
struct pointStruct point1;
point1.x = 13;
point1.y = 14;
```

巢狀結構 – 包含其它結構

- 結構的成員定義中可以包含其它結構

```
struct 結構標籤A {
   資料型態 變數名稱;
   資料型態 變數名稱;
   struct 結構標籤B 變數名稱;
};
```

巢狀結構 – 本身結構

含有自己本身的結構型態指標的結構稱為**自我參考結構(self-referential structure)**

不可以包含自己本身的結構

```
struct 結構標籤A {
   資料型態 變數名稱;

   資料型態 變數名稱;

   struct 結構標籤A 變數名稱;
};
```

可以包含自己本身的指標結構

```
struct 結構標籤A {

   資料型態 變數名稱;

   資料型態 變數名稱;

   struct 結構標籤A *指標變數名稱;
};
```


結構與函式

傳遞結構給函式 – call by value

```c
#include <stdio.h>
struct pointStruct {
  int x;
  int y;
};
typedef struct pointStruct pointType;

int showPoint (pointType point)
{
  printf("[showPoint1]x: %d, y: %d\n", point.x, point.y);
  point.x = 10;
  point.y = 13;
  printf("[showPoint2]x: %d, y: %d\n", point.x, point.y);
  return 0;
}
int main()
{
  struct pointStruct point = {3, 5};
  showPoint(point);
  printf("[main]x: %d, y: %d\n", point.x, point.y);
  return 0;
}
```

直接傳遞結構變數給函式時，與一般變數傳遞相同，是使用 call by value，在函式內會在另外使用一個記憶體空間記錄截購內的值，因此在函式內更改的值外面是無法接收到的。

執行結果

```
dice - struct $ ./傳遞結構給函式-byvalue.exe
[showPoint1]x: 3, y: 5
[showPoint2]x: 10, y: 13
[main]x: 3, y: 5
```

傳遞結構給函式 – 使用指標

```c
#include <stdio.h>
struct pointStruct {
  int x;
  int y;
};
typedef struct pointStruct pointType;

int showPoint (pointType *point)
{
  printf("[showPoint1]x: %d, y: %d\n", point->x, point->y);
  point->x = 10;
  point->y = 13;
  printf("[showPoint2]x: %d, y: %d\n", point->x, point->y);
  return 0;
}
int main()
{
  struct pointStruct point = {3, 5};
  showPoint(&point);
  printf("[main]x: %d, y: %d\n", point.x, point.y);
  return 0;
}
```

使用指標的方式傳遞給函式時，式傳遞結構的位址，在函式內與原呼叫區的結構變數是使用同一個記憶體位置的資料，因此在函式內更改結構的值才能在外面接收到。

執行結果

```
dice - struct $ ./傳遞結構給函式-byvalue.exe
[showPoint1]x: 3, y: 5
[showPoint2]x: 10, y: 13
[main]x: 3, y: 5
```

函式回傳結構型態

```c
#include <stdio.h>
struct pointStruct {
  int x;
  int y;
};
typedef struct pointStruct pointType;

pointType initPoint(int x, int y)
{
  pointType point;
  point.x = x;
  point.y = y;
  return point;
}
int main()
{
  pointType point;
  int x, y;
  scanf("%d %d", &x, &y);

  point = initPoint(x, y);

  printf("x: %d, y: %d\n", point.x, point.y);
  return 0;
}
```

initPoint() 函式回傳的資料型態為 pointType 結構

執行結果

```
dice - struct $ ./函式回傳結構.exe
50 12
x: 50, y: 12
```

延伸的概念

概念1: 結構 V.S. 陣列

- 結構與陣列類似，可同時用一個變數表達多個資料。

- 陣列：同種資料型態的集合
  - 例: 使用 fruits 陣列變數記錄多項水果

```
char fruits[] = [“apple”, “banana”, “strawberry”];
```

- 結構：多像同種或不同種資料型態的集合
  - 例:使用 point 變數表達一個坐標資料

```
struct pointStruct {
  int x;
  int y;
} point;
```

概念2: 名稱不衝突

結構標籤與結構成員的名稱是可以相同的

```c
#include <stdio.h>
struct point {
  int point;
};
typedef struct point t_point;
int main()
{
  int point;
  t_point p;
  scanf("%d", &point);
  p.point = point;
  printf("p.point: %d\n", p.point);
  return 0;
}
```

結構內的成員變數名稱與外部的變數名稱是可以相同的。

不同對象的命名有不同的作用空間，因此只要避免會造成程式閱讀混亂的情況，一般情況下是可以使用相同命名的。

執行結果

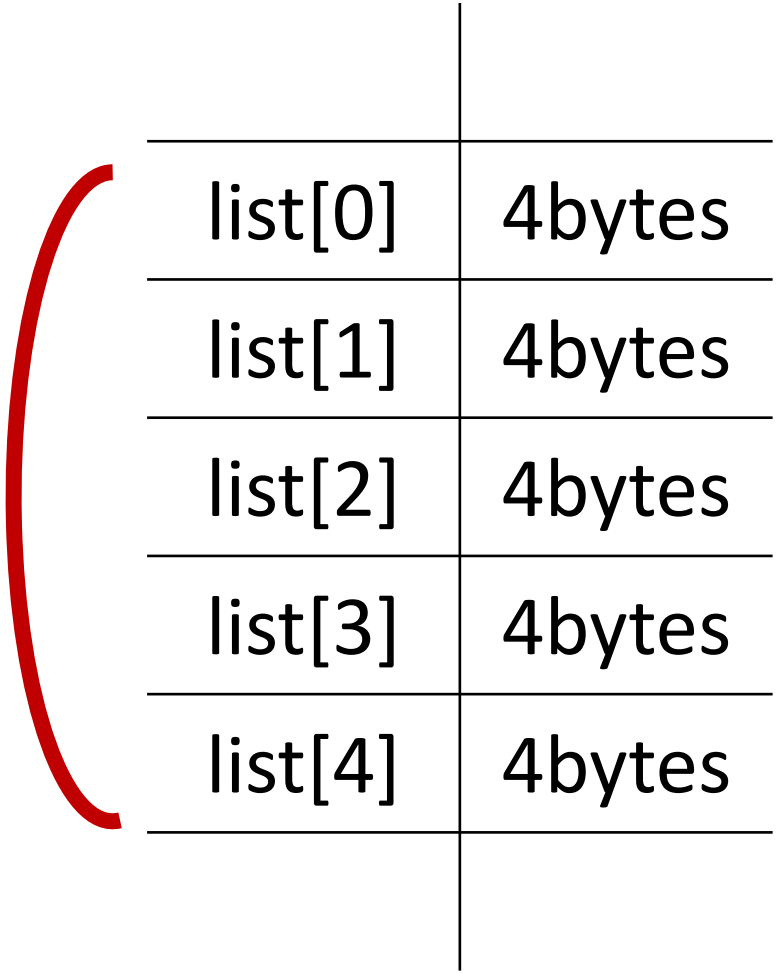
```
dice - struct $ ./名稱不衝突.exe
11
p.point: 11
```

概念3: 記憶體配置 – 陣列

- 陣列的記憶體配置

```
int list[5];
```

佔用一段連續的記憶體空間
sizeof(int) * 5 => 20 bytes



概念3: 記憶體配置 – 結構(1)

- 結構的記憶體配置

```
struct studentStruct
{
  char name[10];
  int age;
  char gender[4];
  char phone[10];
};
```

佔用一段連續的記憶體空間

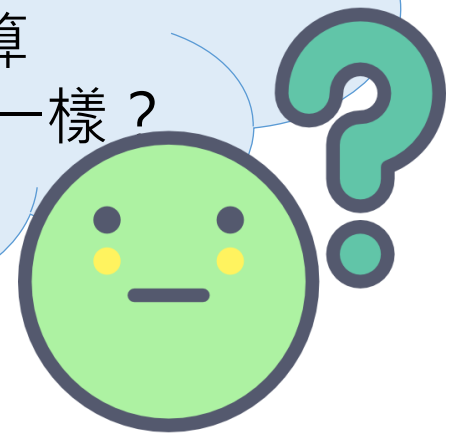
10+4+4+10=> 28 bytes

name[0]	1bytes
name[1]	1bytes
⋮	⋮
name[9]	1bytes
age	4bytes
gender[0]	1bytes
gender[1]	1bytes
gender[2]	1bytes
gender[3]	1bytes
phone[0]	1bytes
phone[1]	1bytes
⋮	⋮
phone[9]	1bytes

概念3: 記憶體配置 – 結構(2)

```c
#include <stdio.h>

struct studentStruct {
  char name[10];
  int age;
  char gender[4];
  char phone[10];
}student;

int main()
{
  printf("size: %d\n", sizeof(student));
  return 0;
}
```

使用 sizeof() 得到結構使用的記憶體大小為 32 bytes，與預期估算(28bytes)的大小不一樣？



執行結果

```
dice - struct $ ./記憶體配置-結構.exe
size: 32
```

概念3: 記憶體配置 – 結構(3)

- 編譯器編譯時，為加快執行速度，會將變數強迫放在記憶體的偶數位址(2 或 4 或 8 或 16的倍數)

- 結構內的每個成員變數也會被放在記憶體的偶數位址(2 或 4 或 8 或 16的倍數)

結構變數的記憶體大小 ≧ 結構內成員的記憶體總和大小

概念3: 記憶體配置 – 結構(4)

記憶體配置-結構.c

```c
#include <stdio.h>

struct studentStruct {
  char name[10];
  int age;
  char gender[4];
  char phone[10];
}student;

int main()
{
  printf("size: %d\n", sizeof(student));
  return 0;
}
```

```
dice - struct $ ./記憶體配置-結構.exe
size: 32
```

結構實際佔用的記憶體空間
10+2+4+4+10+2=32 bytes

額外產生的兩個
2bytes 的空間不會被
使用到，會在記憶體內
形成小**空洞**。

name[0]	1bytes
name[1]	1bytes
⋮	⋮
name[9]	1bytes
X	1bytes
X	1bytes
age	4bytes
gender[0]	1bytes
gender[1]	1bytes
gender[2]	1bytes
gender[3]	1bytes
phone[0]	1bytes
phone[1]	1bytes
⋮	⋮
phone[9]	1bytes
X	1bytes
X	1bytes

概念3: 記憶體配置 – 結構陣列

```c
#include <stdio.h>
struct studentStruct {
  char name[10];
  int age;
  char gender[4];
  char phone[10];
}student[10];

int main()
{
  printf("size: %d\n“,sizeof(student[0])*10);
  return 0;
}
```

由於前面說明的記憶體配置方式，在計算結構陣列大小時，會先取得陣列內單一元素(單一結構變數)的大小後，在乘上陣列大小。

sizeof(結構變數[0]) * 陣列大小

執行結果

```
dice - struct $ ./記憶體配置-結構陣列.exe
size: 320
```

概念4: typedef

- typedef: 替資料型態取新別名(代號)

typedef 已存在的資料型態 新的資料型態名稱

- int
- float
- double
- char
- struct 結構名稱

- Int *
- float *
- double *
- char *
- struct 結構名稱 *

概念4: typedef – 預設資料型態(1)

```
typedef int myInt;
int intVar1 = 10;
myInt intVar2 = 10;
```

1

•intVar1 與 intVar2
同是整數資料型態

2

```
typedef char myChar;
char chVar1 = 'c';
myChar chVar2 = 'c';
```

•chVar1 與 chVar2
同是字元資料型態

```
typedef char* myCharPtr;
char *chPtrVar1 = &chVar1;
myCharPtr chPtrVar2 = &chVar2;
```

3

•chPtrVar1 與 chPtrVar2
同是字元指標資料型態

概念4: typedef – 預設資料型態(2)

```c
#include <stdio.h>

typedef int myInt;
typedef char myChar;
typedef char* myCharPtr;

int main()
{
  myInt intVar = 10;
  myChar chVar = 'c';
  myCharPtr chPtrVar = &chVar;

  printf("intVar: %d\n", intVar);
  printf("chVar: %c\n", chVar);
  printf("value of chPtrVar: %c\n", *chPtrVar);
  return 0;
}
```

執行結果

```
dice - struct $ ./typedef原有資料型態.exe
intVar: 10
chVar: c
value of chPtrVar: c
```

概念4: typedef – 自訂資料型態 struct (1)

```c
struct pointStruct {
  int x;
  int y;
};

typedef struct pointStruct
pointType;
```

```c
typedef struct pointStruct
{
  int x;
  int y;
}pointType;
```

這兩種同樣是將
struct pointStruct
取了一個 pointType 的別名

概念4: typedef – 自訂資料型態 struct (2)

```c
#include <stdio.h>

struct pointStruct {
  int x;
  int y;
};

typedef struct pointStruct
pointType;

int main()
{
  pointType point1;
  pointType point2;
  return 0;
}
```

typedef定義新的資料型態-2.c

```c
#include <stdio.h>

typedef struct pointStruct {
  int x;
  int y;
}pointType;

int main()
{
  pointType point1;
  pointType point2;

  return 0;
}
```

重點整理

- 什麼是結構？
- 如何定義結構？
- 如何宣告結構變數？（有三種）
- 結構變數如何初始化？（有兩種）
- 結構也有陣列與指標
- 巢狀結構
- 如何存取結構成員(一般型態與指標型態)
- 傳遞結構給函式與函式回傳結構型態
- 記憶體配置