CHAPTER 10 - STRUCTURES, UNIONS, BIT MANIPULATIONS, AND ENUMERATIONS

學生資料例子

- 假設共有100筆學生資料，每筆學生資料含：

姓名	char	name[100][10];
學號	char	id[100][10];
出生年月	time_t	birthday[100];
住址	char	address[100][20];
監護人	char	guardian[100][10];
…		

- 想想看寫一個互換兩學生資料的函式swap()，參數列要怎麼寫？

```
void swap(char* namea, char *ida,…..,
          char* nameb, char *idb, …..)
{
      char temp[20];
      strcpy(temp,namea);
      strcpy(namea,nameb);
      strcpy(nameb,temp);
      ….
}
```

  - 若是要處理每一筆學生資料，就必須知道name, id, birthday, address, guardian等都是要處理的對象，漏一個就會出錯。

- **物件的資料成員要組織在一起，才方便處理。**

C語言自定型態

- C語言裡可以透過下面方式，自訂自己的型態
  - struct
  - union
  - typedef
  - enum

定義結構與結構變數(1/3)

- **結構變數**
  - 將相關的變數用一個單一型態的名字集合起來。
  - 常常被使用於檔案存取的單一筆記錄。
  - 常常與指標變數結合形成linked lists, stacks, queues, 與 trees等資料結構來組織資料。

- **定義結構**
  - 範例1:

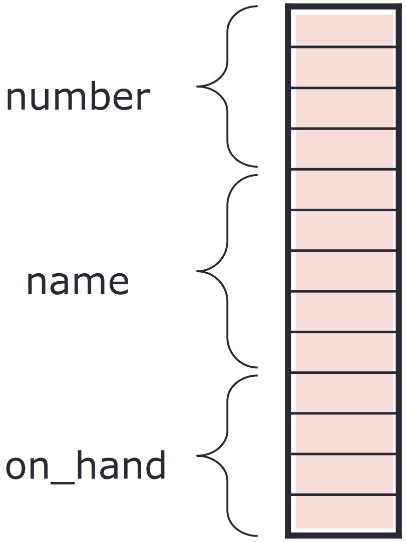
```
struct WareStruct {
      int   id;
      float price;
 };
```

- ● struct 定義結構關鍵字
- ● WareStruct 是結構
- ● struct WareStruct 包含兩個成員:
  int id與float price。

  - 範例2:

```
struct part {
    int number;
    char   name[5];
    int on_hand;
};
```

- ● struct part包含3成員:
int number; char name[5];
int on_hand;



- **定義結構變數:** 給予結構定義後，就可以定義此結構型態的變數。

struct part x, data[100]; //定義了變數x與此結構的陣列data[100]

  - 每個struct part結構變數皆擁有3成員int number; char name[5], int on_hand;

定義結構與結構變數(2/3)

- 結構能包含其它已定義的結構，但不能包含自己這種結構的資料，不過可以有指向自己這種結構的指標。
- **struct僅僅定義型態，並未宣告變數，所以不佔記憶體空間。**
- 宣告結構變數:
  - 範例1: 先定義結構，然後再宣告結構變數。

```
struct point {
   double x,y;
};
struct point onePoint, points[ 100 ] ;
```

  - 範例2:定義結構同時宣告結構變數。

```
struct point { double x,y; } anotherPoint;
```

- 定義結構後，不可再重複定義相同名字的結構。

✕

```
struct A {
     struct A a;
};
```

○

```
struct A {
     struct A *a;
};
struct BasicWareStruct {
  int   id;
  float price;
};
struct ExtendedWareStruct {
   struct BasicWareStruct x;
   int       color;
};
```

若嫌每次定義struct變數還要寫struct這個字很麻煩，你可以用typedef，例如：

typedef struct {double x,y} point;

接下來的變數宣告這樣寫就可以

point onePoint, points[100];

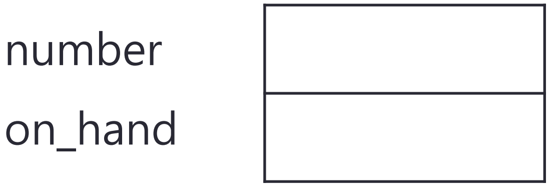
定義結構與結構變數(3/3)

- part1與part2是struct part變數

```
struct part {
      int number;
      int on_hand;
} part1, part2;
```

- part1, part2結構變數各自有記憶體空間

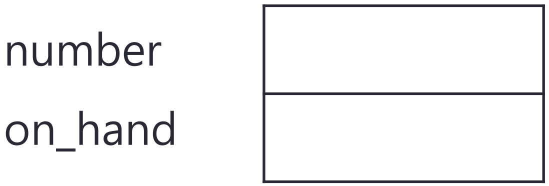
part1有資料成員



number

on_hand

part2有資料成員



number

on_hand

存取結構成員

- 用.存取結構變數成員

```
struct point aPoint;
printf( "%lf %lf\n", aPoint.x, aPoint.y );
```

```
struct point {
   double x;
   double y;
} ;
```

- 用-> 存取結構指標變數成員

```
struct point *aPointPtr = &aPoint;
printf( "%lf\n", aPointPtr->x );
```

結構能包含其它已定義的結構

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
struct student {
  struct person_name name;
  int id, age;
  char sex;
}
```

```
/*
   Using the structure member and
   structure pointer operators */
#include <stdio.h>

/* card structure definition */
struct card {
   char *face; /* define pointer face */
   char *suit; /* define pointer suit */
}; /* end structure card */

int main()
{
   struct card aCard; /* define one struct card variable */
   struct card *cardPtr; /* define a pointer to a struct card */

   /* place strings into aCard */
   aCard.face = "Ace";
   aCard.suit = "Spades";

   cardPtr = &aCard; /* assign address of aCard to cardPtr */

   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      ( *cardPtr ).face, " of ", ( *cardPtr ).suit );

   return 0; /* indicates successful termination */

} /* end main */
```

再看學生資料例子(1/2)

- 假設共有100筆學生資料，每筆學生資料含：

姓名	char	name[10];
學號	char	id[10];
出生年月	time_t	birthday;
住址	char	address[20];
監護人	char	guardian[10];
…		

- 同一物件的資料要組織在一起，才方便處理。

```
struct student_record {
      char          name [10];
      char          id [10];
      time_t        birthday;
      char          address [20];
      char          guardian[10];
} students[100];
```

- 現在swap()怎麼寫？

```
void swap(struct student_record*a,struct student_record* b)
{
   struct student_record temp;
   temp = *a;
   *a    = *b;
    *b    = temp;
}
```

再看學生資料例子(2/2)

- 同一物件的資料將視為一體。

```
struct student_record a, b, *aPtr;
a = b; /*將b的內容複製到a， b之資料成員將會複製到a */
aPtr = &a;
```

- 亦可取得其資料成員
  - 欲取得b之成員name,語法為b.name。

```
printf("%s",b.name);
```

  - 指標aPtr所指的資料成員name,語法為aPtr->name

```
printf("%s", aPtr->name);
```

C結構變數的運算

- 相較於C內定型態的變數，結構變數可以使用的操作較少:
  - 結構變數指派: =
  - 得到結構變數位置: &
  - 存取結構變數成員:
    - 存取一般結構變數成員用 .
    - 存取結構指標變數所指的成員用->
  - 使用 sizeof來決定這個結構的大小
    - 不用會怎樣？ 自己算會算錯嗎？
      - 交由Compiler計算才不易出錯
- 結構變數不可以使用的操作
  - 不能使用算數運算子+,-,*,/,%。
  - 不能使用關係運算子>,>=,<,<=,==,!=。
  - 但是可以自行定義函式完成算數或關係運算子運算。例如比較兩個結構變數，必須自己寫比較函式。

```
struct student_rec a, b;
int res=memcmp(&a,&b,sizeof(a)); //一定對嗎?
```

初始化結構變數

```
struct point {
    double x,y;
};
```

結構變數初始值的寫法

```
strcut point onePoint = { 10, 20 };
```

效果如同

```
struct point onePoint;
onePoint.x = 10;
onePoint.y = 20;
```

陣列結構變數初始值的寫法

```
struct point dataPoint[]
 = {{20, 54}, {90, 880}};
```

效果如同

```
struct point dataPoint[2];
dataPoint[0].x = 20;
dataPoint[0].y = 54;
dataPoint[1].x = 90;
dataPoint[1].y = 880;
```

使用結構變數與函式

- 結構變數可為函式參數
  - 如一般變數，整個變數會複製過去 (Call-by-value)

```
void f(struct part part1) { … }
```

  - 缺點: 當結構變數佔很多位元組時效率不好。

- 可使用指標來傳遞結構變數

```
void f(struct part *part1) {
      part1->number = 100;
          // …
}

strcut part x;
f(&x);
```

- 使用結構可以辦到使用call-by-value方式來傳遞陣列
  - 將陣列宣告為某結構的成員
  - 宣告參數為此結構型態

```
struct my_struct {
            int data[100];
};

void func(struct my_struct a)
```

- 函式回傳值亦可為struct型態

```
struct point build_point(double x, double y)
{
      struct part p;
      p.x = x; p.y = y;
      return p;
}
```

函式呼叫

```
struct point aPoint;
aPoint = build_point(1,10);
```

例子: complex number

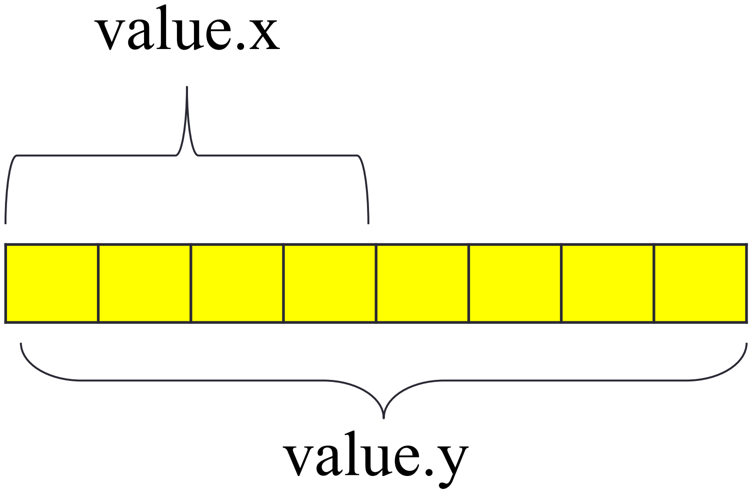
```
#include<stdio.h>
typedef struct {
    float real,imaginary;
} complex_t;

complex_t add_complex(complex_t a, complex_t b)
{
    complex_t c;
    c.real = a.real + b.real;
    c.imaginary = a.imaginary + b.imaginary;
    return c;
}

complex_t read_complex()
{
    complex_t c;
    scanf("%f %f",&c.real, &c.imaginary);
    return c;
}

void print_complex(complex_t a)
{
    printf("(%f+%fi)",a.real,a.imaginary);
}
```

```
void main()
{
    complex_t a,b,c;
    printf("Enter two complex numbers:");
    a = read_complex();
    b = read_complex();
    c = add_complex(a,b);
    printf("complex number:");
    print_complex(a);
    printf("+");
    print_complex(b);
    printf("=");
    print_complex(c);
    printf("\n");
}
```

Unions

- **union**
  - **union** 的成員共用記憶體
- **union** 宣告語法與結構相同

```
union Number {
  int x;
  double y;
};
union Number value;
```



union 型態可用的運算

- 同樣型態 union可以用=
  - **union Number a,b;**
  - **b = a;**
- 取得位置: **&**
  - **union Number *aPtr, a;**
  - **aPtr = &a;**
- 取得union資料成員: **.**
  - **a.x = 0;**
- 使用指標取得union資料成員: **->**
  - **aPtr->x = 0;**

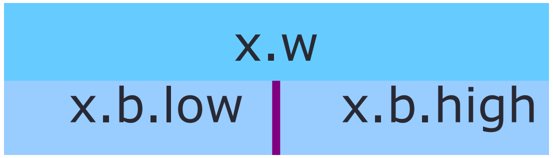
```c
/*
   An example of a union */
#include <stdio.h>

union number {
   int x;
   double y;
};

int main()
{
   union number value;

   value.x = 100;
   printf( "%s\n%s\n%s%d\n%s%f\n\n",
          "Put a value in the integer member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );

   value.y = 100.0;
   printf( "%s\n%s\n%s%d\n%s%f\n",
          "Put a value in the floating member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );
   return 0;
}
```

```
Put a value in the integer member
and print both members.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000.00000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

使用union來方便寫程式

```
typedef union{
   unsigned short w;
   struct {
     unsigned char low, high;
   } b;
}  WORD ;

WORD x;
x.w = 0;
x.b.high = 1;
x.b.low  = 1;
printf("%#x",x.w);    //印出0x101;
```

故意使用union將word與2 bytes安排在一起，方便程式撰寫



使用union來省空間

```
        struct catalog_item {
          int stock_number;
          double price;
          int item_type;
          char title[TITLE_LEN+1];
books     char author[AUTHOR_LEN+1];
          int num_pages;
mug       char design[DESIGN_LEN+1];
          int colors;
shirts
          int sizes;
        };
```

項目要不是書就是大杯子不然就是襯衫，因此將有另一些空間將是閒置的

```
struct catalog_item {
 int stock_number;
 double price;
 int item_type;
 union {
    struct {
     char title[TITLE_LEN+1];
     char author[AUTHOR_LEN+1];
     int num_pages;
    } book;
    struct {
    char design[DESIGN_LEN+1];
    } mug;
   struct {
    char design[DESIGN_LEN+1];
    int colors;
    int sizes;
    } shirt;
  } item;
};
```

使用union製作異質型態

```
struct catalog_item {
 int stock_number;
 double price;
 int item_type;
 union {
   struct {
    char title[TITLE_LEN+1];
    char author[AUTHOR_LEN+1];
    int num_pages;
   } book;
   struct {
   char design[DESIGN_LEN+1];
   } mug;
  struct {
   char design[DESIGN_LEN+1];
   int colors;
   int sizes;
   } shirt;
 } item;
};
```

程式根據item_type來決定該讀book或是mug還是shirt

```
typedef struct {
  int kind;
  union {
    int i;
    double d;
  } u;
} number;

void print(number x)
{
   if (x.kind == INT_KIND)
     printf("%d",x.u.i);
  else
     printf("%f",x.u.d);
}
```

位元運算子 (Bitwise Operators)

- 所有的資料事實上都是一連串的bits
  - 每個 bit的值不是 **0**就是 **1**
  - 連續的 8 bits成為1 byte
  - 整數，浮點數，字元等型態以不相同的方式解讀一連串的bits

Operator	Name	Description
&	bitwise AND	The bits in the result are set to **1** if the corresponding bits in the two operands are both **1**.
|	bitwise OR	The bits in the result are set to **1** if at least one of the corresponding bits in the two operands is **1**.
^	bitwise exclusive OR	The bits in the result are set to **1** if exactly one of the corresponding bits in the two operands is **1**.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with **0** bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
~	One’s complement	All **0** bits are set to **1** and all **1** bits are set to **0**.

- &= Bitwise AND assignment operator
- |= Bitwise inclusive OR assignment operator
- ^= Bitwise exclusive OR assignment operator
- <<= Left-shift assignment operator
- >>= Right-shift assignment operator

	AND (z = x & y;)							
x	1	0	1	0	1	0	1	1
y	1	1	0	1	1	0	1	1
z	1	0	0	0	1	0	1	1

	OR (z = x | y;)							
x	0	0	1	0	1	1	1	1
y	0	1	1	1	1	1	1	1
z	0	1	1	1	1	1	1	1

	AND (z = x & y;)							
x	1	0	1	0	1	1	1	1
y	1	0	0	0	0	0	0	0
z	1	0	0	0	0	0	0	0

	XOR (z = x ^ y;)							
x	1	0	1	0	1	1	1	1
y	1	1	0	1	0	1	0	0
z	0	1	1	1	1	0	1	1

	z = x << 1;							
x	1	0	1	0	1	1	1	1
z	0	1	0	1	1	1	1	0

XOR應用範例

- 一個n元素的整數陣列裡，除了其中一個整數之外，其他整數都兩兩存在。找出那個落單的整數。

- XOR有
  - 交換律: a ^ b = b ^ a
  - 結合律: (a ^ b) ^ c = a ^ (b ^ c)

    a ^ b ^ d ^ c ^ a ^ b ^ c = (a ^ a) ^ (b ^ b) ^ (c ^ c) ^ d = 0 ^ 0 ^ 0 ^ 0 ^ d

- 程式

```
single = 0
for(i = 0; i < n; ++i) {
        single ^= data[i];
}
```

```c
/*
   Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR and bitwise complement operators */
#include <stdio.h>

void displayBits( unsigned );

int main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
   printf( "The result of combining the following\n" );
   displayBits( number1 );
   displayBits( mask );
   printf( "using the bitwise AND operator & is\n" );
   displayBits( number1 & mask );

   number1 = 15;
   setBits = 241;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator | is\n" );
   displayBits( number1 | setBits );

   number1 = 139;
   number2 = 199;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( number2 );
   printf( "using the bitwise exclusive OR operator ^ is\n" );
   displayBits( number1 ^ number2 );

   number1 = 21845;
   printf( "\nThe one's complement of\n" );
   displayBits( number1 );
   printf( "is\n" );
   displayBits( ~number1 );

   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 31;

   printf( "%7u = ", value );

   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         putchar( ' ' );
   }

   putchar( '\n' );
}
```

MASK created with only one set bit

i.e. (**10000000 00000000 00000000 00000000**)

The **MASK** is constantly **AND**ed with **value**.

MASK only contains one bit, so if the **AND** returns true it means **value** must have that bit.

value is then shifted to test the next bit.

```
The result of combining the following
   65535 = 00000000 00000000 11111111 11111111
       1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
       1 = 00000000 00000000 00000000 00000001

The result of combining the following
      15 = 00000000 00000000 00000000 00001111
     241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
     255 = 00000000 00000000 00000000 11111111

The result of combining the following
     139 = 00000000 00000000 00000000 10001011
     199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
      76 = 00000000 00000000 00000000 01001100

The one's complement of
   21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

```
void displayBits(unsigned short value)
{
  unsigned short c, displayMask = 1 << 15;
  printf("%7u = ",value);
  for(c = 1; c <= 16; c++) {
    putchar(value & displayMask ? '1': '0');
    value <<=1;
  }
}
```

value	1	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0
displayMask	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
&	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

output '1'

value<<1	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0
displayMask	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
&	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

output '0'

value<<1	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0
displayMask	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
&	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

output '1'

Idioms for bitwise operations

- The zeroth bit is the least significant bit.
  - Set the ith bit of an unsigned *x* to one

    x |= 1u<<i; $(x = x|(1<<i))$
  - Set the ith bit of an unsigned *x* to zero

    x &= ~(1u<<i);
  - Read the value of the ith bit of an unsigned x

    x&(1u<<i) ? 1 : 0;
  - x multiplied by 2^{i} $\times 2^{n}$

    x <<= i;
  - x divided by 2^{i} $\div 2^{n}$

    x >>= i;

必考!

Bit Fields

- Bit field
  - 更有效率的使用記憶體
  - 只能與 **int** or **unsigned配合使用**

- 宣告bit fields (Declaring bit fields)
  - **unsigned** or **int** 成員後加上**:**與一個數字表示此bit field有幾bits
  - Example:

```
struct BitCard {
  unsigned face : 4;
  unsigned suit : 2;
  unsigned color : 1;
};
```

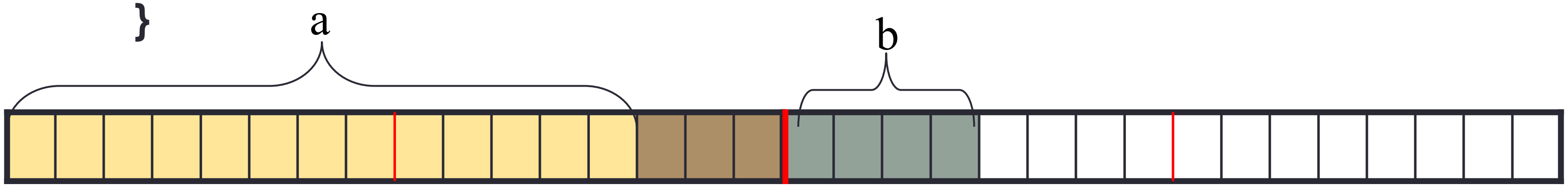
Bit Fields

- 無名的bit field (Unnamed bit field)
  - 為了錯開bit field

```
struct Example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
}
```

  - 無名的bit field 大小若為0將會讓下一個bit field出現在下一個新的儲存單位(int，unsigned)

```
struct Example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
}
```



Enumeration Constants

- 列舉(Enumeration)
  - 用identifiers來表示一個整數集合的數
  - 其實會從0開始每個加**1**
  - Example:

    enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

    - 會產生Months這種型態，有12種值分別為JAN,…,DEC.
      - **事實上是1,2,…,12**
      - **enum Months u;**
      - **for(u=JAN; u <= DEC; ++u) { }**
  - 列舉變數能assign他們自己的列舉常數

```
#include<stdio.h>

enum Bool {FALSE, TRUE};

void main()
{
		enum Bool x,y,z;
		x = 0;
		y = TRUE;
		z = FALSE;
		printf("x=%d y=%d z=%d\n",x,y,z);
}
```

```c
/*
   Using an enumeration type */
#include <stdio.h>

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };

int main()
{
   enum months month;
   const char *monthName[] = { "", "January", "February",
                               "March", "April", "May",
                               "June", "July", "August",
                               "September", "October",
                               "November", "December" };

   for ( month = JAN; month <= DEC; month++ )
      printf( "%2d%11s\n", month, monthName[ month ] );

   return 0;
}
```

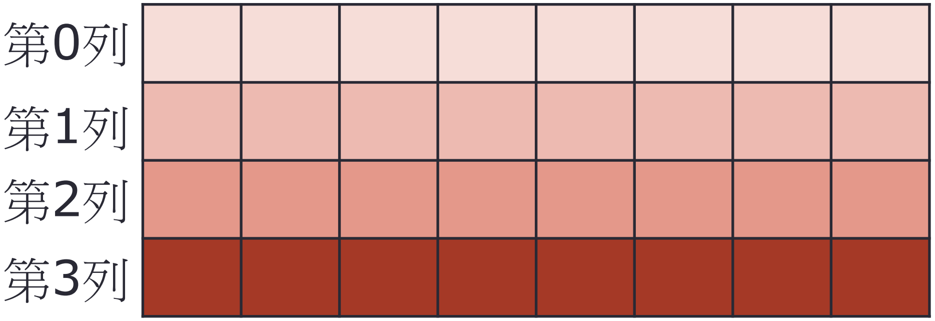
```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

基礎資料結構

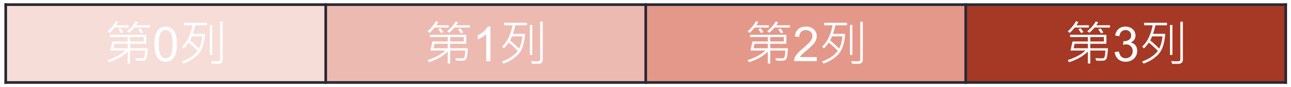
陣列資料結構

- 可透過陣列足標迅速地隨機存取陣列裡任一筆資料。
- C語言陣列是以列為主的順序安排陣列元素。

int A[4][8];//32個元素



A[0][0],A[0][1],…,A[0][7],A[1][0],…,A[3][0],A[3][1],…,A[3][7]



若A[0][0]位置為α， A[i][j]位置為α+(i*8+j)*sizeof(int)

int B[7][5][6];
B[0][0][0],B[0][0][1],…,B[0][0][5],B[0][1][0], …,B[6][0][0],…,B[6][4][5]

若B[0][0][0]位置為β， B[i][j][k]位置為β+(i*5*6+j*6+k)*sizeof(int)

挪移陣列裡資料以便插入一個陣列元素

欲插入資料9到A[0]與A[1]間

[0]	[1]	[2]	[3]	…	[n-2]	[n-1]	[n]
10	8	7	6	…	1	0	

資料往後挪移來空出A[1]

[0]	[1]	[2]	[3]	…	[n-2]	[n-1]	[n]
10	8	8	7	…	..	1	0

```
for(int j=n; j >1; j--) {
    A[j] = A[j-1];
}
```

將9放入A[1]

[0]	[1]	[2]	[3]	…	[n-2]	[n-1]	[n]
10	9	8	7	…	..	1	0

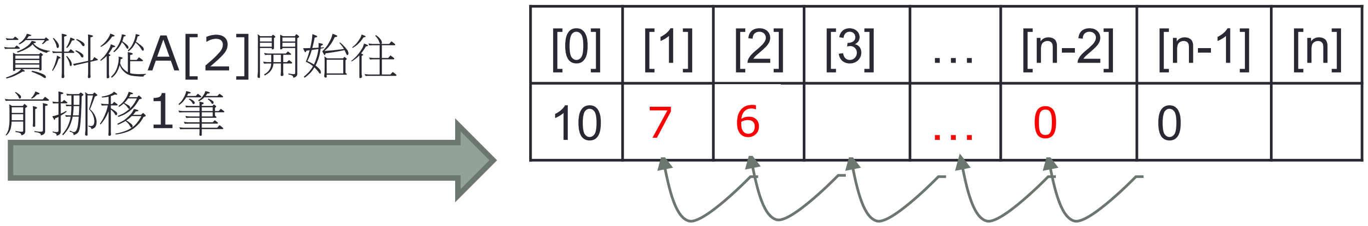
```
//A共有n筆資料，插入資料到A[i]
for(int j=n; j >i; j--) A[j] = A[j-1]; //資料往後挪以便空出A[i]
A[i] = newdata;
n++; //增加一筆資料
```

挪移陣列裡資料以便刪除一個陣列元素

欲刪除A[1]，並保留剩餘n-1筆資料維持原本順序並存放於A[0],A[1],…,A[n-1]

[0]	[1]	[2]	[3]	…	[n-2]	[n-1]	[n]
10	8	7	6	…	1	0	

資料從A[2]開始往前挪移1筆



[0]	[1]	[2]	[3]	…	[n-2]	[n-1]	[n]
10	7	6		…	0	0	

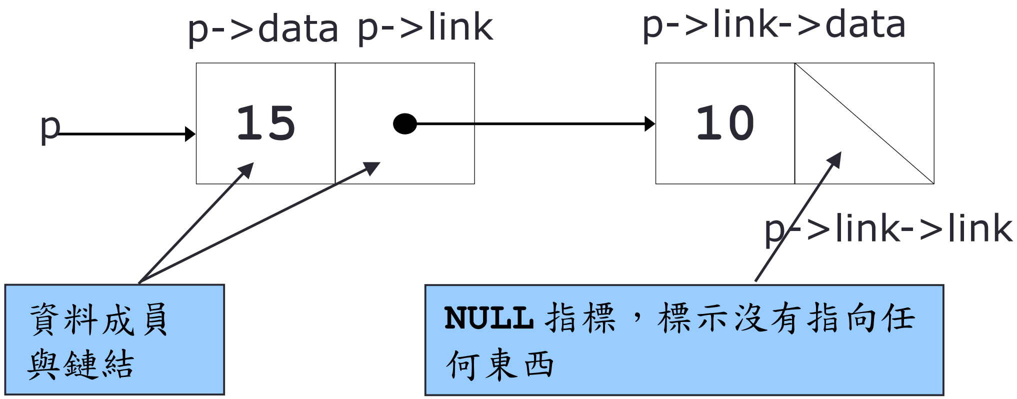
```
n--;
for(int j=i; j<n; j++) {
   A[j] = A[j+1];
}
```

```
//A共有n筆資料，欲刪除A[i]
n--;
//將資料從A[i+1]起，往前挪移一筆
for(int j=i; j < n; j++)  A[j] = A[j+1];
```

自我參照結構

- 結構的成員含指向自己這個結構型態的物件的指標變數

```
struct node {
   int data;
   struct node *link;
}
```



  - **data**: 資料成員
  - **link**: 鍊結成員
    - 指向一個同樣型態的物件。
    - 稱作鍊結。透過鍊結可建立一個節點與另一個節點的關聯。
    - p->data 為 15
    - p->link->data 為 10
    - p->link->link為 NULL

- 可以透過這個指標變數將這個型態的物件組織在一起。可以組織成鍊結串列(linked list)，樹(tree)，圖(graph) 。

動態配置記憶體 (Dynamic memory allocation)

- 動態配置記憶體函式void* **malloc(int size)**
  - 動態配置size bytes的記憶體
    - 使用sizeof
  - 回傳的型態為 **void ***
    - 如果空間不夠, returns **NULL**
  - 範例

    struct node *newPtr =(struct node*) malloc(sizeof(struct node));

- 記憶體釋放函式 **free**
  - **free (newPtr);**
    - **newPtr指向你用malloc要到**記憶體空間

  - 注意當newPtr為NULL或非malloc所得到的位置，free可能會當掉。

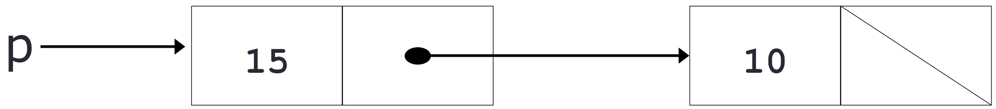
鍊結串列(Linked Lists)

- 將數個自我參照結構變數(稱節點node)透過鍊結成員將這些結點串在一起。

```
struct node {
   int data;
   struct node *link; //單向鍊結
}
```

```
struct node {
   int data;
   struct node *llink, *rlink; //雙向鍊結
}
```

- 當資料適合組織成一個循序的順序，並且時常改變資料局部結點關係時，就會考慮使用鍊結串列。

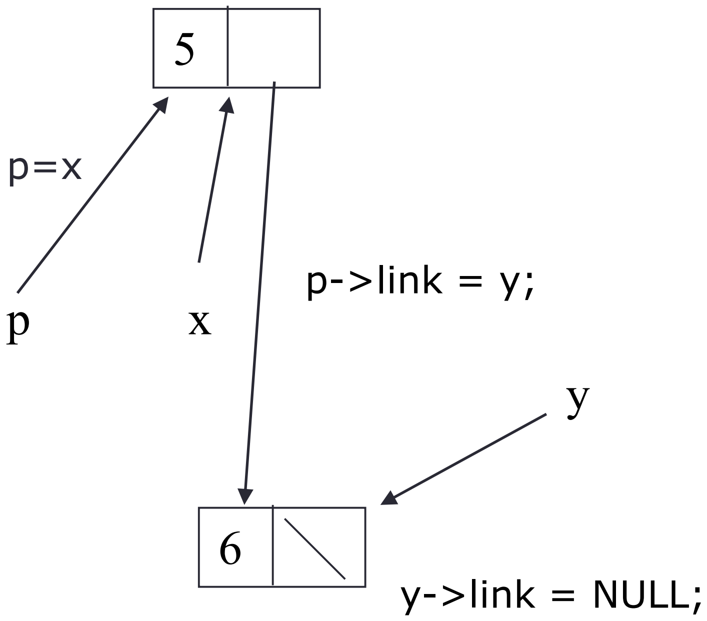


```
p = (struct node*) malloc(sizeof(struct node));
p->data = 15;
p->link = (struct node*) malloc(sizeof(struct node));
p->link->data = 10;
p->link->link  = NULL;
```

```
struct node {
int data;
struct node* link
} *p, *x, *y;
```

```
x = (struct node*) malloc(sizeof(struct node));
x->data = 5;
p         = x;
y = (struct node*) malloc(sizeof(struct node));
y->data = 6;
y->link = NULL;
p->link = y;
```

```
x = (struct node*) malloc(sizeof(struct node));
x->data = 5;
```

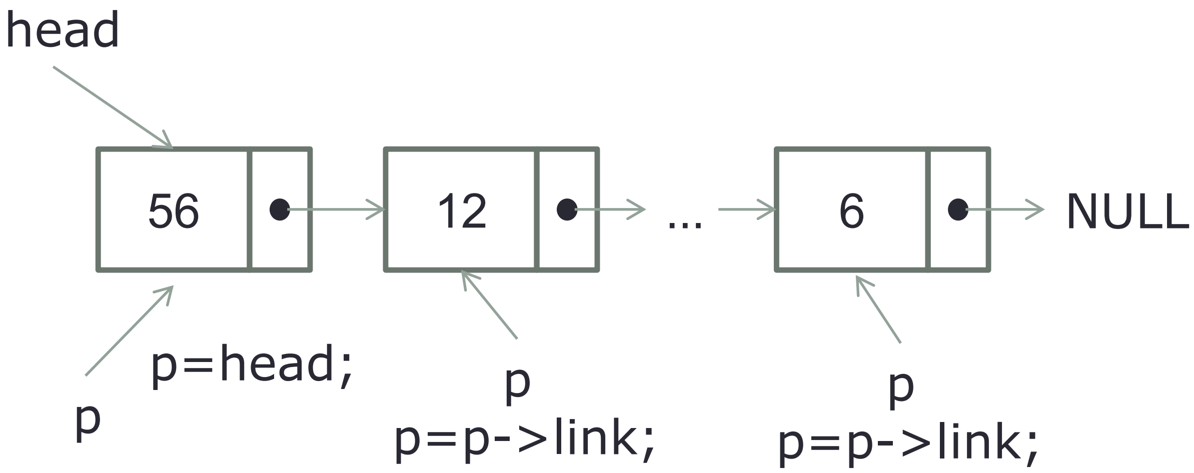


```
y = (struct node*) malloc(sizeof(struct node));
y->data = 6;
```

```
printf(“%d\n”,p->data); //輸出5
printf(“%d\n”,p->link->data); //輸出6
```

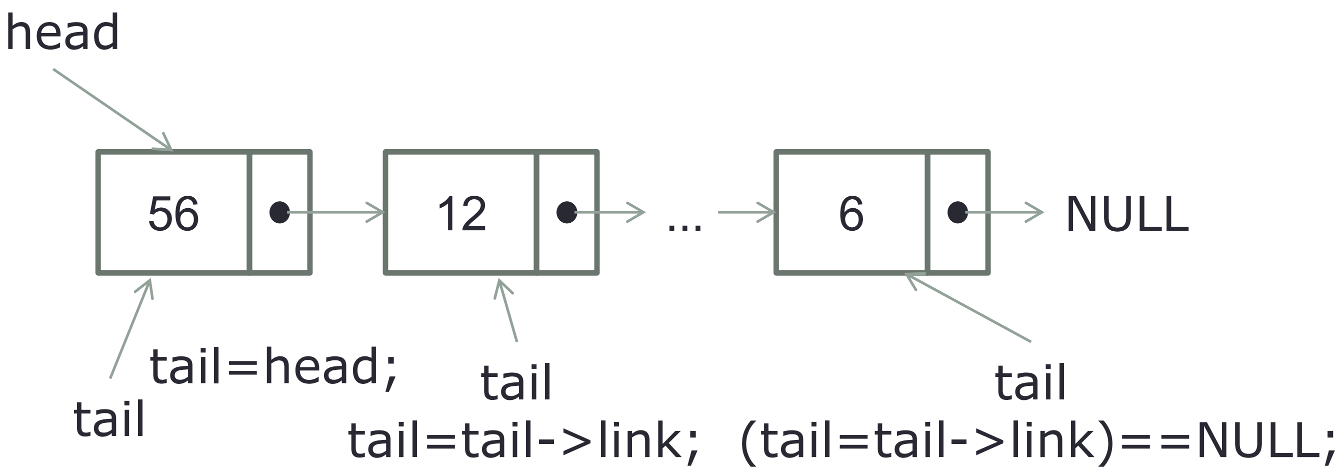
在鏈結串列上移動

```
for(p = head; p != NULL; p = p->link) {
    printf("%d\n", p->data);
}
```



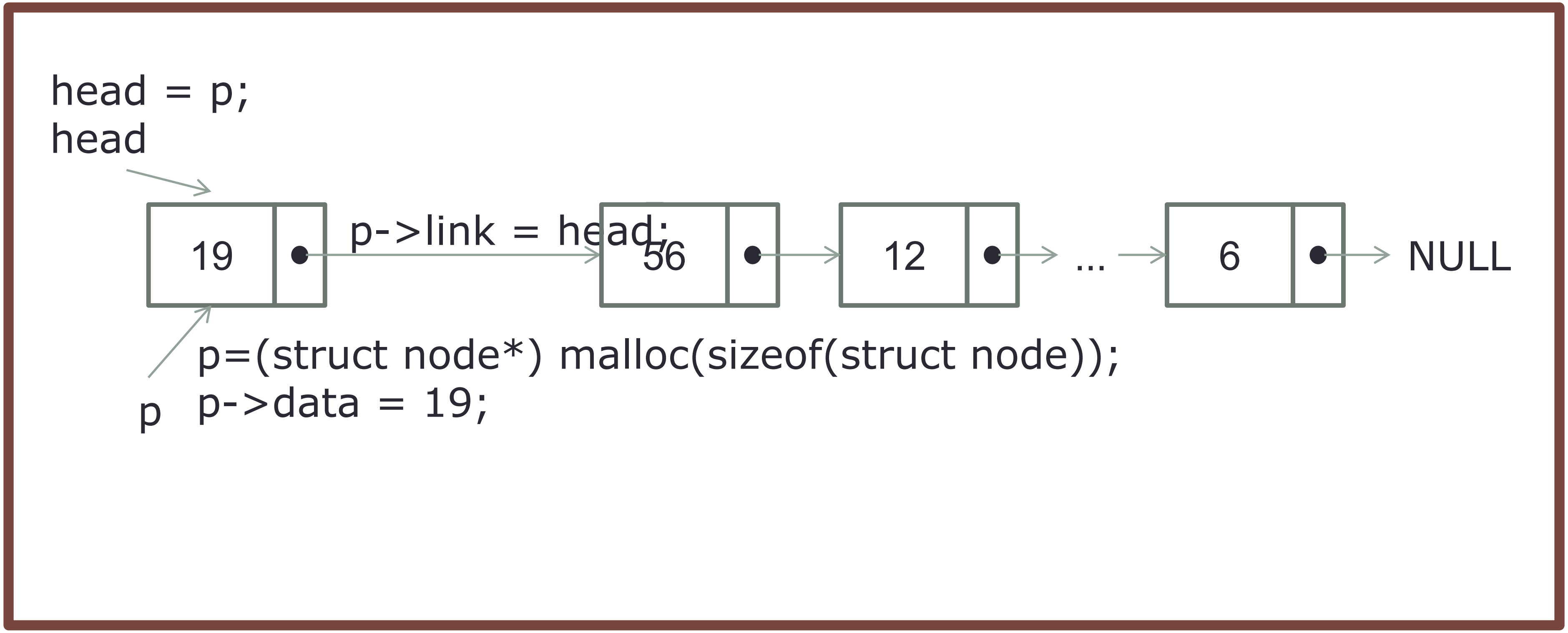
找到單向鍊結串列尾端節點

```
if (head == NULL) {
  tail = NULL;
} else {
  for(tail=head; tail->link != NULL; tail = tail->link);
}
```



插入結點至鏈結串列前端

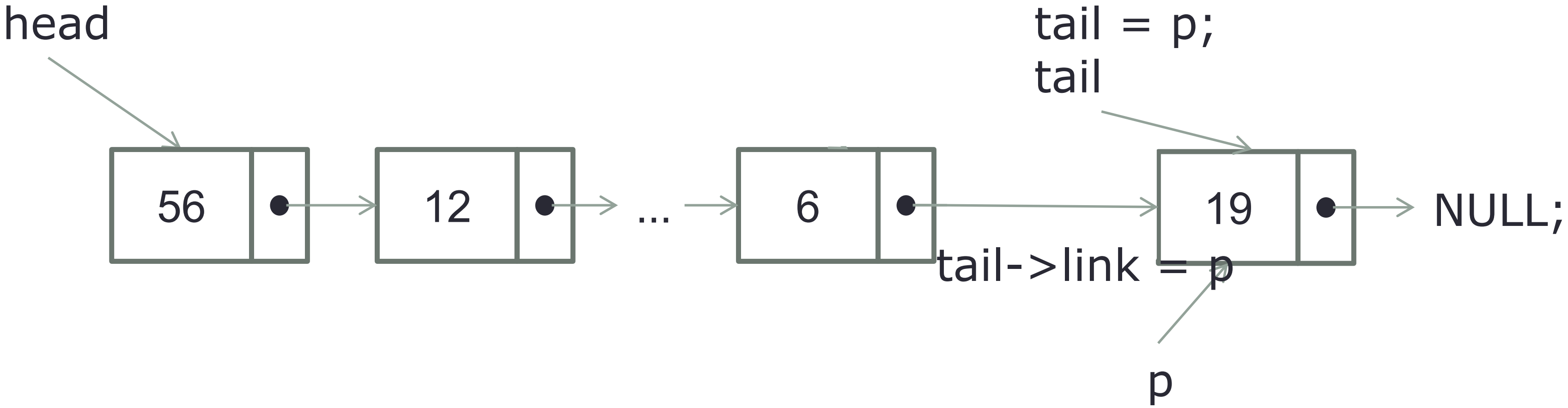
```
struct node* p = (struct node*) malloc(sizeof(struct node));
p->data = data;
p->link  = head;
head     = p;
```



串列插入一個元素只需改變少數鍊結

插入結點於鏈結串列尾端

```
struct node* p = (struct node*) malloc(sizeof(struct node));
p->data = data;
p->link  = NULL;
tail->link= p;
tail       = p;
```

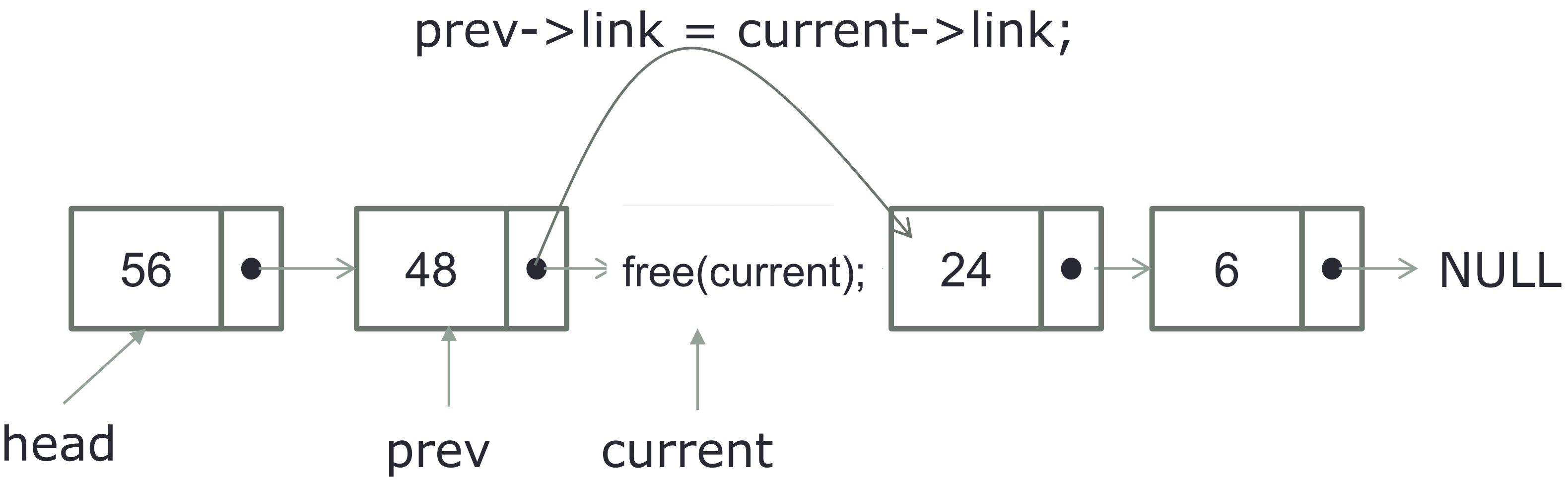


```
p=(struct node*) malloc(sizeof(struct node
p->data = 19;
p->link = NULL;
```

串列插入一個元素只需改變少數鍊結

於鏈結串列刪除一個節點

```
prev->link = current->link;
free(current);
```



```
if (prev==NULL) {//欲刪除的節點為第一個節點
    head = head->link;
    free(current);
} else {{//欲刪除的節點非第一個節點
    prev->link = current->link;
    free(current);
}
```

串列刪除一個元素只需改變少數鍊結