C Language

point

- ***point : 一種變數或物件，它裡面存的不是一般的數值，而是記憶體位置***

```c
#include <stdio.h>

void func(int *ptr){
    (*ptr)++;
}

int main(){
    int num = 5;
    func(&num);
    printf("num:%d\n",num);
    return 0;
}

// ptr++: 它將指標 ptr 的值增加，使其指向下一個記憶體位置。

// (*ptr)++: 使用 *ptr 解引用指標，取得指標所指向的值，然後對該值進行遞增操作 ++。
```

- ***解釋以下指標意義***

```c
int a;
int *a;
int **a;
int a[10];
int *a[10];
int [*a](10);
int (*a)(int);
int (*a[10])(int);

// 一個整數型別
// 一個指向整數的指標
// 一個指向指標的指標，而"指向的指標"是指向一個整數型別
// 一個有10個整數型的陣列
// 一個有10個指標的陣列，該指標是指向一個整數型別
// 一個指向有10個整數型陣列的指標
// 一個指向函數的指標，該函數有一個整數型參數並返回一個整數
// 一個有10個指標的陣列，該指標指向一個函數，該函數有一個整數型參數並返回一個整數
```

- ***寫一個function讓變數a跟b能夠交換，不透過暫存變數***

```c
#include <stdio.h>

void swap(int *a, int *b) {
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}

int main() {
    int a = 5;
    int b = 10;
    printf("Before: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After: a = %d, b = %d\n", a, b);
    return 0;
}
```

- ***回答printf的答案***

```c
#include <stdio.h>

int main(void) {
    char *str[] = {
        {"MediaTekOnlineTesting"},
        {"WelcomeToHere"},
        {"Hello"},
        {"EverydayGenius"},
        {"HopeEverythingGood"}
    };
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    char* m = str[4] + 4;
    char* n = str[1];
    char* p = *(str+2) + 1;
    int *q = &(a + 1)[3];
    printf("1. %s\n", *(str+1));
    printf("2. %s\n", (str[3]+8));
    printf("3. %c\n", *m);
    printf("4. %c\n", *(n+3));
    printf("5. %c\n", *p + 1);
    printf("6. %d\n", *q);
    return 0;
}
// 1. WelcomeToHere
// 2. Genius
// 3. E
// 4. c
// 5. f
// 6. 5
```

- ***回答陣列的答案***

```c
#include <stdio.h>

int main() {
    int a[] = {6, 7, 8, 9, 10};
    int *p = a;
    *(p++) += 100;
    *(++p) += 50;
    for (int i = 0; i < 5; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
// a[0] = 106
// a[1] = 7
// a[2] = 58
// a[3] = 9
// a[4] = 10
```

- ***different between pointer and array (memory)***

就記憶體方向來看，指標所用的記憶體位置不為連續，而陣列所配置的空間為連續。

call by value, call by reference, call by address

- ***call by value : 當將一個變數作為參數傳遞給一個函式時，函式接收的是該變數的副本，而不是原始變數本身(C 只有 call by value)***

```
#include <stdio.h>

int swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main(void)
{
    int a = 5;
    int b = 10;
    swap(a, b);
    printf(" a = %d b = %d", a, b); // a = 5,b = 10
}

// 數值沒交換是因為在 C 語言中，默認的參數傳遞是call by value，函數內部改變的參數並不能影響到函數外的變
```

- ***call by reference : 當將一個變數作為參數傳遞給一個函式時，函式接收的是該變數的記憶體位址(c++才有)***

```cpp
#include <cstdio>

void swap(int &c, int &d)
{
    int temp = c;
    c = d;
    d = temp;
}
int main()
{
    int a = 5, b = 10;
    swap(a, b);
    printf(" %d %d ", a, b);
    return 0;
}

// 在 C 語言中沒有Call by reference,故編譯時會有錯誤
```

- ***call by address：當將一個變數作為參數傳遞給一個函式時，函式接收的是該變數的記憶體位址***

```cpp
#include <stdio.h>

int swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void)
{
    int a = 5;
    int b = 10;
    swap(&a, &b);
    printf(" a = %d b = %d", a, b);// a = 10, b = 5
}

// 透過指標去傳遞位址(Call by address)，可將函數內的變數作為指向該變數的指標傳遞給函數，從而讓函數內的變
```

variable scope and lifetime

- ***local：僅活在該函式內，存放位置在 stack 或 heap 記憶體中***

```c
#include <stdio.h>

void count()
{
    int c = 1;
    printf("%d\n", c); // c = 1 c = 1...
    c++;
}

int main(void)
{
    for(int i = 0; i < 10; i++)
        count();
    return 0;
}
```

- ***static：生命周期跟程式執行期間一樣長，而範圍 (scope) 則維持不變，即在宣告的函式之外仍無法存取此變數***

```c
#include <stdio.h>

void count()
{
    static int c = 1;
    printf("%d\n", c); // c = 1 c = 2...
    c++;
}

int main(void)
{
    for(int i = 0; i < 10; i++)
        count();
    return 0;
}
```

- ***global：所有區域皆可使用此變數***

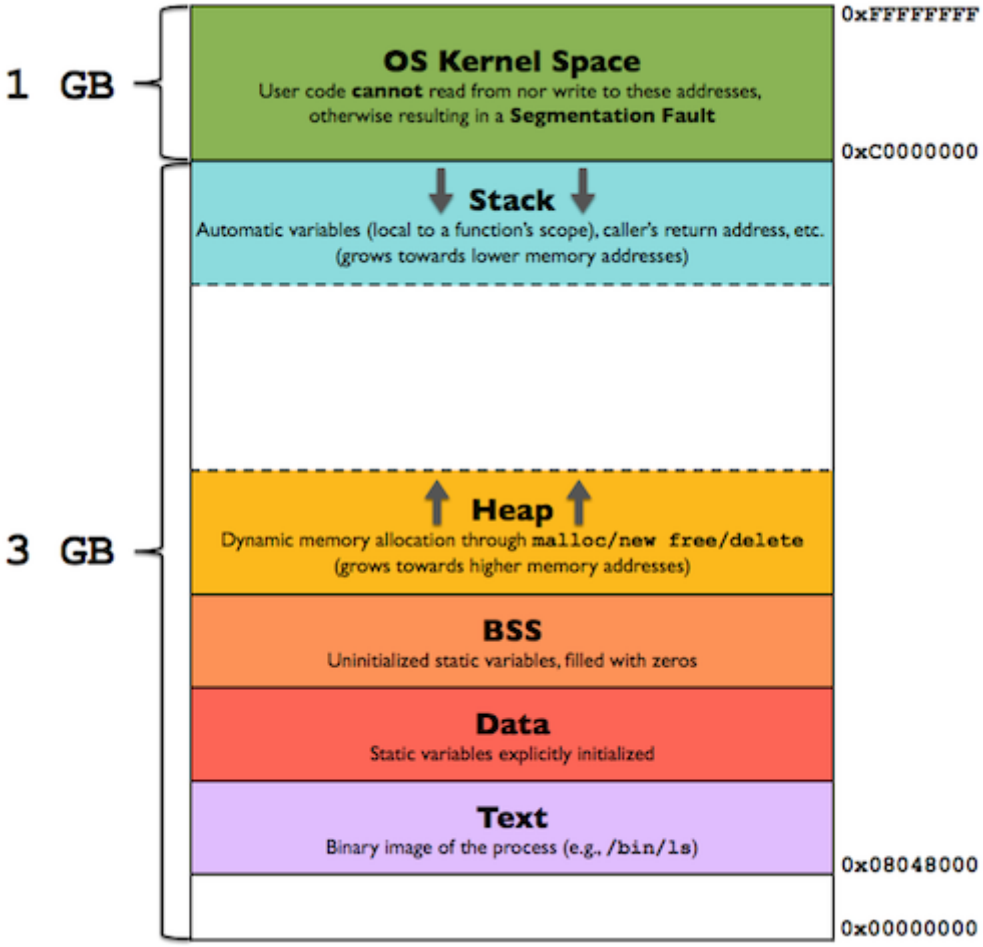
```c
#include <stdio.h>

int c = 1;

void count()
{
    printf("%d\n", c); // c = 1 c = 2...
    c++;
}

int main(void)
{
    for(int i = 0; i < 10; i++)
        count();
    return 0;
}
```

- ***記憶體的配置***

Stack ： 存放函數的參數、區域變數等，由空間配置系統自行產生與回收，會稱作 stack 是由於其配置遵守 LIFO（L

Heap ： 一般由程式設計師分配釋放，執行時才會知道配置大小，如 malloc/new 和 free/delete

Global ： 包含 BSS（未初始化的靜態變數）、data section（全域變數、靜態變數）和 text/code（常數字元）



extern

- ***extern：用於聲明外部變數，如果變數需要在多個文件中共享，就需要使用 extern 聲明該變數；如果變數只在當前文件中使用，可以不使用 extern 關鍵字聲明***

```
// var.c
int a = 10;


// main.c
#include "var.c"

extern int a;

int main() {
    printf("a = %d\n", a); //a = 10
    return 0;
}
```

const

- ***const : 通常表示只可讀取不可寫入的變數，常用來宣告常數，特性如下***

```
#include <stdio.h>

int main(void)
{
    int n1 = 10, n2 = 100;

    const int* ptr1 = &n1; // ptr1是一個指向const的整數指標
    *ptr1 = 20; // X
    ptr1 = &n2; // O (ptr1 = 100)

    int* const ptr2 = &n1; // ptr2是一個指向整數的const指標
    *ptr2 = 20; // O (ptr2 = 20)
    ptr2 = &n2; // X
}
```

volatile

- ***volatile : 編譯器不對 volatile 修飾的變數進行最佳化處理,而是每次都去讀取變數實際上最新的數值，應用如下***

1. 修飾中斷處理程式中(ISR)中可能被修改的全域變數
2. 修飾多執行緒(multi-threaded)的全域變數
3. 設備的硬體暫存器(如狀態暫存器)

```
extern const volatile unsigned int rt_clock;
// 這是在 RTOS kernel 常見的一種宣告：rt_clock通常是指系統時鐘，它經常被時鐘中斷進行更新。所以它是volat
```

inline

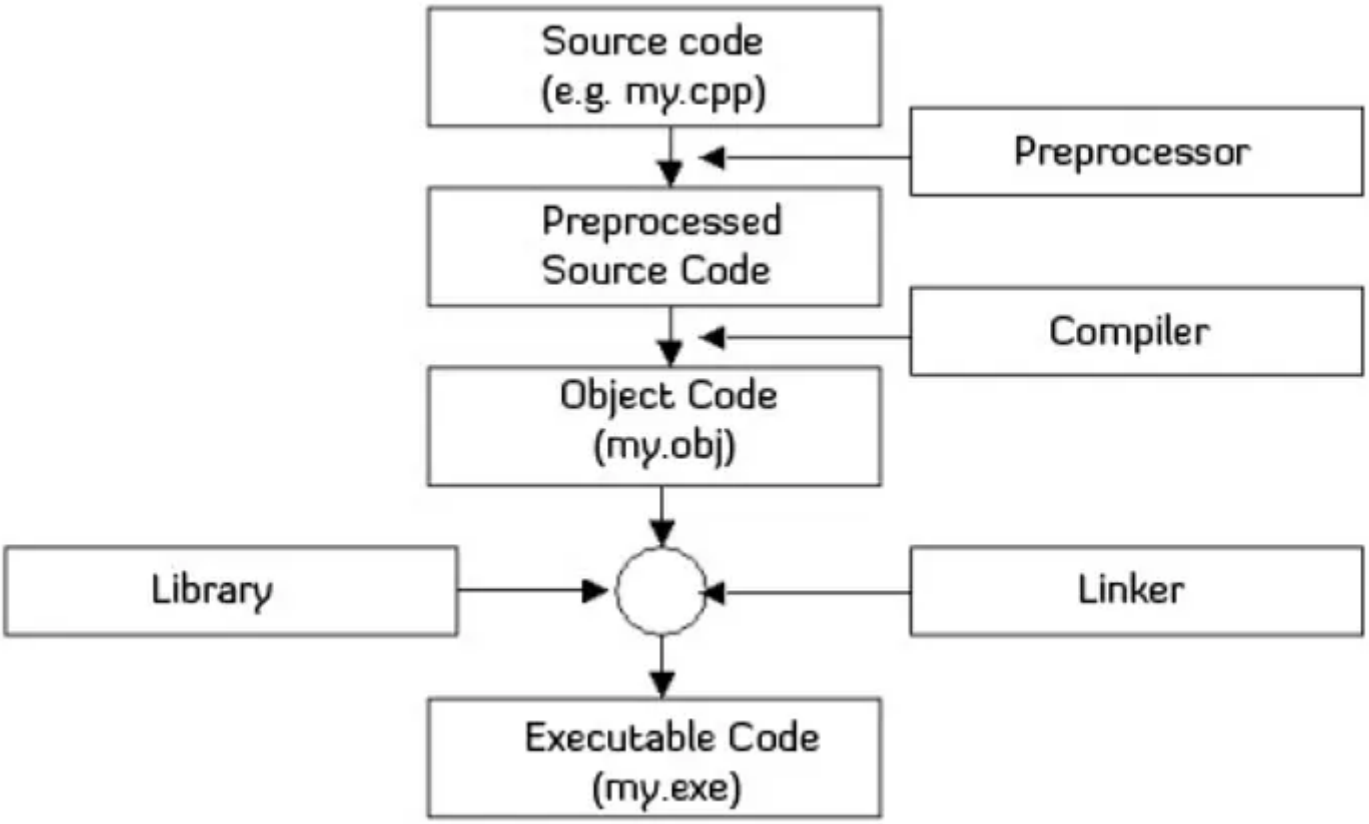
- ***inline寫法 :***

```
inline int square(int x)
  return x * x;
```

- ***Macro寫法 :***

```
# define SQUARE(x) ((x) * (x))
printf("\n %d", SQUARE(5)); // 若在主程式中，下述能得到 25，看起來沒有問題
printf("\n %d", SQUARE(3+2)); // 但如果是以下，卻會得到 11 (3+2 * 3+2)
```

- ***差異：***
  - macro：在前置處理器(preprocessor)階段時,直接進行文字替換
  - inline：在編譯(compile)階段時,直接取代function



#define

- ***#define：是巨集的一種，在前置處理器(preprocessor)執行時處理，將要替換的程式碼展開做文字替換。define 語法範例如下：***

```
#define PI 3.1415926
#define A(x) x
#define MIN(A,B)  ((A) <= (B) ? (A) : (B))
```

- ***引入防護和條件編譯：防範 #include 指令重複引入的問題***

```
#ifndef MYHEADER // 避免重複引入
#define MYHEADER
...
#endif
```

different between interrupt and polling

- ***interrupt : 具即時性，當中斷觸發時，處理器會暫停目前處理的任務，轉而去執行中斷相關程序，等到處理完畢時才會回到原本的任務上***
- ***polling : 它是一種定時檢查的方式，當檢查到有事件發生時才會去執行它***

struct

- ***struct : 結構是一種使用者自定的型態，它可將不同的資料型態串在一起***

```
#include <stdio.h>

struct student{
  int id;
  char name[10];
};

int main(void) {
  student john = {291, {'j', 'o', 'h', 'n', '\0'}};
  printf("學號: %d\n", john.id);
  printf("姓名: %s\n", john.name);
  return 0;
}
```

- ***struct佔幾個byte***

```
typedef struct MyStruct
{
    char a[2];
    int  b;
    double c;
    int *Pint;
    char d;
    char*Pchar;
};
//ans = 1+3(對齊int)+4+8+4+1+3+4(由於(4+1+3+4)不是8的倍數故需要補4)+4=32byte

typedef struct MyStruct
{
    char a;
    char b;
    struct str2
    {
        int d;
    } c;
};
//ans = 1+1+4+2(對齊int)=8byte

typedef struct MyStruct
{
    char a;
    int b;
    char c;
    char* d;
    double* e;
    struct str2
    {
        int f;
        char g;
        struct str3
        {
            char* p;
        }n;
    } m;
};
//ans = 1+3+4+1+3+4+4+4+1+3+4=32byte

typedef struct MyStruct
{
    char a;
    char b;
    struct str2
    {
        char c;
        char d;
    };
```

```
};
//ans = 1+1=2byte  *並沒有申明這個結構體的變數所以str2不用計算
```

union

- ***union：在語法結構上，union與 struct 類似，都是使用者自定義的資料結構，最大差異在於 union 結構中的各變數是共用記憶體位置，並且在任何時候，只有一個變數的值是有效的，這取決於最後一次賦值的變數***

```
union myUnion {
    int i;
    float f;
    char c;
};

int main() {
    union myUnion u;
    u.i = 42;
    printf("u.i = %d\n", u.i);  // output : u.i = 42
    u.f = 3.14;
    printf("u.f = %f\n", u.f);  // output : u.f = 3.140000
    u.c = 'A';
    printf("u.c = %c\n", u.c);  // output : u.c = A
    printf("u.i = %d\n", u.i);  // output : u.i = 1092616192
    //在對 f 和 c 進行賦值之後，u.i 的值也發生了變化，這是因為 i、f 和 c 共享同一塊記憶體
    return 0;
}
```

enum

- ***enum：是一種常數定義方式，可以提升可讀性，enum 裡的識別字會以 int 的型態，從指定的值開始遞增排列 (預設為 0)***

```
enum color1 {
    red,
    green,
    blue,
    yellow
};
// red = 0, green = 1...

enum color2 {
    red=10,
    green,
    blue=20,
    yellow
};
//red = 10, green = 11, blue = 20, yellow = 21
```

sizeof

- ***sizeof : 各型別大小***

Type	64-bit	32-bit
string	8	4
char	1	1
point	8	4
short	2	2
int	4	4
long	8	4
long long	8	8
size_t	8	4
double	8	8
long double	16	12

- ***二維陣列大小***

```c
#include <stdio.h>
int main()
{
    int arr[3][4];
    printf("1:%d\n", sizeof(arr));  // 結果為 48 (3 * 4 * 4(int))
    char str[2][10];
    printf("2:%d\n", sizeof(str));  // 結果為 20 (2 * 10 * 1(char))
    int* ptr[5][3];
    printf("3:%d\n", sizeof(ptr));  // 結果為 60 (5 * 3 * 4(int*))
    struct Point
    {
        int x;
        int y;
    };
    struct Point points[4][3];
    printf("4:%d\n", sizeof(points));  // 結果為 96 (4 * 3 * 8(int+int=8))
    return 0;
}
```

bit operation

- ***setting a bit***

```c
int set_bit(int x, int n)
    return x | (1 << n);
```

- ***clearing a bit***

```c
int clear_bit(int x, int n)
    return x & ~(1 << n);
```

- ***fliping a bit***

```c
int flip_bit(int x, int n)
    return x ^ (1 << n);
```

- ***checking a bit***

```c
int check_bit(int x, int n)
    return (x >> n) & 1;
```

- ***回答ans的數值***

```c
#include <stdio.h>

int main() {
    long ans = 0;
    short a = 0x1234;
    short b = 0x5600;
    ans += a << 16;
    ans += b << 0;
    ans += (b >> 2) + 0x22;
    printf("a=%x\n",ans);   // 12346BA2
}
```

bubbleSort

- ***bubbleSort版本1：***

```c
#include <stdio.h>

void bubbleSort(int arr[],int n){
    for(int i=0; i<n-1 ;i++){
        for(int j=i+1; j<n; j++){
            if(arr[i] > arr[j]){
                int temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }
        }
    }
}

int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    for (int i=0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

- ***bubbleSort版本2：***

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 7, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

quick sort

```c
#include <stdio.h>

void quick_sort(int data[], int left, int right)
{
    if (left >= right) {
        return 0;
    }

    int l = left;
    int r = right;
    int key = data[left];

    while (l != r) {
        while (data[r] > key && l < r) {
            r--;
        }
        while (data[l] <= key && l < r) {
            l++;
        }
        if (l < r) {
            int temp = data[l];
            data[l] = data[r];
            data[r] = temp;
        }
    }

    data[left] = data[l];
    data[l] = key;

    quick_sort(data, left, l - 1);
    quick_sort(data, l + 1, right);
}

int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    quick_sort(arr, 0, n-1);
    for (int i=0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

設定一個絕對位址為0x67a9的整數型變數的值為0xaa55

```c
#include <stdio.h>

int main(void)
{
    int *ptr = (int *)0x67a9;
    printf("p1=%x\n", ptr); // p1=67a9
    ptr = (int *)0xaa55;
    printf("p2=%x\n", ptr); // p2=aa55
}
```

N是否為判斷2的次方

```c
int isPowerof2(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

連續呼叫 func 10 次，印出的值為何?

```c
#include <stdio.h>

void func(void)
{
    static int i = 0 ;  //若沒static, i會一直歸零
    i++ ;
    printf("%d" , i ) ;
}

int main(void)
{
    for(int i=0; i<10; i++)
        func(); // 12345678910
}
```

i++ & ++i

```c
#include <stdio.h>

int main()
{
    int i = 5;
    int j = i++;
    printf("i1 = %d\n",i);
    printf("j1 = %d\n",j);

    i = 5;
    j = ++i;
    printf("i2 = %d\n",i);
    printf("j2 = %d\n",j);
    return 0;
    //ans : 6 5 6 6
}
```

寫個function判斷基數偶數

```c
if (num % 2 == 0) {
    printf("even\n");
} else {
    printf("odd\n");
}
```

寫個function計算有幾個位元是 1

```c
int func(int x){
    int sum=0;
    while(x){
        x &= x-1;
        sum++;
    }
    return sum;
}
```

以下define與typedef的用法誰較佳

```
#define dPS struct s *
dPS p1, p2;
struct s * p1, p2;
// p1為一個指向結構s的指標，p2為一個實際的結構s。

typedef struct s * tPS;
tPS p3, p4;
struct s * p3;
struct s * p4;
// p3/4為一個指向結構s的指標

//Ans ： typedef
```

What is the output of the following program

```
int main() {
    unsigned int a = 6;
    int b = -20;
    (a + b > 6) ? puts(">6") : puts("<= 6");
}

// 當表達式同時存在有符號類型和無符號類型時皆都自動轉換為無符號類型。
// 因此-20變成了一個非常大的正整數，所以該表達式計算出的結果大於6。
```

The faster way to an integer multiply by 7

```
int main() {
    int i = 1;
    i = (i << 2) + (i << 1) + (i << 0);
    printf("i = %d\n",i);
}
```

declaration（宣告）和 definition（定義）的差異

- ***差異：宣告是告訴編譯器一個變數的名稱和類型，但不分配存儲空間，而定義則會分配儲存空間***

```
int num; //宣告

num = 10; //定義
```

- ***(每個變數和函數只能有一個定義，但可以有多個宣告)***

```
// 檔案1.c
int globalVar; // 宣告外部變數

// 檔案2.c
extern int globalVar; // 外部變數的宣告
globalVar = 100; // 定義外部變數的值為100
```

Reverse a string

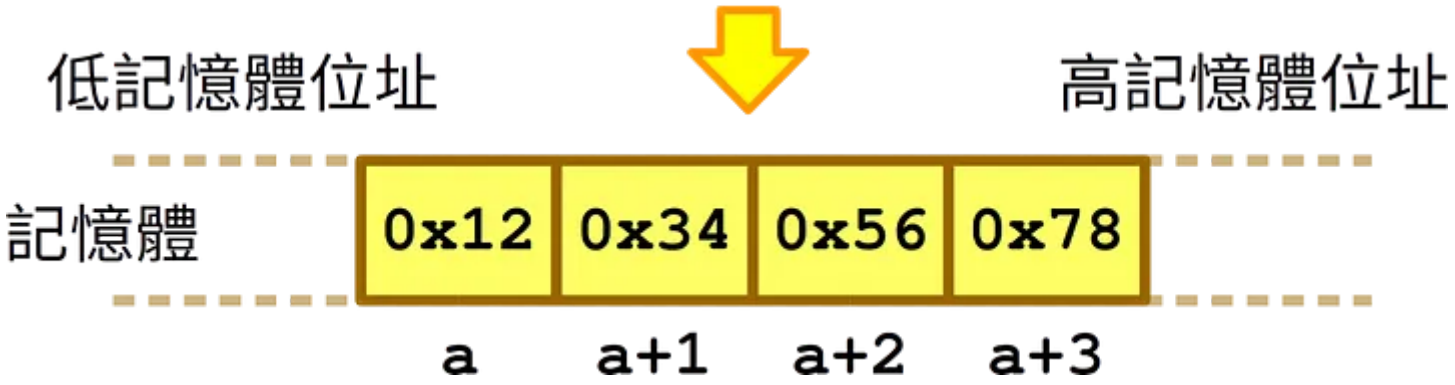
```
#include <stdio.h>
#include <string.h>

void reverseString(char *str) {
    int left = 0;
    int right = strlen(str) - 1;
    while (left < right) {
        char temp = str[left];
        str[left] = str[right];
        str[right] = temp;
        left++;
        right--;
    }
}

int main() {
    char str[] = "ABCDE";
    reverseString(str);
    printf("反轉後字串: %s\n", str);
    return 0;
}
```

判斷Big-Endian or Little-Endian

- ***Big / Little-Endian：他們是CPU中兩種不同位元組排序***
  Big-Endian：最高位的位元組會放在最低的記憶體位址上
  Little-Endian：最高位的位元組會放在最高的記憶體位址上

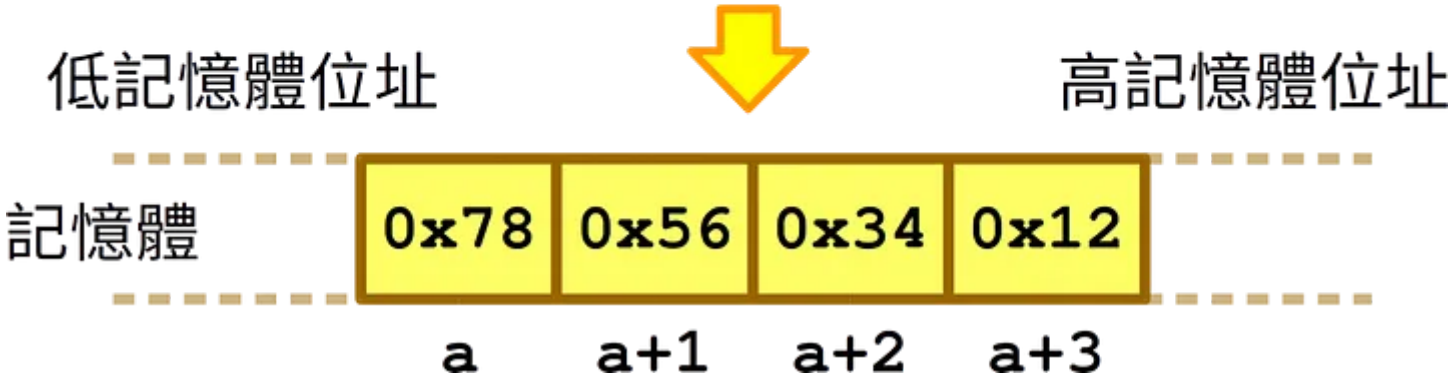
Big-Endian

0x12345678



Little-Endian

0x12345678



```c
#include <stdio.h>
int main() {
    typedef union {
        unsigned int i;
        unsigned char c[4];
    } EndianTest;
    EndianTest t;
    t.i = 0x12345678;
    if(t.c[0] == 0x12 && t.c[1] == 0x34 && t.c[2] == 0x56 && t.c[3] == 0x78){
        printf("Big Endian!!");
    }else if(t.c[0] == 0x78 && t.c[1] == 0x56 && t.c[2] == 0x34 && t.c[3] == 0x12){
        printf("Little Endian!!");
    }else{
        printf("Other Endian!!");
    }

    return 0;
}
// 需要用union的原因是因為他們共用同一個記憶體位置,而struct會因記憶體對齊可能導致錯誤
```

給一個int a[20]已排序的陣列，請寫一個function(a, size)能印出0~500的數字，且不包含a陣列內的元素

```c
#include <stdio.h>

void function(int *a,int size)
{
 for(int i=0; i<=500; i++)
 {
    if(i == *a)
        a++;
    else
        printf("%d\n",i);
    }
}

int main() {
    int nums[20] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,20};
    int size = 500;
    function(nums,size);
    return 0;
}
```

給一個int a[20]已排序的陣列，請寫一個function(a, size, b) 能依照參數b(b = 0~4)別印出該區間的數字，且不包含a陣列內的元素，例如 b =0, 印出0~99 b = 1, 印出100~199

```c
void function(int *a, int size, int b)
{
    int *ptr = a;
    int i;

    while (*ptr < b * 100) {
        ptr++;
    }

    for (i = b*100; i<(b+1)*100; i++) {
        if (*ptr == i)
             ptr++;
        else
            printf("%d\n", i);
    }
}

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

印出下列圖形

```
    *
   **
  ***
 ****
*****
```

```c
#include <stdio.h>

int main() {
    for(int i=0 ;i<5; i++){
        for(int space=0; space<4-i; space++){
            printf(" ");
        }
        for(int star=0; star<=i; star++){
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

費式數列，寫一個函數，輸入值是位置的值"n"，要找出相對應的值

- 一般解法

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1)
        return n;

    int prev = 0;
    int current = 1;
    int next;

    for (int i = 2; i <= n; i++) {
        next = prev + current;
        prev = current;
        current = next;
    }

    return current;
}

int main() {
    int n;
    printf("input:");
    scanf("%d",&n);
    printf("%d\n",function(n));
    return 0;
}
```

- ***遞迴解法***

```c
int function(int n) {
  if(n == 0){
    return 0;
  }
  if(n == 1){
    return 1;
  }

  return function(n - 1) + function(n - 2);
}

int main() {
    int n;
    printf("input:");
    scanf("%d",&n);
    printf("%d\n",function(n));
    return 0;
}
```

binary search

```c
#include <stdio.h>

int binary_search(int arr[], int left, int right, int target){
    while(left <= right){
        int mid = left + (right - left)/2;
        if(arr[mid] == target)
            return mid;
        else if(arr[mid] > target)
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}

int main(){
    int arr[] ={1,2,3,5,7,9,12,18,22};
    int n = sizeof(arr)/sizeof(arr[0]);
    int result = binary_search(arr,0,n-1,2);
    printf("res=%d\n",result);
    return 0;
}
// 考虑以下情况：

// 如果直接使用 mid = (right - left) / 2，那么 mid 的范围将始终是从 0 到 (right - left) / 2。
// 但实际上，我们希望 mid 的范围是从 left 到 right。
// 因此，为了保持 mid 在正确的范围内，我们需要加上 left 的偏移量，即 mid = left + (right - left) / 2。

// 这样，mid 的计算结果将是一个介于 left 和 right 之间的值，确保每次迭代时都在正确的搜索范围内进行比较，
```

給一個unsigned short, 問換算成16進制後,四個值是否相同? 若是回傳1,否則回傳0

```
int function(unsigned short num) {
    unsigned short temp[4];
    temp[0] = (num&0xF000) >> 12;
    temp[1] = (num&0x0F00) >> 8;
    temp[2] = (num&0x00F0) >> 4;
    temp[3] = num&0x000F;
    if((temp[0] ^ temp[1] ^ temp[2] ^ temp[3]) == 0){
        return 1;
    }
    else{
        return 0;
    }
}

int main() {
    unsigned short num = 0xAAAA;
    printf("ans:%d\n",function(num));
    return 0;
}
```

求一個數的最高位1在第幾位

```
#include <stdio.h>

int function(int num){
    int cnt = 0;
    while(num){
        if(num&1 == 1){
            cnt++;
        }
        num >>= 1;
    }
    return cnt-1;
}

int main() {
    int num = 15;
    printf("ans:%d\n",function(num));
    return 0;
}
```

最大公因數 遞迴寫法

```c
#include <stdio.h>

int gcd(int a, int b) {
    if(b == 0)
        return a;
    return gcd(b,a%b);
}

int main() {
    printf("GCD: %d\n", gcd(15,5));

    return 0;
}
```

0~500個數字每次隨機 取一個數字出來，但下次在抽出時不可以出現已經抽過的數字，問你如何時實現。

- ***一般解法***

```c
#include <stdio.h>
#include <stdlib.h>

#define NUM 501

void func(int *nums){
    int count = 0;
    while(count < NUM){
        int temp = rand() % NUM;
        if(nums[temp] == 0){
            nums[temp] = 1;
            count++;
            printf("%d ",temp);
        }
    }
}

int main(){
    int nums[NUM];
    for(int i=0; i<NUM; i++){
        nums[i] = 0;
    }
    func(nums);
    return 0;
}
```

- ***進階解法***

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int nums[501];
    for(int i=0; i<501; i++){
        nums[i] = i;
    }
    int final = 500;
    int choose;
    while(final >= 0){
        choose = rand()%(final+1);
        printf("%d ",nums[choose]);
        swap(&nums[choose], &nums[final]);
        final--;
    }
}
```

請問以下MIN()的結果為何?

```c
#define MIN(a,b) (a < b ? a : b)
int result = 2 * MIN(6,10);
// return 10
```

#error

- ***#error : 指令會讓預處理器產生一個錯誤消息，並停止編譯。它通常被用來在預處理時檢測錯誤或者不符合要求的條件，比如檢測程序是否被正確地編譯、是否使用了正確的編譯選項、是否定義了需要的marco等等***

```c
#ifndef DEBUG
#error "You must define the DEBUG macro"
#endif
```

Explain lvalue and rvalue

- ***lvalue：左值通常指的是運算式後還保留其狀態的一個物件，通常指的是所有的變數都是左值***
  rvalue：右值通常指的是一個運算式過後其狀態就不會被保留了，也就是一個暫時存在的數值

```
int a = 5;  // a 是 lvalue,5 是 rvalue
```

印出菱形

```c
#include<stdio.h>
#include<stdlib.h>

void diamond()
{
    int i,input,star,space,mid;
    printf("input:");
    scanf("%d",&input);
    mid = (input/2) + 1;
    for(i=1; i<=mid; i++){
        for(space=i; space<mid; space++)    printf(" ");
        for(star=1; star<=(2*i)-1; star++)  printf("*");
        printf("\n");
    }
    for(i=mid-1; i>0; i--){
        for(space=i; space<mid; space++)    printf(" ");
        for(star=1; star<=(2*i)-1; star++)  printf("*");
        printf("\n");
    }
}

void empty_diamond()
{
    int i, input, star, space, mid;
    printf("input");
    scanf("%d", &input);
    mid = (input / 2) + 1;
    for (i = 1; i <= mid; i++) {
        for (space = i; space < mid; space++)
            printf(" ");
        for (star = 1; star <= (2 * i) - 1; star++) {
            if (star == 1 || star == (2 * i) - 1)
                printf("*");
            else
                printf(" ");
        }
        printf("\n");
    }
    for (i = mid - 1; i > 0; i--) {
        for (space = i; space < mid; space++)
            printf(" ");
        for (star = 1; star <= (2 * i) - 1; star++){
            if (star == 1 || star == (2 * i) - 1)
                printf("*");
            else
                printf(" ");
        }
```

```
            printf("\n");
        }
    }

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

ISR中斷解釋

寫出ADC動作原理:透過timer一直去抓取數值...