

C Language

1. point解釋：記錄某個變數、陣列或函式的記憶體位址

基本概念：

1. 指標變數的宣告

```
c  
int *p; // p 是一個指向 int 型態的指標變數
```

p 本身存放的是某個 int 變數的記憶體位址。

2. 指標的賦值

```
c  
int a = 10;  
p = &a; // p 存放 a 的位址
```

&a 代表變數 a 的記憶體位址，這個位址被存入 p。

3. 透過指標存取值

```
c  
printf("%d\n", *p); // 輸出 10
```

*p 代表「存放在 p 指向的記憶體位址中的值」，所以 *p 取得的是 a 的值。

小結：

- 要輸出指標的記憶體位址 → 使用 %p，例如 printf("%p\n", (void *)p);
- 要輸出指標指向的值 → 使用 *p，例如 printf("%d\n", *p);
- 不要用 %d 輸出指標，因為 p 不是 int，會導致未定義行為。

2. example

```
#include <stdio.h>

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

int main(){
    int num = 5;
    func(&num);
    printf("num:%d\n",num);      //ans : 6, ptr++ : num = 5
    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 ++。
```

4. 解釋以下指標意義

```
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
```

- `int *a[10]` 是「10 個指標」，每個指標指向 `int`。
- `int (*a)[10]` 是「1 個指標」，它指向 `int[10]` 陣列。

5. 寫一個function讓變數a跟b能夠交換，不透過暫存變數

```
#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;
}
```

6. 回答printf的答案

```
#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
```

陣列索引運算規則：

c

複製 編輯

`a[i]` 等價於 `*(a + i)`

所以：

c

複製 編輯

`(a + 1)[3] == *(a + 1 + 3) == *(a + 4)`

這表示 `a[4]` 的值，而 `a[4]` 在陣列中是：

c

複製 編輯

```
a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
     0 1 2 3 4 5 6 7 8 9 <-- 索引值
```

`a[4] = 5`，但 `&(a + 1)[3]` 是它的位址。

因此：

c

複製 編輯

```
q = &(a[4]) // q 指向 a[4] 的位址
```

1. `a + 1` 代表 `a[1]` 的位址。
2. `(a + 1)[3]` 其實是 `a[4]`，所以 `&(a + 1)[3]` 取得 `a[4]` 的位址。
3. `q` 指向 `a[4]`，所以 `*q` 的值是 `5`。
4. 輸出 `6. 5`。

7. 回答陣列的答案

```
#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
```

8. 回答以下問題的答案

```
#include <stdio.h>

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int *p = (int*)(&a+1);
    printf("%d\n", *(a+1));
    printf("%d\n", *(p-1));
    return 0;
}
// Ans : 2,5
```

程式碼分析

C

複製 編輯

```
int a[5] = {1, 2, 3, 4, 5};  
int *p = (int*)(&a+1);
```

1. a 與 &a 的區別

- a 代表 陣列的起始位址，即 `&a[0]`，型別是 `int*`。
- &a 代表 整個陣列的位址，型別是 `int (*)[5]`（指向 `int[5]`）。

重點：

- a + 1 代表 a 的位址加上一個 int，即 `&a[1]`。
- &a + 1 代表 &a 加上一個 完整陣列（5 個 int 的大小），即 a 之後的記憶體區塊。

2. &a + 1 計算

C

複製 編輯

```
int *p = (int*)(&a+1);
```

&a 是指向 `int[5]` 的指標，而 &a + 1 代表：

- 跳過 5 個 int 的空間，因此 &a + 1 指向 a 之後的記憶體區塊（a[5] 之後的位址）。

因為 p 是 `(int*)(&a+1)`，所以 p 變成：

C

複製 編輯

```
p = a + 5;
```



因為 `p` 是 `(int*)(&a+1)` · 所以 `p` 變成 :

```
C 複製 編輯  
p = a + 5;
```

即 指向 `a[5]` 的記憶體位址 (但 `a[5]` 本身未定義) 。

3. `*(a+1)` 計算

```
C 複製 編輯  
printf("%d\n", *(a+1));
```

因為 `a` 指向 `a[0]` · 所以 :

```
C 複製 編輯  
*(a+1) == a[1] == 2
```

所以 第一個輸出是 `2` 。

4. `*(p-1)` 計算

```
C 複製 編輯  
printf("%d\n", *(p-1));
```

4. *(p-1) 計算

C

復製 編輯

```
printf("%d\n", *(p-1));
```

我們知道：

C

復製 編輯

```
p = a + 5;
```

所以 p - 1 代表：

C

復製 編輯

```
p - 1 = (a + 5) - 1 = a + 4
```

而 a[4] 是 5，所以：

C

復製 編輯

```
*(p-1) == a[4] == 5
```

所以 第二個輸出是 5。

9. 請問輸出為何

```
#include <stdio.h>

int main() {
    char i[ ] = "Hello";
    char a = 1;
    char *p = &a;
    int n = 10;
    printf("%d %d %d\n", sizeof(i), sizeof(p), sizeof(n));
    return 0;
}
// 6,8,4
```

10. 請問輸出為何

```
#include <stdio.h>

int main(){
    int arr[] = {10,20,30,40,50};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf("%d", (ptr2-ptr1));
    printf("%d", (char*)ptr2 - (char*)ptr1);
}
// 5,20(5*4)
```

運算解析

1. `ptr2 - ptr1`

指標相減 `(ptr2 - ptr1)` 表示 兩個指標相差幾個 `int` 大小的元素。

- `ptr1 = arr` (指向 `arr[0]`) 。
- `ptr2 = arr + 5` (指向 `arr[5]` · 這裡 `arr[5]` 在陣列範圍外 · 但 C 允許這樣的指標運算) 。
- `ptr2 - ptr1` 計算 :

$$(arr + 5) - (arr) = 5$$

輸出 : 5

2. `(char*)ptr2 - (char*)ptr1`

這裡 `(char*)ptr2` 和 `(char*)ptr1` 先被轉型為 `char*` · 之後再相減。

- 在 C 中 · `char*` 指標是以 byte 為單位 運算的。
- 假設 `int` 的大小是 4 bytes (這在大多數系統上成立) 。
- `ptr2 = arr + 5` · 也就是 `ptr2` 指向 `arr[5]` 的記憶體位址 :

```
ini 复制 编辑

ptr1 = arr    (指向 arr[0])
ptr2 = arr+5  (指向 arr[5] · 其實是 arr[0] + 5 * sizeof(int))
```

- 轉換成 `char*` 後 · 計算記憶體位址差距 :

$$(char*)ptr2 - (char*)ptr1 = (5 \times sizeof(int))$$

假設 `sizeof(int) = 4` · 則結果為 $5 \times 4 = 20$



11. 請問輸出為何

```
#include <stdio.h>

int main(){
    int arr1[] = {10, 20};
    int arr2[] = {10, 20};
    int arr3[] = {10, 20};
    int *p = arr1;
    int *q = arr2;
    int *r = arr3;
    ++*p;
    *q++;
    *++r;
    printf("%d %d %d\n", arr1[0], arr1[1], *p);
    printf("%d %d %d\n", arr2[0], arr2[1], *q);
    printf("%d %d %d\n", arr3[0], arr3[1], *r);
}
// 11 20 11
// 10 20 20
// 10 20 20
```

關鍵指標運算

1. `++*p;`

C

複製 編輯

```
++*p;
```

- `p = arr1;` (指向 `arr1[0]`) 。
- `*p` 代表 `arr1[0]` , 所以 `++*p` 會讓 `arr1[0]` 加 1 。
- `arr1[0] = 10 + 1 = 11` 。

2. `*q++;`

C

複製 編輯

```
*q++;
```

- `q = arr2;` (指向 `arr2[0]`) 。
- `*q++` 的執行順序：
 1. 先取 `*q` 的值 (即 `arr2[0] = 10`) 。
 2. 再讓 `q` 自增 (指向 `arr2[1]`) 。
- 但 `*q` 的值沒有改變 , 所以 `arr2[0]` 仍然是 `10` , `q` 變成指向 `arr2[1]` 。

3. `*++r;`

C

複製 編輯

```
*++r;
```

- `r = arr3;` (指向 `arr3[0]`) 。
- `*++r` 的執行順序：
 1. `++r` 先讓 `r` 指向下一個元素 (即 `arr3[1]`) 。
 2. `*r` 只是取值 · 但不修改 `arr3[1]` 的值 。
- 所以 `arr3` 完全沒有改變 · `r` 現在指向 `arr3[1]` 。

最終變數狀態

變數	變化前	變化後	指標指向
<code>arr1[0]</code>	10	11	<code>p → arr1[0]</code>
<code>arr1[1]</code>	20	20	
<code>arr2[0]</code>	10	10	
<code>arr2[1]</code>	20	20	<code>q → arr2[1]</code>
<code>arr3[0]</code>	10	10	
<code>arr3[1]</code>	20	20	<code>r → arr3[1]</code>

輸出結果

C

複製 編輯

```
printf("%d %d %d\n", arr1[0], arr1[1], *p);
```

- arr1[0] = 11
- arr1[1] = 20
- *p = arr1[0] = 11

複製 編輯

```
11 20 11
```

C

複製 編輯

```
printf("%d %d %d\n", arr2[0], arr2[1], *q);
```

- arr2[0] = 10
- arr2[1] = 20
- *q = arr2[1] = 20 (因為 q 被移動到 arr2[1])

複製 編輯

```
10 20 20
```

```
C
```

復製 編輯

```
printf("%d %d %d\n", arr3[0], arr3[1], *r);
```

- `arr3[0] = 10`
- `arr3[1] = 20`
- `*r = arr3[1] = 20` (因為 `r` 被移動到 `arr3[1]`)

```
10 20 20
```

12. 請問輸出為何

```
#include <stdio.h>

void f1(int *p, int *q){
    p = q;
    *p = 2;
}

void f2(int *p, int *q){
    *p = *q;
    *p = 2;
}

int main(){
    int i=0, j=1;
    f1(&i, &j);
    printf("%d %d\n", i, j);

    i=0, j=1;
    f2(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}
// 0 2
// 2 1
```

f1 的情況：為什麼 `*p = 2` 讓 `j = 2`

```
c  
void f1(int *p, int *q){  
    p = q; // p 指向 q 的地址  
    *p = 2; // p 指向的位置的值設為 2  
}
```

1. 初始狀態：

- `p = &i` (指向 `i` 的地址) · `q = &j` (指向 `j` 的地址)。
- 此時 `i = 0` · `j = 1`。

2. `p = q`：

- 這一步把 `q` 的值 (`j` 的地址) 賦給 `p`。
- 現在 `p` 不再指向 `i`，而是指向 `j`。
- 所以，`p` 和 `q` 都指向 `j` 的地址。

3. `*p = 2`：

- `*p` 表示「`p` 指向的記憶體位置的值」。
- 因為 `p` 現在指向 `j`，所以 `*p = 2` 就是在修改 `j` 的值。
- 結果：`j` 被設為 2，`i` 沒被碰過，還是 0。

4. 最終結果：

- `i = 0` · `j = 2`。
- 輸出：`0 2`。

為什麼 `j = 2`？

因為在 `f1` 中，`p = q` 讓 `p` 指向了 `j`，所以後面的 `*p = 2` 修改的是 `j` 的值。

f2 的情況：為什麼 `*p = 2` 沒讓 `j = 2`

```
c
void f2(int *p, int *q){
    *p = *q; // 把 q 指向的值賦給 p 指向的位置
    *p = 2;   // p 指向的位置設為 2
}
```

1. 初始狀態：

- `p = &i` (指向 `i` 的地址) · `q = &j` (指向 `j` 的地址)。
- 此時 `i = 0` · `j = 1`。

2. `*p = *q` :

- `*q` 是 `q` 指向的值 · 也就是 `j` 的值 (1)。
- `*p` 是 `p` 指向的位置 · 也就是 `i`。
- 所以 `*p = *q` 把 `j` 的值 (1) 複製到 `i` · 現在 `i = 1`。
- `p` 和 `q` 的指向沒變 · `p` 還是指向 `i` · `q` 還是指向 `j`。

3. `*p = 2` :

- `*p` 表示「`p` 指向的記憶體位置的值」。
- `p` 仍然指向 `i` (因為 `f2` 沒改變 `p` 的指向)。
- 所以 `*p = 2` 把 `i` 的值從 1 改成 2。
- `j` 完全沒被碰過 · 還是 1。

4. 最終結果：

- `i = 2` · `j = 1`。
- 輸出：`2 1`。

為什麼 `j = 1`？

因為在 `f2` 中 · `p` 一直指向 `i` · `*p = 2` 修改的是 `i` 的值 · 而 `j` 沒有被任何操作影響 · 所以保持初始值 1。

13. 請問輸出為何

```
#include <stdio.h>

int main(){
    int ref[]={8,4,0,2};
    int *ptr;
    int index;
    for(index=0, ptr=ref; index<2; index++,ptr++)
        printf("%d %d\n", ref[index], *ptr);
        (*ptr++);
    printf("%d %d\n", ref[index], *ptr);
}

/*
* 8 8
* 4 4
* 0 2
*/
```

14. 請問輸出為何

```
#include <stdio.h>

int main() {
    char *str[ ][2] =
    { "professor", "Justin" ,
      "teacher", "Momor" ,
      "student", "Caterpillar"};  
  

    char *str2[ ][3] =
    { "professor", "Justin" ,
      "teacher", "Momor" ,
      "student", "Caterpillar"};  
  

    char str3[3][10] = {"professor", "Justin", "etc"};
    printf("%s\n",str[1][1]);
    printf("%s\n",str2[1][1]);
    printf("%c\n",str3[1][1]);
}  

// Momor
// student
// u
```

1. `str` 的定義和行為

```
c

char *str[][2] = {
    "professor", "Justin",
    "teacher", "Momor",
    "student", "Caterpillar"
};
```

- `char *str[][2]`：
 - 這是一個二維陣列，裡面的元素是指標 (`char *`)，指向字串的開頭。
 - 第一維大小未指定，編譯器根據初始化的內容推斷為 3 (3 組資料)。
 - 第二維大小是 2，每組有 2 個字串指標。
 - 每個字串 (例如 `"professor"`) 在記憶體中以字元陣列形式儲存，並以空字符 `\0` 結尾。
- 初始化：
 - `str[0][0] = "professor", str[0][1] = "Justin"`
 - `str[1][0] = "teacher", str[1][1] = "Momor"`
 - `str[2][0] = "student", str[2][1] = "Caterpillar"`
- `printf("%s\n", str[1][1])`：
 - `str[1][1]` 是 `"Momor"`，這是一個字串指標 (`char *`)。
 - `%s` 格式化符號會從該指標開始，逐一打印字元，直到遇到 `\0`。
 - 輸出：`"Momor"`

2. `str2` 的分析 (這裡我之前弄錯了)

```
c  
char *str2[][3] = {  
    "professor", "Justin",  
    "teacher", "Momor",  
    "student", "Caterpillar"  
};
```

- 類型：`char *str2[][3]` 是一個二維陣列，元素是指標 (`char *`)。
- 結構：
 - 第一維大小由編譯器推斷，但第二維明確指定為 3。
 - 我之前的錯誤在於假設它是「每行 3 個元素」，但初始化方式表明這可能是編譯器處理上的特殊情況。
- 初始化行為：
 - C 語言中，二維陣列的初始化是按順序填入的，並且會根據第二維的大小重新組織資料。
 - 這裡提供了 6 個字串 (`"professor", "Justin", "teacher", "Momor", "student", "Caterpillar"`)，而第二維是 3。
 - 編譯器會將這 6 個字串放入一個連續的陣列，並按每行 3 個的方式分配：
 - `str2[0][0] = "professor", str2[0][1] = "Justin", str2[0][2] = "teacher"`
 - `str2[1][0] = "Momor", str2[1][1] = "student", str2[1][2] = "Caterpillar"`
 - 第一維大小被推斷為 2 (因為 6 個元素 $\div 3 = 2$ 行)。
- `printf("%s\n", str2[1][1])`：
 - `str2[1][1]` 是 `"student"`，一個字串指標。
 - `%s` 打印整個字串。
 - 輸出：`student`

3. `str3` 的分析

```
c

char str3[3][10] = {"professor", "Justin", "etc"};
```

- 類型：`char str3[3][10]` 是一個二維字元陣列，每行最多存 10 個字元（包括 `\0`）。
- 初始化：
 - `str3[0] = "professor"` (9字元 + `\0`)
 - `str3[1] = "Justin"` (6字元 + `\0`，剩餘填 `\0`)
 - `str3[2] = "etc"` (3字元 + `\0`，剩餘填 `\0`)
- `printf("%c\n", str3[1][1])`：
 - `str3[1]` 是 `"Justin"`。
 - `str3[1][1]` 是第 2 個字元（索引從 0 開始），即 `'u'`。
 - `%c` 打印單個字元。
 - 輸出：`u`

15. 請問輸出為何？

```
#include <stdio.h>

int main() {
    int cnt = 10;
    const char *pc = "welcome";
    while(*pc++)
        cnt++;
    printf("cnt:%d\n", cnt);
}
```

// 17

16. 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。

17. call by value, call by reference, call by address解釋?

call by value : 當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)

```
#include <stdio>

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

```
#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，從而讓函數內的變數

18. local variable：僅活在該函式內，存放位置在 stack 或 heap 記憶體中

這範例太爛

```
#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;
}
```

19. static variable：？



static 的主要用途

1. 在函式內的 static 變數：保留值不變
2. 在檔案範圍的 static 變數：只在該檔案內可見
3. 在函式前的 static：限制函式的可見範圍

◆ 1. 在函式內的 `static` 變數

● 特性

- 靜態區域存儲：變數在程式開始時分配記憶體，直到程式結束才釋放（生命周期 = 程式整體）。
- 初始值只設定一次，即使函式多次呼叫，變數的值仍然保留。

► 範例

```
C 複製 編輯

#include <stdio.h>

void counter() {
    static int count = 0; // 只初始化一次，後續函式呼叫會保留 `count` 的值
    count++;
    printf("Count: %d\n", count);
}

int main() {
    counter(); // Count: 1
    counter(); // Count: 2
    counter(); // Count: 3
    return 0;
}
```

❖ 重點

- `static int count = 0;` 這行程式碼只執行一次（在 `counter()` 第一次呼叫時）。
- 之後，每次呼叫 `counter()`，變數 `count` 不會被重新初始化，而是保留前一次的值。

◆ 2. 在檔案範圍 (File Scope) 宣告 static 變數

💡 特性

- `static` 變數如果宣告在 函式外部，它的作用範圍 (scope) 只限於該檔案。
- 這對於 避免變數名稱衝突 很有幫助。

📝 範例

file1.c

```
C 複製 編輯

#include <stdio.h>

static int value = 10; // 這個變數只在 file1.c 可見

void printValue() {
    printf("Value: %d\n", value);
}
```

file2.c

```
C 複製 編輯

#include <stdio.h>

// extern int value; // 這裡不能存取 `value`，因為它在 file1.c 內是 `static`

int main() {
    // printf("%d\n", value); // 錯誤：value 不能在這裡使用
    return 0;
}
```

📌 重點

- `static int value = 10;` 限制 `value` 只能在 `file1.c` 內使用，`file2.c` 無法存取它。
- 這樣可以避免不同檔案間的全域變數名稱衝突。

◆ 3. 在函式前使用 `static` (靜態函式)

💡 特性

- `static` 限制函式的作用範圍只在該檔案內，其他檔案無法存取。
- 避免命名衝突，特別是在多人開發的專案中很有用。

📝 範例

file1.c

C 複製 編輯

```
#include <stdio.h>

static void myFunction() { // 這個函式只能在 file1.c 內使用
    printf("Hello from static function!\n");
}

int main() {
    myFunction(); // 可以呼叫
    return 0;
}
```

file2.c

C 複製 編輯

```
#include <stdio.h>

// extern void myFunction(); // 錯誤：myFunction 不能在這裡使用，因為它在 file1.c 是 `static` 

int main() {
    // myFunction(); // 錯誤：無法存取
    return 0;
}
```

📌 重點

- `static void myFunction();` 限制 `myFunction` 只能在 `file1.c` 內部使用，`file2.c` 無法存取它。

20. global variable : 所有區域皆可使用此變數

```
#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;
}
```

21. **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** 單明該變數；如果變數只在當前文件中使用，可以不使用 **extern** 關鍵字單明

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

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

extern int a;

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

22. 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

1. 定義與作用

`const` 用於聲明一個變量為常量，表示其值在初始化後不可被修改。

2. 主要特點

不可修改：用 `const` 單詞的變量在初始化後不能被重新賦值。例如：

```
const int MAX_USERS = 100;  
MAX_USERS = 200; // 錯誤！不能重新賦值
```

必須初始化：在 C 語言中，`const` 變量必須在聲明時初始化，否則編譯器會報錯。例如：

```
const int MY_VAR; // 錯誤！const 變量必須初始化  
const int MY_VAR = 10; // 正確
```

23. 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volatile

24. inline 寫法

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

25. 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)
```

26. inline / Macro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



27. #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))

// Example :
#include <stdio.h>

#define MUX(a,b) a*b
int main() {
    printf("%d\n", MUX(10+5,10-5));
    return 0;
}
// 55
```

28. 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
```

29. #define swap

```
#include <stdio.h>

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

int main(){
    int a = 5;
    int b = 10;
    printf("before a=%d,b=%d\n",a,b);
    swap(a,b); // = {int temp = a; a = b; b = temp;}
    printf("after a=%d,b=%d\n",a,b);

    return 0;
}
```

30. 解釋 Interrupt 的處理流程

Interrupt (中斷) 的意義

它的核心意義是打斷當前正在進行的任務，去處理另一個更緊急或需要即時響應的事件。

1. 觸發中斷(Interrupt)
2. 儲存目前 CPU 的執行狀態
3. CPU 查詢中斷向量表，並跳轉到對應之 ISR
4. 執行 ISR 裡的內容
5. ISR 執行完成後繼續執行被中斷打斷的原始程式

*什麼是中斷向量表？

中斷向量表是一個儲存在記憶體中的表格，表格的每一項（稱為「向量」）對應一個特定中斷或異常類型，並記錄了該中斷對應的處理程序（ISR）的記憶體地址。當 CPU 接收到中斷時，它會參考這個表格，找到並跳轉到正確的處理程式。

*ISR 是 Interrupt Service Routine 的縮寫，中文通常翻譯為中斷服務常式或中斷處理程式。它是計算機系統中專門用來處理中斷 (Interrupt) 的一段程式碼。當某個中斷事件發生時，CPU 會暫停當前任務，跳轉到對應的ISR去執行，完成特定工作後再返回原本的程式流程。

31. Interrupt 有哪些

1. 外部中斷 (External Interrupt)：這種中斷是由於外部事件或設備引發的。MCU可以設定外部中斷接腳，當接腳狀
2. 計時器中斷 (Timer Interrupt)：MCU通常具有內部的計時器/計數器模組，可以用於定時和計數操作。這種中斷是
3. 串列通訊中斷 (Serial Communication Interrupt)：MCU通常支援不同的串列通訊協議，如UART (Universal A
4. ADC中斷 (Analog-to-Digital Converter Interrupt)：MCU通常具有ADC模組，用於將類比信號轉換為數位值。
5. 內部中斷 (Internal Interrupt)：這種中斷是由MCU內部事件觸發的。例如，存儲器錯誤、數學錯誤或其他不正常

32. different between interrupt and polling

interrupt :

- 提高效率：不必讓CPU一直輪詢（polling）檢查是否有事件發生，減少資源的浪費。
- 即時性：保證關鍵事件（例如硬體故障或用戶輸入）能被快速處理。

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

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

34. 不同類型的Byte大小

以下都為16位元

Type	64-bit	32-bit
char	1	1
short	2	2
int	4	4
long	8	4

Type	64-bit	32-bit
long long	8	8
double	8	8
long double	16	12
point	8	4
size_t	8	4

35. struct佔幾個byte

```
#include <stdio.h>

typedef struct MyStruct1
{
    char a[2];
    int b;
    double c;
    int *Pint;
    char d;
    char* Pchar;
} MyStruct1;
```

```
typedef struct MyStruct2
{
    char a;
    char b;
    struct str2
    {
        int d;
    } c;
}MyStruct2;
```

```
typedef struct MyStruct3
{
    char a;
    int b;
    char c;
    char* d;
    double* e;
    struct str3
    {
        int f;
        char g;
        struct str33
        {
            char* p;
        }n;
    } m;
}MyStruct3;
```

```

typedef struct MyStruct4
{
    char a;
    char b;
    struct str4
    {
        char c;
        char d;
    };
}MyStruct4;

int main() {
    printf("1. %zu\n", sizeof(MyStruct1));
    printf("2. %zu\n", sizeof(MyStruct2));
    printf("3. %zu\n", sizeof(MyStruct3));
    printf("4. %zu\n", sizeof(MyStruct4));
    return 0;
}
//ans = 4(2+2對齊)+4+8+8+8(1+7對齊)+8 = 40Byte
//ans = 4(1+1+2)+4 = 8byte
//ans = 4(1+3)+4+8(1+7後面接8所以要+7才能是8的倍數)+8+8+4+4(1+3因為+3後就是40了為8的倍數)+8 = 48byte
//ans = 1+1 = 2byte *並沒有申明這個結構體的變數所以str2不用計算

```

36. 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;
}
```

37. 請問輸出為何

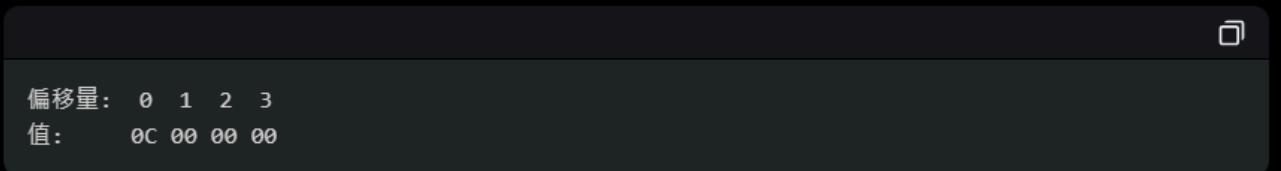
```
#include <stdio.h>

union AA{
    char a[2];
    int s;
};

int main()
{
    union AA aa = {0};
    aa.a[0] = 12;
    aa.a[1] = 1;
    printf("%x\n", aa.s);
    printf("%zu\n", sizeof(aa));
    return 0;
}
// 10C = 0x01 0C
// 4
```

2. 賦值

- `aa.a[0] = 12;`
 - 將偏移量0的字節設為12 (16進位為 `0x0C`)。
 - 內存變為：



- `aa.a[1] = 1;`
 - 將偏移量1的字節設為1 (16進位為 `0x01`)。
 - 內存變為：



3. 輸出 aa.s

- `printf("%x\n", aa.s);`
 - 因為 `union` 的成員共享內存，`aa.s` 是對這4字節的整數解釋。
 - 但這涉及到端序（endianness）：
 - 小端序（Little Endian）：低位字節放在低地址（大多數現代系統，如x86）。
 - 內存：`0C 01 00 00` 表示 `int s = 0x0000010C`。
 - 輸出：`10c`（16進位）。
 - 大端序（Big Endian）：高位字節放在低地址（某些嵌入式系統）。
 - 內存：`0C 01 00 00` 表示 `int s = 0xC010000`。
 - 輸出：`c010000`。
 - 假設是小端序（更常見），`aa.s` 的值是 `0x0000010C`，輸出為 `10c`。

為什麼不是 $2 + 4$ ？

- 如果是 `struct`，成員會依次排列：

```
c

struct AA {
    char a[2]; // 2字節
    int s;      // 4字節
};
```

- `struct AA` 的布局：
 - `a[2]`：偏移量 0-1（2字節）。
 - `int s`：需要 4 字節對齊，可能在偏移量 2-5（總共 6字節），或者填充到 8字節（取決於對齊規則）。
- 總大小是成員大小的總和（加上可能的填充），可能是 6 或 8字節。
- 但在 `union` 中，不是相加，而是取最大值：
 - `char a[2]` 需要 2字節。
 - `int s` 需要 4字節。
 - `union AA` 必須能容納 `int s`，所以大小是 4字節，而不是 $2 + 4$ 。

38. 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
```

39. 二維陣列大小

```
#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 120 (5 * 3 * 8(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;
}
```

40. 各sizeof大小

```
#include <stdio.h>

int main() {
    char *s = "hello";
    char s1[]={'h','e','l','l','o'};
    int s2[]={ 'h','e','l','l','o'};

    printf("%d\n", sizeof(s)); //ponit 32位元為4byte / 64位元為8byte
    printf("%d\n", sizeof(s1)); //1*5 = 5
    printf("%d\n", sizeof(s2)); //4*5 = 20
    return 0;
}
```

41. setting a bit

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

42. clearing a bit

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

43. fliping a bit

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

44. checking a bit

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

45. 回答ans的數值

```
#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  
}
```

46. 回答ans的數值2

```
#include <stdio.h>  
  
int main() {  
    printf("ans=%d\n",fun(456)+fun(123)+fun(789));  
}  
  
int fun(int x){  
    int count = 0 ;  
    while(x){  
        count++ ;  
        x = x & (x-1) ;  
    }  
    return count ;  
}
```

47. 0x12345678 轉換為 0x87654321

```
#include <stdio.h>

unsigned int func(int x) {
    unsigned int n[8],sum = 0;
    n[0] = ((x & 0x0000000F) <<28);
    n[1] = ((x & 0x000000F0) <<20);
    n[2] = ((x & 0x00000F00) <<12);
    n[3] = ((x & 0x0000F000) <<4);
    n[4] = ((x & 0x000F0000) >>4);
    n[5] = ((x & 0x00F00000) >>12);
    n[6] = ((x & 0xF0000000) >>20);
    n[7] = ((x & 0xF0000000) >>28);
    for(int i=0; i<8; i++){
        sum += n[i];
    }
    return sum;
}

int main() {
    unsigned int data = 0x12345678;
    printf("轉換前: 0x%x\n", data);

    unsigned int swapped = func(data);
    printf("轉換後: 0x%x\n", swapped);

    return 0;
}
```

48. 紿一個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 = 0xFFFF;
    printf("ans:%d\n",function(num));
    return 0;
}
```

49. bubbleSort版本1

```
#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;
}
```

50. bubbleSort版本2

```
#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;
}
```

51. 設定一個絕對位址為0x67a9的整數型變數的值為0xaa55

```
#include <stdio.h>

int main(){
    int *ptr = (int *)0x67a9;
    *ptr = 0xaa55;
}
```

52. N是否為判斷2的次方

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

53. 連續呼叫 func 10 次，印出的值為何？

```
#include <stdio.h>  
  
void func(void)  
{  
    static int i = 0 ;  
    i++ ;  
    printf("%d" , i ) ;  
}  
  
int main(void)  
{  
    for(int i=0; i<10; i++)  
        func();  
}  
//若沒static，i會一直歸零  
// 12345678910
```

54. i++ & ++i

```
#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
```

- `i = 5;`: 將變數 `i` 初始化為 5。
- `j = i++;`：
 - 這裡使用了後置遞增 (`i++`)，意思是「先使用 `i` 的值，再將 `i` 增加 1」。
 - 執行時，`j` 會被賦予 `i` 的當前值 (5)，然後 `i` 會自增為 6。
- `printf("i1 = %d\n", i);`：輸出 `i` 的值，此時 `i = 6`。
- `printf("j1 = %d\n", j);`：輸出 `j` 的值，此時 `j = 5`。

- `i = 5;`: 將 `i` 重新設為 5。
- `j = ++i;`:
 - 這裡使用了前置遞增 (`++i`)，意思是「先將 `i` 增加 1，再使用新的值」。
 - 執行時，`i` 先自增為 6，然後 `j` 被賦予 `i` 的新值 (6)。
- `printf("i2 = %d\n", i);`: 輸出 `i` 的值，此時 `i = 6`。
- `printf("j2 = %d\n", j);`: 輸出 `j` 的值，此時 `j = 6`。

55. 寫個function判斷基數偶數

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

56. 寫個function計算有幾個位元是 1

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

57. 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。

58. The faster way to an integer multiply by 7(乘以7)

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

59. declaration (宣告) 和 definition (定義) 的差異

3. 宣告與定義的主要區別

特性	宣告 (Declaration)	定義 (Definition)
目的	告訴編譯器名稱和類型	分配記憶體或提供具體實現
記憶體分配	不分配記憶體 (變數) 或無實現 (函數)	分配記憶體 (變數) 或提供實現 (函數)
次數限制	可以多次宣告同一個名稱	同一個名稱只能定義一次 (同一作用域)
是否包含定義	宣告不一定是定義	定義同時也是一種宣告
關鍵字範例	extern, 函數原型, 前向宣告	變數初始化, 函數體, 結構體內容

範例 1：變數的宣告與定義

```
c

// file1.c
extern int x;           // 壓告 x，但不分配記憶體
void print_x(void) {
    printf("%d\n", x);
}

// file2.c
int x = 100;            // 定義 x，分配記憶體並初始化
```

60. Big / Little-Endian：他們是CPU中兩種不同位元組排序

差異說明：

Big-Endian (大端序) :

定義：最高有效位元組 (Most Significant Byte, MSB) 儲存在最低的記憶體位址。

範例：對於一個32位整數 `0x12345678` :

記憶體儲存順序 (從低位址到高位址) : `12 34 56 78`

直觀理解：就像我們寫數字的順序，從高位到低位。

使用平台：常見於網路協議 (如TCP/IP) 、某些RISC架構 (如PowerPC、SPARC) 。

Little-Endian (小端序) :

定義：最低有效位元組 (Least Significant Byte, LSB) 儲存在最低的記憶體位址。

範例：對於同樣的32位整數 `0x12345678` :

記憶體儲存順序 (從低位址到高位址) : `78 56 34 12`

直觀理解：低位字節優先，與Big-Endian相反。

使用平台：常見於x86、x86-64架構 (如大多數PC) 。

61. 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
#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",fibonacci(n));
    return 0;
}
```

//遞迴解法

```
int function(int n) {  
    if (n <= 1)  
        return n;  
  
    return function(n - 1) + function(n - 2);  
}  
  
int main() {  
    int n;  
    printf("input:");  
    scanf("%d", &n);  
    printf("%d\n", function(n));  
    return 0;  
}
```

62. binary search

```
#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;
}
```

63. 求一個數的最高位1在第幾位

```
#include <stdio.h>

int function(int num){
    if (num == 0) {
        return -1; // 特殊情況：num = 0，沒有 1
    }

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

int main() {
    int num = 16;
    printf("ans:%d\n",function(num));
    return 0;
}
```

64. 最大公因數 遞迴寫法

```
#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(25,10));

    return 0;
}
```

65. 請問以下MIN()的結果為何？

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

66. #error

在C語言中，#error是一個預處理器指令，用於在編譯時生成錯誤訊息。當編譯器遇到#error指令時，它會立即停止編譯並顯示錯誤訊息。

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

67. 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
```

73. if(b() && a()) 這樣的寫法會有啥問題？

&& 運算子具有短路求值的特性：如果 b() 返回 false，則 a() 不會被執行，因為整個表達式已經確定為 false。

74. 判斷閏年(能被4整除但不能被100整除,或是能被400整除)

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

int main()
{
    int year;
    printf("請輸入西元年 : ");
    scanf("%d",&year);
    if((year%4) == 0 && (year%100) != 0 || (year%400) == 0)
    {
        printf("%d是閏年\n",year);
    }
    else
    {
        printf("%d是平年\n",year);
    }
    return 0;
}
```

75. 輸入一unsigned int n，當輸入0則輸出0，輸入1-32為輸出32，33-64輸出64，65-96輸出96 (32進位)

```
#include <stdio.h>

int mul_32(unsigned int x){
    x = x + 31;
    x = x & ~31;
    return x;
}

int main()
{
    printf("%d\n",mul_32(5));
    return 0;
}
```

76. 寫出質數function

一般解法

```
#include <stdio.h>

int func(int num){
    if(num <= 1){
        return 0;
    }
    for(int i=2; i<num; i++){
        if(num%i == 0){
            return 0;
        }
    }
    return 1;
}

int main()
{
    int input;
    printf("input:");
    scanf("%d",&input);

    if(func(input)){
        printf("yes\n");
    }
    else{
        printf("no\n");
    }
    return 0;
}
```

68. 有九顆看起來一模一樣的球 但是有一顆不一樣重 也不知道它是比較輕還比較重 用一個天秤最少要量幾次可以"確保"找出這顆球？

Ans : 3次

將9顆球分成A,B,C共3堆，每堆3顆球

A.ooo B.ooo C.ooo

[第1次]

拿A,B堆共6顆球分別放在天秤兩端

(i) 假設不一樣重，就能知道想找的球在A,B其中之一

(ii) 假設一樣重，就知道不一樣重的球是剩下沒秤的3顆其中之一

[第2次]

換掉輕的3顆(假設B較A輕)，拿另外C堆的3顆上來量

假設天秤平衡了，就能知道不一樣重的球較輕

是剛剛被換掉的B堆3顆其中之一

[第3次]

拿B堆3顆的其中2顆來量

如果平衡了，那麼要找的就是剩下的第3顆較輕的球

69. 從1數到100 喊到100的人獲勝 每一次最少喊一個數 最多喊七個數 先攻的人喊到幾時便保證必勝？

Ans :

如果自己要喊100，對方必須只能喊99~93，

自己要喊 92，對方必須只能喊91~85，

.....

自己要喊 12，對方必須只能喊11~5，

自己要喊 4

因此只要先喊到4，則先喊的人必獲勝，往後原則就是喊 $100 - 8n$

70. 有個商品賣30元 成本25元 客人用100元紙鈔跟商人買了商品 商人沒錢找所以拿了這張紙鈔去跟隔壁攤販換零錢

**找給客人後來隔壁攤販跑來說那是假鈔 所以商人又賠了
100元給隔壁攤販 試問商人總共虧多少錢？**

Ans :

$$-100 + 5 = -95$$

71. 用若干個砝碼組合出1~100公克，請問砝碼最少數量為幾個？

Ans :

7個 (1, 2, 4, 8, 16, 32, 64)

72. 有1支手電筒和5個人，這5個人要過橋，過橋單趟每個人分別需要花費1、3、5、11、13分鐘。橋一次最多只能有兩個人在上面，而且每次過橋都一定要拿著手電筒過去，請問最少花費幾分鐘所有人可以過完橋？

1、3 過去，1回來 ($3 + 1 = 4$ min.)

11、13過去，3回來 ($13 + 3 = 16$ min.)

1、3過去 or 1、5過去，1回來 ($3 + 1 = 4$ min. or $5 + 1 = 6$ min.)

1、5過去 or 1、3過去 (5 min. or 3 min.)

總共花費29分鐘

I2C Protocol

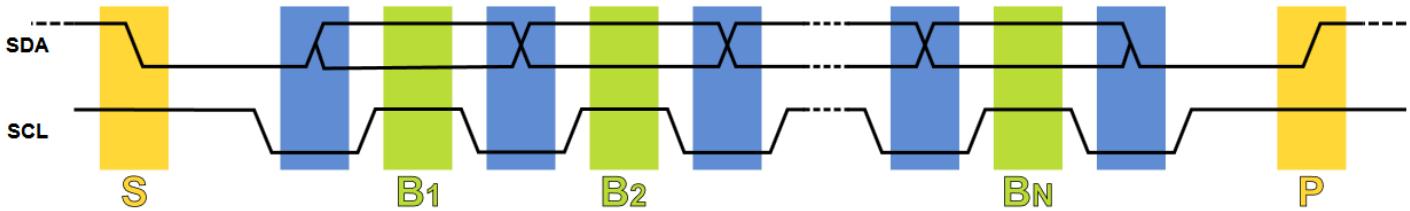
介紹

1. SDA : Serial Data Line, holds Data or address signal

2. SCL : Serial Clock Line, holds Clock signal

時序

1. I²C bus 上無任何活動時，SCL 和 SDA 都維持在 high
2. SCL 為 low 時，SDA 的狀態"可改變"(藍色)
3. SCL 為 high 時，SDA 上的資料"有效"(綠色)
4. SCL 為 high 時，若 SDA 變動則兩種特殊狀況：
 - SDA H->L : START
 - SDA L->H : STOP



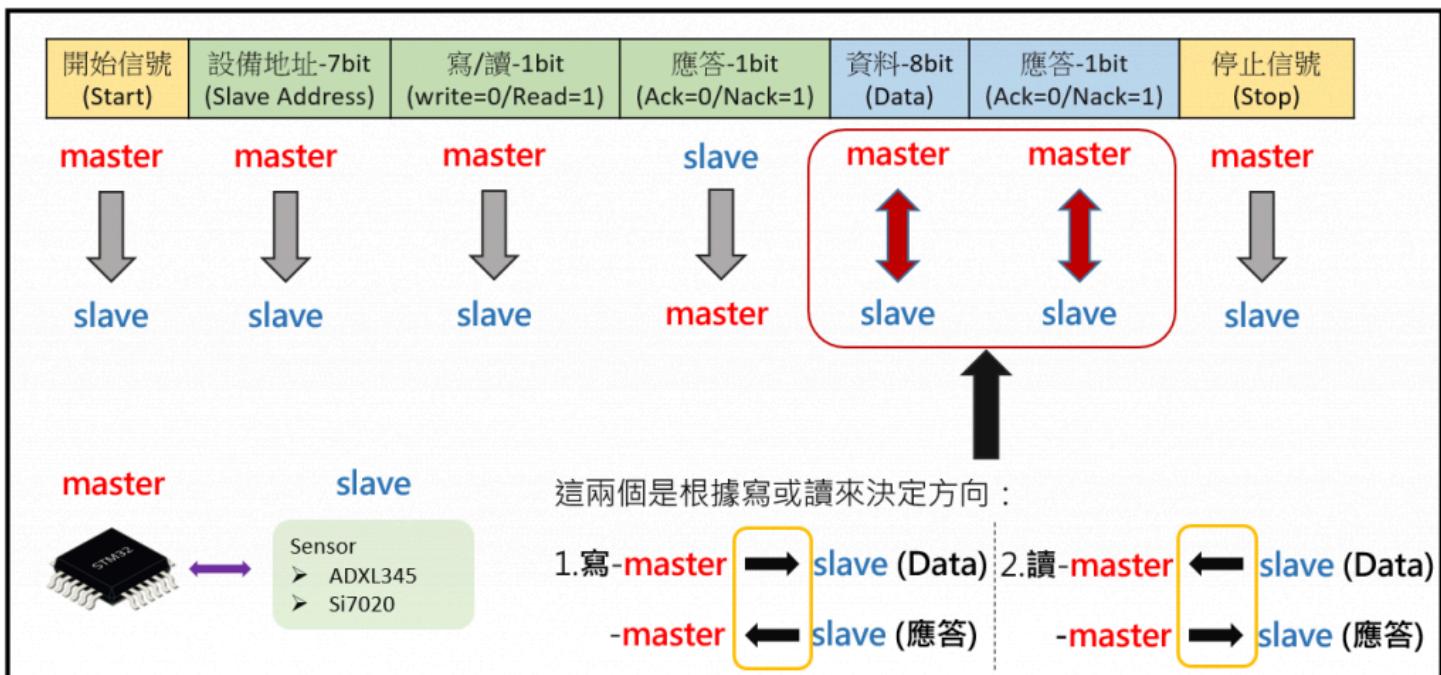
ack

1. 每一個 Byte 的資料傳輸結束後，會跟著一個 ack bit。這個 ack bit 固定由接收方產生，有以下兩種：
 - 當 master 是傳送方、slave 是接收方，ack 由 slave 回應
 - 當 master 是接收方、slave 是傳送方，ack 由 master 回應

2. ack 定義：

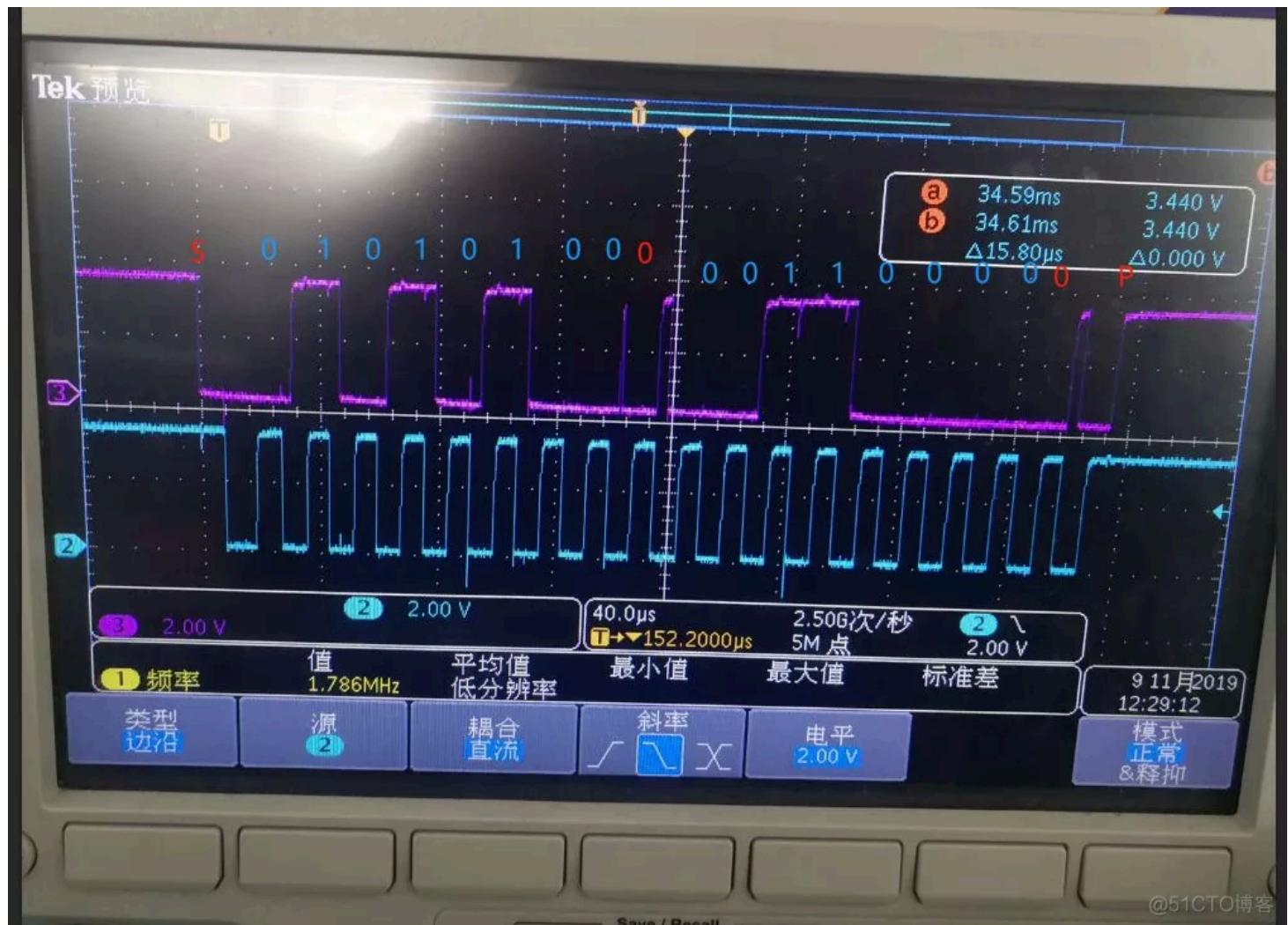
0(ack) - OK

1(nack) - FAIL



I2C寫入範例

1. 紫色(SDA)、藍色(SCL)
2. 當 SDA 由 H->L 且 SCL 為 H，開始資料傳輸
3. Master 寫入設備地址(0b0101010)後再寫入write byte(0)
4. Slave 回應 0(ack)
5. Master 寫入資料(0b00110000)
6. Slave 回應 0(ack)，寫入完成
7. SDA 由 L->H 且 SCL 為 H，結束資料傳輸



@51CTO博客

I2C讀取範例

1. 紫色(SDA)、藍色(SCL)
2. 當 SDA 由 H->L 且 SCL 為 H，開始資料傳輸
3. Master 寫入設備地址(0b0101010)後再寫入read byte(1)
4. Slave 回應 0(ack)
5. Slave 丟出資料(0b00110000)
6. Master 回應 1(nack)，停止讀取data
7. SDA 由 L->H 且 SCL 為 H，結束資料傳輸



SPI Protocol

介紹

SPI 的通訊協定有兩個重要的參數 CPOL/CPHA，說明如下。

1、CPOL (Clock Polarity)

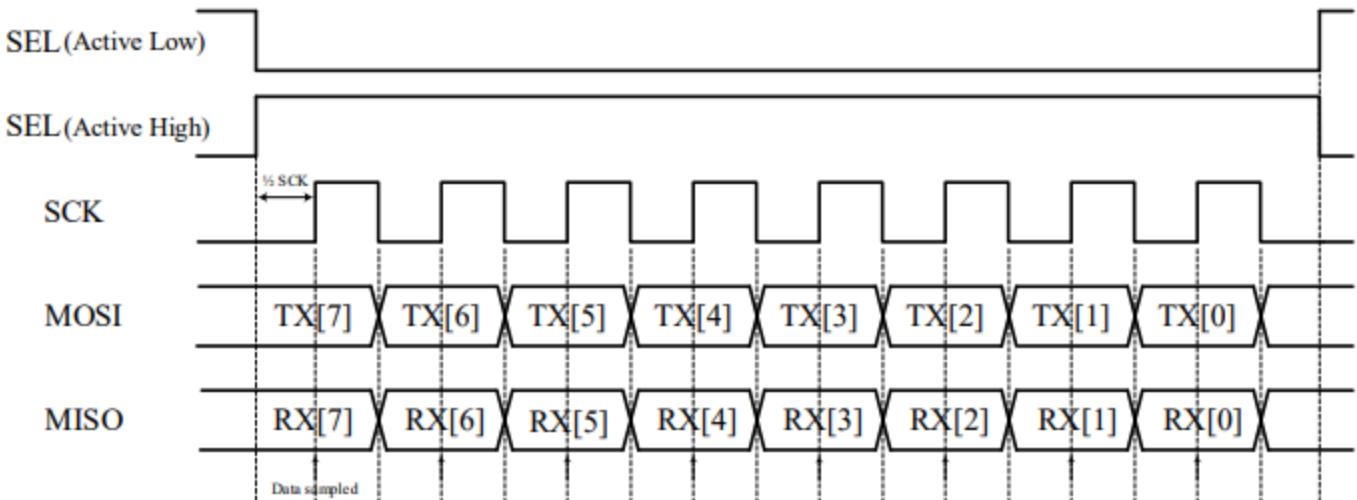
- CPOL=0，時脈極性為 0 代表 SCK 空閒狀態為“L”
- CPOL=1，時脈極性為 1 代表 SCK 空閒狀態為“H”

2、CPHA (Clock Phase)

- CPHA=0，時脈相位為 0 代表數據會在“第一個 SCK 變更準位”時被採樣
- CPHA=1，時脈相位為 1 代表數據會在“第二個 SCK 變更準位”時被採樣

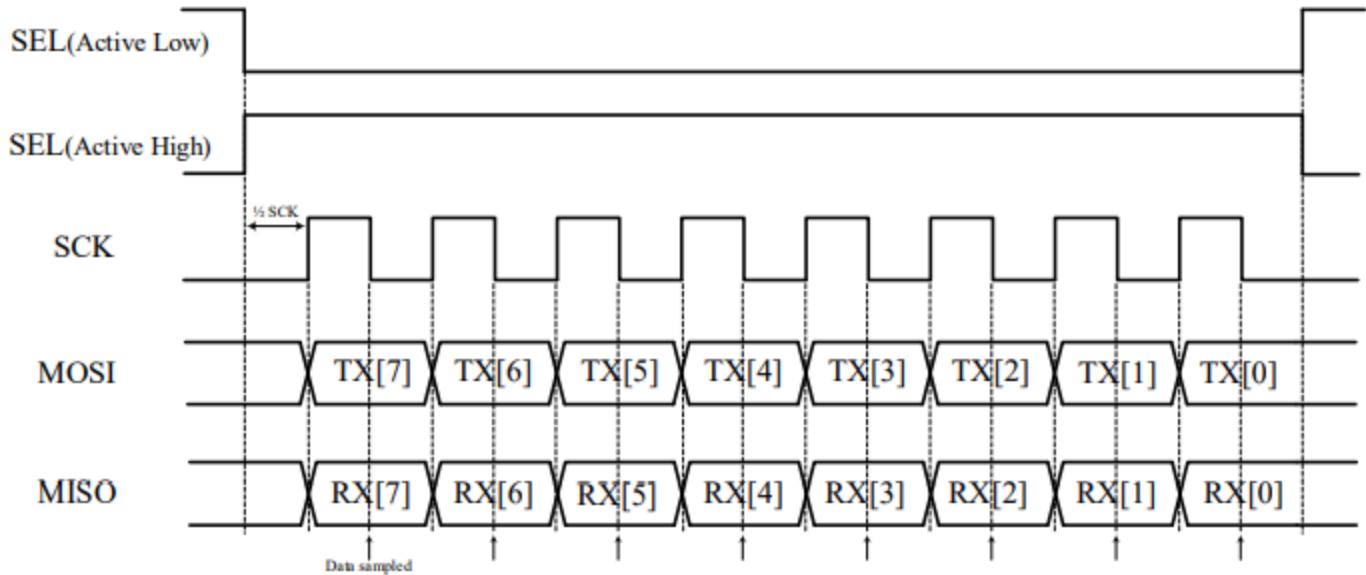
CPOL = 0 / CPHA = 0

CPOL=0、CPHA=0：數據會在 SCK 上升邊緣被採樣，在 SCK 下降邊緣時改變數據準位並在半週期內完成資料門鎖。



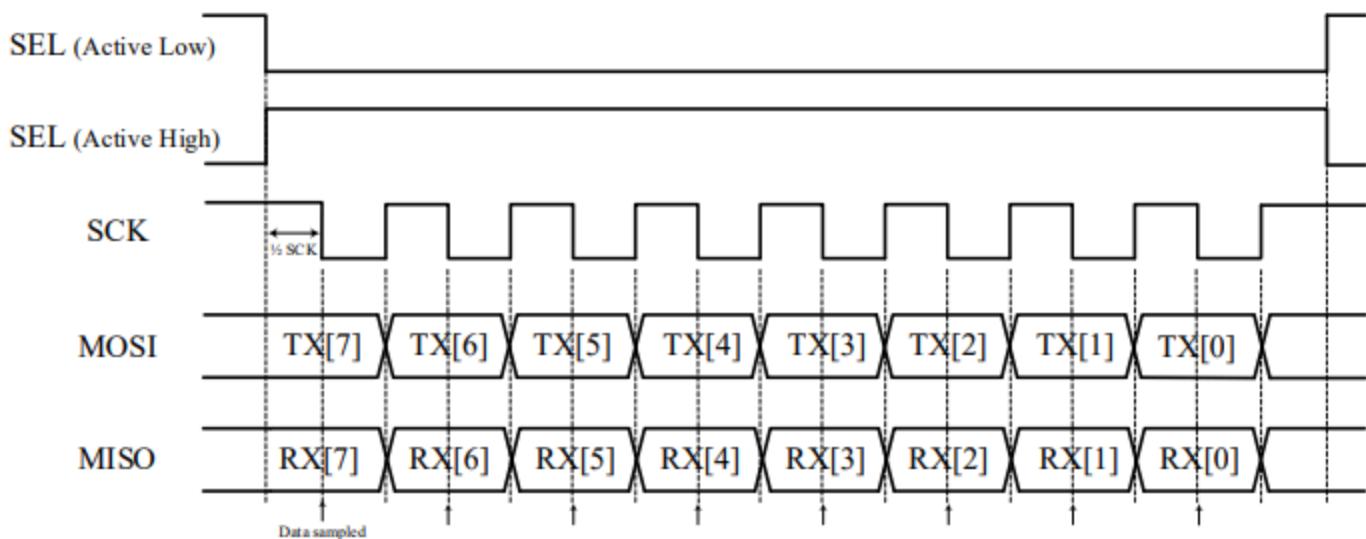
CPOL = 0 / CPHA = 1

CPOL=0、CPHA=1：數據會在 SCK 下降邊緣被採樣，在 SCK 上升邊緣時改變數據準位並在半周期內完成資料門鎖。



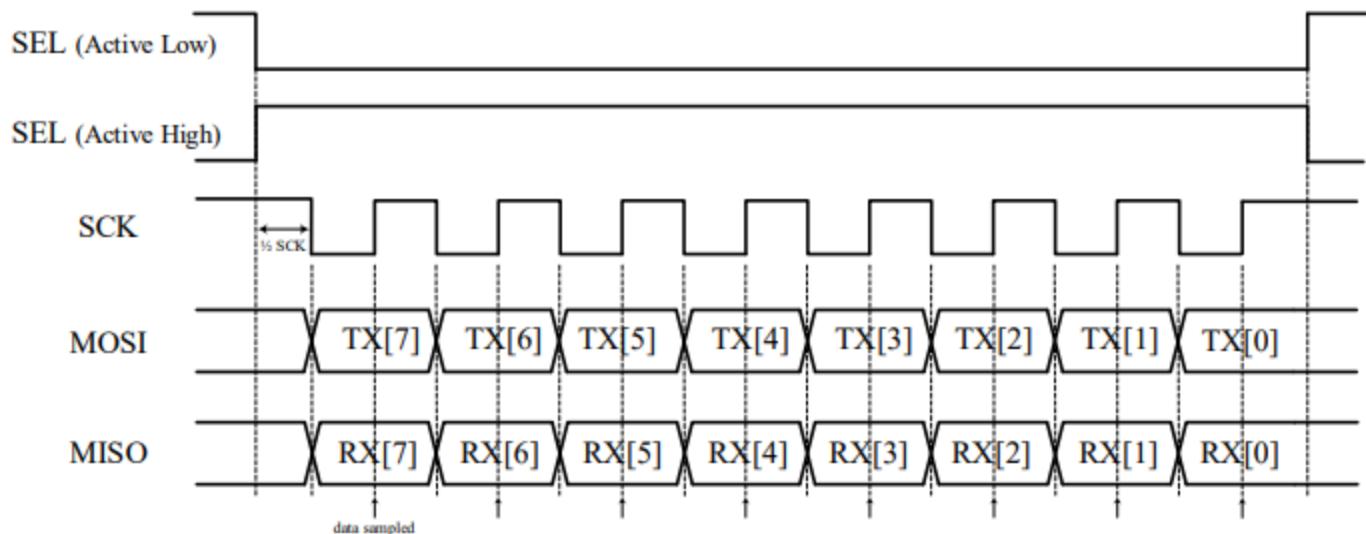
CPOL = 1 / CPHA = 0

CPOL=1、CPHA=0：數據會在 SCK 下降邊緣被採樣，在 SCK 上升邊緣時改變數據準位並在半周期內完成資料門鎖。



CPOL = 1 / CPHA = 1

CPOL=1、CPHA=1：數據會在 SCK 上升邊緣被採樣，在 SCK 下降邊緣時改變數據準位並在半週期內完成資料門鎖。



SPI 與 I2C 的比較

特性	SPI	I2C
線數	4 (SCLK, MOSI, MISO, SS/CS)	2 (SDA, SCL)
通訊模式	全雙工	半雙工
速率	數 MHz 至數十 MHz	100 kHz 至 3.4 MHz
主從架構	單主多從	多主多從
引腳輸出	推挽	開漏 (需上拉電阻)
協議複雜度	簡單，無固定格式	較複雜 (地址、ACK 等)
從設備選擇	每個從設備需獨立 SS/CS	通過地址選擇
應用場景	高速、簡單設備	低速、多設備