

1. 什麼是 C-string / <cstring>

1. C-string 定義

- a. 本質：char 陣列或 char*，用 \0（null 字元）作結尾。
- b. 例：

```
char s1[] = "Hello";    // 編譯器自動放 '\0'，長度 6 格
char s2[10] = "Hi";     // 'H' 'i' '\0' 之後都是未定義垃圾
char* s3 = "ABC";       // 指向常量字串（在教科書例子常見）
```

2. C++ 中的標頭檔

- a. C：#include <string.h>
- b. C++：#include <cstring>（之後可以寫 std::strlen...，很多教材直接寫 strlen）

3. 絕對要記住的規則

- a. 每個 C-string 必須有一個 '\0' 作結尾。
- b. 字串函式（strlen/strcpy/...）都依賴 '\0' 才會停止，沒有 '\0' 會一直讀到記憶體外、崩潰。
- c. 目標陣列（destination）一定要有足夠空間（含 '\0'）。

2. 長度函式

2.1 strlen

```
size_t strlen(const char *str);
```

- 功能：計算字串長度（不包括 '\0'）。
- 回傳：字元數目（size_t 是整數型別）。
- 範例：

```
char s[] = "Hello";
```

```
size_t n = strlen(s); // n = 5
```

3. 比較函式

3.1 strcmp

```
int strcmp(const char *s1, const char *s2);
```

- 功能：逐字元比較 s1 與 s2（區分大小寫）。
- 回傳：
 - > 0 : $s1 > s2$
 - $= 0$: 兩字串內容完全相同
 - < 0 : $s1 < s2$
- 比較的順序是「字典序」（lexicographical order），用 unsigned char 的大小比。
- 範例：

```
strcmp("ABC", "ABD"); // 負值
```

```
strcmp("ABC", "ABC"); // 0
```

```
strcmp("ABD", "ABC"); // 正值
```

3.2 strncmp

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- 功能：只比較「前 n 個字元」。
- 用法與 strcmp 一樣，只是多了長度限制，可避免讀太長。
- 範例：

```
strncmp("ABCDEFGH", "ABCxyz", 3); // 只看前三個，結果 = 0
```

4. 複製函式

4.1 strcpy

```
char* strcpy(char *dest, const char *src);
```

- 功能：將 `src` 字串連同 `'\0'` 一起複製到 `dest`。
- 回傳：`dest`（方便串接呼叫）。
- 重要：`dest` 必須有足夠空間，可以放下 `src + '\0'`，否則 `buffer overflow`。
- 範例：

```
char dest[20];  
strcpy(dest, "Hello"); // dest 變成 "Hello"
```

4.2 strncpy

```
char* strncpy(char *dest, const char *src, size_t n);
```

- 功能：最多複製 `n` 個字元。
- 情況：
 - 若 `src` 長度 $< n$ ：會複製到 `'\0'`，並把剩餘位置補 `'\0'`。
 - 若 `src` 長度 $\geq n$ ：只複製 `n` 個字元，不保證結尾有 `'\0'`。
- 使用時常配合手動補 `'\0'`：

```
char dest[10];  
strncpy(dest, src, 9);  
dest[9] = '\0'; // 確保結尾
```

5. 串接（接在後面）

5.1 strcat

`char* strcat(char *dest, const char *src);`

- 功能：把 `src` 的內容接在 `dest` 字串尾部（`dest` 原本的 `'\0'` 位置）。
- 回傳：`dest`。
- 注意：`dest` 必須有足夠空間放下原來內容 + `src` + `'\0'`。
- 範例：

```
char s[20] = "Hello";  
strcat(s, " World");    // s 變成 "Hello World"
```

5.2 strncat

`char* strncat(char *dest, const char *src, size_t n);`

- 功能：從 `src` 最多接上 `n` 個字元到 `dest` 尾部，然後加 `'\0'`。
- 同樣要確保 `dest` 空間足夠。
- 範例：

```
char s[10] = "Hello";  
strncat(s, "ABCDEF", 3); // s 變成 "HelloABC"
```

6. 搜尋字元 / 子字串

6.1 strchr：找單一字元

`char* strchr(const char *str, int ch);`

- 功能：在 `str` 中尋找第一次出現的 `ch`（會把 `ch` 轉成 `char`）。
- 回傳：
 - 找到：指向那個位置的指標
 - 找不到：`nullptr`
- 範例：

```
char s[] = "A,B,C";  
char *p = strchr(s, ','); // p 指向 s[1] 的 ','
```

6.2 `strrchr`：從後面找字元

```
char* strrchr(const char *str, int ch);
```

- 功能：找到「最後一次」出現的 `ch`。
- 用法同 `strchr`。

6.3 `strstr`：找子字串

```
char* strstr(const char *haystack, const char *needle);
```

- 功能：在 `haystack` 裏尋找子字串 `needle`（區分大小寫）。
- 回傳：
 - 找到：`haystack` 中第一個匹配位置的指標
 - 找不到：`nullptr`
- 範例：

```
char s[] = "I like C programming";  
char *p = strstr(s, "C pro"); // 指向 "C programming" 的 'C'
```

7. 字串切割（**tokenize**）

7.1 strtok

`char* strtok(char *str, const char *delim);`

- 功能：用某些分隔符（例如逗號、空格）把字串拆成一段一段「token」。
- 用法模式（必須記）：

```
char s[] = "A,B,C";
char *token = strtok(s, ",");    // 第一次傳入原字串
while (token != nullptr) {
    // 使用 token
    token = strtok(nullptr, ","); // 之後傳入 nullptr
}
```

- 注意：
 - strtok 會在原字串上寫入 '\0'，把分隔符變成結尾，破壞原字串。
 - 不是 thread-safe，不適合多線程共享同一字串。

8. 記憶體操作（**binary**，用於任意資料）

雖然不是「只處理字串」，但也在 `<cstring>` 裏，非常常用。

8.1 memset

`void* memset(void *ptr, int value, size_t num);`

- 功能：將從 ptr 開始的 num 個 byte 都填成 value（取 value 的低 8 bit）。
- 常用來初始化陣列為 0：

```
int a[100];
```

```
memset(a, 0, sizeof(a)); // 全部 bytes 設為 0
```

8.2 memcpy

```
void* memcpy(void *dest, const void *src, size_t num);
```

- 功能：從 src 複製 num 個 byte 到 dest。
- 不適合重疊區域，若 src、dest 有重疊要用 memmove。
- 范例：

```
memcpy(dest, src, n);
```

8.3 memmove

```
void* memmove(void *dest, const void *src, size_t num);
```

- 功能：與 memcpy 類似，但支援重疊區域，會安全處理。

8.4 memcmp

```
int memcmp(const void *p1, const void *p2, size_t num);
```

- 功能：比較 num 個 byte 的內容。
- 回傳規則類似 strcmp (>0、=0、<0)。

Examples on loop-design

Find the [maximum](#) value in an array of integers

Common mistake of students:

```
int findMax(int A[], int n) // n = no. of elements in A[]
{
    int max; //variable to store the result

    max = 0; // incorrect initialization

    for (int i = 0; i < n; i++)
        if (A[i] > max)
            max = A[i];

    return max; //wrong result if numbers in A[] are negative
}
```

```
unsigned wordCount(const char *a)
{
    // Count the number of words in a[].
    // A word is a consecutive sequence of non-white space
    // chars delimited by white-space char or '\0'.

    /* General loop design consideration:

        inspect each char in a[] iteratively

        if (the char is white-space char)
            what to do
        else // the char is non-white space char
            what to do
    */

    unsigned count = 0;
    bool isWS = true; // is white-space

    for (int i = 0; a[i] != '\0'; i++)
    {
        if (isspace(a[i])
            isWS = true;
        else
        {
            if (isWS) // transition from white-space to
                count++; // non-white space char
                        // a[i] is first char of a word
            isWS = false;
        }
    }
    return count;
}
```

jhuhhhh

Find the location of the minimum value in an array of integers.
If the array is empty, return -1 to represent an invalid index position.

// precondition: A[] is unordered, A[] may be empty

```
int findMinLoc_1(int A[], int n)
{
    int min;

    min = n > 0 ? 0 : -1;
    for (int i = 1; i < n; i++)
        //assert: ???
        if (A[i] < A[min])
            min = i;

    return min; // return -1 if A[] is empty
}
```

// precondition: A[] is in ascending ordered

```
int findMinLoc_2(int A[], int n)
{
    return n > 0 ? 0 : -1;
}
```

Remove duplicated values in an array of integers

Unordered array

```
A[] = {3, 6, 4, 0, 3, 3, 5, 6, 4, 3, 2, 8}
n = 12
```

After removing duplicated values

```
A[] = {3, 6, 4, 0, 8, 2, 5}
n = 7
```

Note that the order of the numbers in the resultant array is not important, because the array is unordered.

```
// Precondition: A[] is unordered
void removeDup(int A[], int& n) // n is passed by reference
{
    for (int i = 0; i < n-1; i++)
    {
        // assert: A[0..i] are distinct
        int j = i+1;
        while (j < n) // should not use for-loop, why ??
        {
            if (A[j] == A[i])
            {
                // move the last element to location j
                A[j] = A[n-1];
                n -= 1; // no. of element in A[] is reduced
            }
            else
                j++; // increment j is conditional
        }
    }
    /* Postcondition: elements in A[0..n-1] are distinct, and
       A[] is unordered. */
}
```

這一頁是在講「把兩個已排序的整數陣列合併成一個新的排序陣列」的演算法與程式。

我由上到下說明每一部分，你可以直接抄重點。

1. 問題與前置條件

標題：Merge two sorted arrays of integers

//Precondition: A[] and B[] are sorted in ascending order

前置條件（precondition）：

- A[] 和 B[] 都已經是 由小到大排序好 的整數陣列。
- 我們要把它們合併到 C[] 裡，仍然保持由小到大。

2. 函式介面

```
void merge(const int A[], int na,  
          const int B[], int nb,  
          int C[], int& nc) // nc is passed by reference
```

各參數意思：

- `const int A[]`：第一個已排序陣列 A，內容不會被修改（`const`）。
- `int na`：陣列 A 的元素數目。
- `const int B[]`：第二個已排序陣列 B。
- `int nb`：陣列 B 的元素數目。
- `int C[]`：輸出陣列 C，用來存放合併後的結果（呼叫者先建立好）。
- `int& nc`：以 **reference** 方式傳入，用來回傳「C[] 裡實際寫入了多少個元素」。

也就是說，呼叫者會先提供足夠大的 C[]，再呼叫：

```
int C[na + nb];  
int nc;  
merge(A, na, B, nb, C, nc);  
// 結束後：C[0..nc-1] 是排序好的合併結果
```

3. 區域變數與初始化

```
{  
    // array C[] has been created by the calling function
```

```
int i, j, k;
```

```
i = j = k = 0;
```

- i ：目前在 $A[]$ 裡的索引（正在看 $A[i]$ ）。
- j ：目前在 $B[]$ 裡的索引（正在看 $B[j]$ ）。
- k ：目前在 $C[]$ 裡要寫入的位置（下一個要填 $C[k]$ ）。

初始化：

- 一開始從三個陣列的開頭開始： $i = 0, j = 0, k = 0$ 。

4. 主合併迴圈

```
while (i < na && j < nb)
{
    //assert: C[0..k-1] is sorted in ascending order
    if (A[i] <= B[j])
    {
        C[k] = A[i];    // the 3 statements can be replaced
        k++;           // by C[k++] = A[i++];
        i++;
    }
    else
        C[k++] = B[j++];
}
```

4.1 迴圈條件

`while (i < na && j < nb) :`

- 只要 **A** 和 **B** 都還沒走到尾，就繼續比較。
- 一旦其中一個陣列用完（例如 `i == na`），就跳出這個迴圈。

註解：

`//assert: C[0..k-1] is sorted in ascending order`

- 「不變式（invariant）」：在每次迴圈開始時，`C[0..k-1]` 一定已經是排序好的。

4.2 內部比較與寫入

`if (A[i] <= B[j]) :`

- 比較 `A[i]` 和 `B[j]` 中誰較小（或相等）。
- 因為 `A[]`、`B[]` 都是已排序的，兩者的較小者就是目前剩餘所有數中最小的那一個。

兩種情況：

1. `A[i] <= B[j]` :

`C[k] = A[i];`

`k++;`

`i++;`

`// 等價寫法：C[k++] = A[i++];`

- a. 把 `A[i]` 放到 `C[k]`。
- b. `k++`：C 的下一個寫入位置向後移一格。
- c. `i++`：A 的指標向後移一格。

2. 否則 ($A[i] > B[j]$) :

$C[k++] = B[j++]$;

- a. 把 $B[j]$ 寫入 $C[k]$ ，然後 k 、 j 都加一。
- b. 這一行等價於：

$C[k] = B[j]$;

$k++$;

$j++$;

迴圈重複：每一次都從 $A[i]$ 、 $B[j]$ 中選出較小的數，放到 $C[k]$ ，因此 $C[]$ 一直保持排序。

5. 把剩下的元素複製到 $C[]$

當其中一個陣列已經「用完」時（例如 $A[]$ 已走完），另一個陣列裡的元素一定都比 $C[]$ 中現有的元素大，而且本身已排序，所以可以直接整段接到後面。

```
//copy the remaining elements in A[] or B[] to C[]
```

```
while (i < na)
```

```
     $C[k++] = A[i++]$ ;
```

```
while (j < nb)
```

```
     $C[k++] = B[j++]$ ;
```

- 第一次 `while (i < na)` :
 - 若 A 還有剩餘元素，就依次將 $A[i]$ 複製到 $C[k]$ ，直到 $i == na$ 。
- 第二次 `while (j < nb)` :
 - 若 B 還有剩餘元素，就依次將 $B[j]$ 複製到 $C[k]$ ，直到 $j == nb$ 。

注意：兩個迴圈最多只會有一個真正執行（因為必然是 A 或 B 其中之一先用完）。

6. 回傳 C[] 長度

```
nc = k;  
}
```

- 此時 k 正好是 C 裡填入元素的個數（下一個空位的位置），因此：
 - C 的有效元素範圍是 C[0..k-1]。
- 把 k 透過 reference 參數 nc 回傳給呼叫者。

Recursive binary search

```
// Precondition: A[] is sorted in ascending order  
// A[low..high] represents the search range  
  
int binSearch_R(int A[], int low, int high, int key)  
{  
    if (low > high) // search range is empty  
        return -1; // key is not found  
  
    int mid = (low + high) / 2; // probe the mid-point of  
    if (A[mid] == key) // the search range  
        return mid;  
  
    if (A[mid] < key) // recursion with reduced search range  
        return binSearch_R(A, mid+1, high, key);  
    else  
        return binSearch_R(A, low, mid-1, key);  
}
```

我先把這一頁在說甚麼講清楚，再逐行拆解程式，讓你之後可以自己寫在筆記上。

一、整體目標：用遞迴列出所有排列（permutations）

這頁的主題是：

給你一個字元陣列 `x[s..e]`，用遞迴列出這些符號的所有排列。

例如 `x = "abcd"`，`s = 0`，`e = 3`，就要列出：

`abcd`, `abdc`, `acbd`, `acdb`, `adbc`, `adcb`, ...（總共 $4! = 24$ 種）

思路：

1. 先決定「第一個位置」放哪一個字元；
2. 然後對「剩下的字元」再做同樣的事情（用遞迴處理）。

二、swap 函式：交換兩個字元（pass-by-reference）

```
void swap(char& a, char& b)    // swap a with b
{
    char t = a;
    a = b;
    b = t;
}
```

說明：

- 這是自己寫的交流函式，用來把兩個 `char` 的值互相交換。
- 參數型別是 `char&`：
 - `&` 代表 **reference**（參考），即「別名」。
 - 這樣 `swap(x[s], x[k])` 會直接交換陣列 `x` 裡的兩個元素，不需要回傳值。

- 交換步驟：
 - 用暫存變數 t 保存 a 原來的值；
 - 把 b 放到 a；
 - 再把 t 放回 b。

三、permute 函式：遞迴產生排列

```
void permute(char x[], int s, int e)
// Enumerate the permutations of the symbols in x[s..e]
{
    if (s == e)          // base case
        cout << x << endl;
    else                  // recursion
    {
        for (int k = s; k <= e; k++)
        {
            //1. there are e-s+1 choices for the first symbol
            swap(x[s], x[k]);

            //2. once we fix the first symbol,
            //    find the permutations of the remaining e-s
            //    symbols by recursion
            permute(x, s+1, e);

            //3. restore the original order of the array
            swap(x[s], x[k]);
        }
    }
}
```

3.1 參數意義

- `char x[]`：字元陣列（C-style 字串），例如 "abcd"。
- `int s`：目前要決定的「起始位置」（start index）。
- `int e`：子陣列的「結束位置」（end index）。

函式要做的事：

列出 `x[s..e]` 中所有字元的排列，並將結果直接印出。

3.2 Base case：if (s == e)

```
if (s == e) // base case
    cout << x << endl;
```

- 當 `s == e` 時，代表：
 - 我們已經處理到子陣列的最後一個位置；
 - `x` 內的字元順序此時已經形成一個完整排列。
- 所以直接 `cout << x` 印出整個字串。

這是遞迴的終止條件：

當只剩下一個位置要放時，不再往下遞迴，直接輸出。

3.3 遞迴步驟（recursion case）

```
else
{
    for (int k = s; k <= e; k++)
    {
```

```

//1. there are e-s+1 choices for the first symbol
swap(x[s], x[k]);

//2. once we fix the first symbol,
//   find the permutations of the remaining e-s
//   symbols by recursion
permute(x, s+1, e);

//3. restore the original order of the array
swap(x[s], x[k]);
}
}

```

核心想法：

第 s 個位置可以放子陣列 $x[s..e]$ 裡面的任何一個字元，共有 $e-s+1$ 種選擇；選定後，對剩下 $x[s+1..e]$ 再做同樣的事。

詳解：

1. for (int k = s; k <= e; k++)
 - a. k 從 s 跑到 e：
 - i. $k = s$ ：把原本的 $x[s]$ 放在第 s 位置；
 - ii. $k = s+1$ ：把 $x[s+1]$ 搬到第 s 位置；
 - iii. ...一直到 $k = e$ 。
 - b. 因此，共有 $e-s+1$ 個不同的候選字元可以放在第 s 個位置。
2. 第一步：固定第 s 個位置

```
swap(x[s], x[k]);
```

- a. 把 $x[k]$ 換到第 s 個位置上。
 - b. 這代表「現在假設第 s 個位置選擇的是原本的第 k 個字元」。
3. 第二步：對剩下位置做遞迴

```
permute(x, s+1, e);
```

- a. 現在第 s 位已固定，所以接下來要排列的是 $x[s+1..e]$ 。
- b. 於是遞迴呼叫 `permute(x, s+1, e)`。

這就是：

“once we fix the first symbol, find the permutations of the remaining $e-s$ symbols by recursion”

4. 第三步：回溯（**backtracking**）——恢復原本順序

```
swap(x[s], x[k]);
```

- a. 遞迴回來後，必須把剛才的交換復原，否則下一個 k 會在「被弄亂的陣列」上繼續操作。
- b. 再 `swap` 一次可以恢復原狀，這是典型的 **backtracking** 模式：
 - i. 做選擇（`swap`）
 - ii. 遞迴探索
 - iii. 撤銷選擇（再 `swap` 回去）

這樣當 `for` 迴圈換下一個 k 時， x 又回到「尚未選擇第 s 個位置」的原始狀態，不會相互干擾。

四、**testPermute**：測試函式

```
void testPermute()
{
    char x[] = "abcd";

    permute(x, 0, 3); // print permutations of 4 symbols
}
```

- `char x[] = "abcd";`
 - 建立一個 C-style 字串：`x[0]='a', x[1]='b', x[2]='c', x[3]='d', x[4]='\0'`。
- `permute(x, 0, 3);`
 - `s = 0`，表示從第一個字元位置開始決定；
 - `e = 3`，表示最後一個有效字元的 `index`。
 - 因此函式會列出 "abcd" 的所有 $4! = 24$ 個排列。

五、可以寫在筆記上的簡短總結

你可以整理成幾句話：

1. `permute(x, s, e)`：列出 `x[s..e]` 的所有排列。
2. Base case：`s == e` → 陣列目前的順序就是一個完整排列 → 直接 `cout << x`。
3. Recursion case：
 - a. 對 `k = s..e`：
 - i. 交換 `x[s]` 和 `x[k]`，把第 `k` 個字元放到第 `s` 位置；
 - ii. 遞迴呼叫 `permute(x, s+1, e)` 排列剩下部分；
 - iii. 再 `swap` 回來恢復原狀（backtracking）。
4. `testPermute()` 用 "abcd" 測試，會印出所有 24 種排列。

1. 整體目的：用 C++ class 來封裝「分數」

標題寫的是：

Modeling of fraction using C++ class

意思是：現在不再用 `struct` + 一堆普通函式，而是用 **class**

把「分子、分母」和它們的規則（invariants）封裝在一起，提供一個安全的 `fraction` 型別。

2. #include <ostream>

最上面：

```
#include <ostream>
```

- 這是為了使用 `std::ostream` 這個輸出串流型別（`cout` 的型別）。
- 後面要宣告 `operator<<` 的時候會用到 `ostream&`。

3. class fraction 與 friend operator<<

類別開頭大致是：

```
class fraction
{
    friend std::ostream& operator<<(std::ostream& os, const fraction& f);
    // overload operator<< so that you can output
    // a fraction object with "cout << f"
    // purpose類似 Java 的 toString()
```

3.1 friend 的意思

- 這一行是在宣告一個友元函式（friend function），用來重載輸出運算子 `<<`。
- 型別解讀：
 - 回傳：`ostream&`（通常是 `cout` 那條輸出串流本身，方便串接）。
 - 參數：
 - 第一個：`ostream& os` → 目標輸出串流，例如 `cout`。
 - 第二個：`const fraction& f` → 要輸出的 `fraction` 物件。

使用效果是：

```
fraction f(...);  
cout << f;    // 編譯器會呼叫 operator<<(cout, f);
```

為甚麼要 friend？

- 之後 numerator、denominator 會放在 private: 裡面。
- 一般「外部函式」不能直接讀 private 成員。
- 把 operator<< 宣告成 friend，代表「雖然它不是成員函式，但被賦予存取 private 成員的權限」，方便在印出時直接用 f.numerator、f.denominator。

可以把它想成：

這是 C++ 版本的「toString() + 輸出」，讓你可以用 cout << f 自然地印出分數。

4. private: 成員變數

private:

```
int numerator;    // member variable  
int denominator; // (instance variables in Java)
```

- numerator：分子
- denominator：分母

為甚麼要放在 private:？

- 這樣外部程式不能直接隨便改，必須透過 class 提供的建構子 / 成員函式來設定。
- 好處是可以在這些介面裡強制執行「表示不變量（representation invariants）」：
 - 分母不能為 0；
 - 分數要保持約簡形式；

- 負號放置規則一致（例如分母永遠為正）等等。

這就是「資料封裝」與「隱藏實作細節」。

5. public:：建構子與對外介面

5.1 預設建構子：fraction()

public:

```
fraction()    // default constructor, no explicit parameter
{
    numerator = 0;
    denominator = 1;
}
```

- 這是**預設建構子**：沒有參數，在你寫 `fraction f;` 時自動被呼叫。
- 它把 `fraction` 初始化成 0/1：
 - 分子 = 0
 - 分母 = 1
- 這樣一來，每個被建立出來的 `fraction` 物件一開始就是合法的分數，不會有「未初始化的垃圾值」。

5.2 帶參數建構子：fraction(int n, int d)

投影片下一段：

```
fraction(int n, int d) // ensure conformant with the
                        // representation invariants
{
    if (d == 0)
    {
```

```

        cerr << "ERROR: denominator is zero." << endl;
        exit(0); // terminate the program
    }
    if (n == 0)
    {
        numerator = 0;
        denominator = 1;
    }
    ...
}

```

（下面的 `else` 和約簡動作可能在下一頁繼續。）

5.2.1 介面意義

- 這個建構子讓你可以這樣建立一個分數：

```

fraction f1(1, 2); // 代表 1/2
fraction f2(-3, 4); // 代表 -3/4

```

- `n` 是外面傳入的「分子」，`d` 是「分母」。

註解說：

ensure conformant with the representation invariants

意思是：在這裡會檢查並修正

`n`、`d`，確保建構出來的物件一定符合事先定下來的「內部規則」。

5.2.2 檢查分母是否為 0

```

if (d == 0)
{
    cerr << "ERROR: denominator is zero." << endl;
    exit(0);
}

```

```
}
```

- 若 `d == 0` :
 - 用 `cerr` (`error stream`) 輸出錯誤訊息。
 - 呼叫 `exit(0)`; 直接終止整個程式。
 - 實務上可能會用 `throw` 丟 `exception` ; 這裡用最簡單的方式讓你知道「這是致命錯誤」。

這裡是在強化一個**關鍵 invariant** :

任何時候，一個合法的 `fraction` 的 `denominator` 絕對不能是 0。

5.2.3 處理分子為 0 的情況

```
if (n == 0)
{
    numerator = 0;
    denominator = 1;
}
```

- 如果傳入的分子 `n` 是 0，無論 `d` 是多少（非 0），數值上這個分數都是 0。
- 為了讓內部表示形式統一，作者將它固定成 0/1 :
 - 分子設成 0；
 - 分母設成 1。

這也屬於「表示不變量」的一部分：

只要分子為 0，內部就統一存成 0/1，而不是 0/3、0/-5 這種各種版本。

我幫你把第 9 頁整段 `code` 從上到下說清楚，主要是在說明：

在 `class fraction` 裡面怎樣「重載運算子」(`operator==`、`operator+`) 以及一個 `print()` 成員函式，最後順便說明為甚麼這個 `class` 不需要自己寫 `destructor`。

為方便對照，我先用接近投影片的樣子寫出來（邏輯上等價）：

```

// overload (redefine) the operators
bool operator==(const fraction& other)
{
    return (this->numerator == other.numerator) &&
           (this->denominator == other.denominator);
    // Pointer "this" points to the implicit object.
}

fraction operator+(const fraction& other)
{
    int n = numerator * other.denominator +
           other.numerator * denominator;

    int d = denominator * other.denominator;
    fraction r(n, d); // Note that r is a local variable !!
    return r;         // r ceases to exist after function return.
                      // It is OK if the function returns by-value.
}

void print()
{
    cout << numerator << "/" << denominator;
}

// Other operators and functions in the class (not shown).
// Remark: No need to define a destructor for this class
// because creation of object instance does not involve
// dynamic memory allocation.

}; // end of class definition

```

下面逐段解釋。

1. operator==：比較兩個 fraction 是否相等

```
bool operator==(const fraction& other)
{
    return (this->numerator == other.numerator) &&
           (this->denominator == other.denominator);
    // Pointer "this" points to the implicit object.
}
```

1.1 這是甚麼？

- 這是一個 **成員函式 (member function)**，用來重載 == 運算子。
- 形參：const fraction& other
 - 代表「右邊那個分數」，例如 f1 == f2 裡面的 f2。
- 回傳型別是 bool：
 - true：兩個分數「在內部表示上」完全相同；
 - false：否則。

1.2 誰是左邊？誰是右邊？

對於成員版的 operator==：

f1 == f2

會被編譯器解讀為：

f1.operator==(f2);

- 左邊的 f1 是「**隱式物件 (implicit object)**」；

- 右邊的 f2 對應到形參 other。

在函式內，隱式物件透過 **指標 this** 來存取：

- this 是 fraction*；
- this->numerator 就是「左邊那個 fraction 的 numerator」；
- other.numerator 是右邊傳進來的那個 fraction 的分子。

其實這裡 this-> 可以省略，寫成：

```
return numerator == other.numerator &&  
       denominator == other.denominator;
```

效果一樣，老師只是藉機提醒你：每個成員函式裡都有一個隱含的 **this** 指標。

1.3 注意：這裡比較的是「表示」而不是「數值」

- 這個 operator== 是直接比較 numerator、denominator 是否完全相同。
- 前面建構子已經把 fraction 統一約簡、處理好正負號，所以：
 - 若數值相等，內部表示也應該一樣，例如都變為 1/2；
 - 像 2/4 不會被保留，它會在建構時被約成 1/2。
- 正因為 class 保證了 **representation invariants**，這種簡單比較才是正確的。

2. operator+：兩個 fraction 相加

```
fraction operator+(const fraction& other)  
{  
    int n = numerator * other.denominator +  
           other.numerator * denominator;  
  
    int d = denominator * other.denominator;  
    fraction r(n, d); // local variable
```

```
    return r;          // OK to return by value
}
```

2.1 這是甚麼？

- 這是成員函式版本的 `operator+`，用來實現「分數加法」。
- 用法：

```
fraction f3 = f1 + f2;  // 等價於 f1.operator+(f2);
```

同樣：f1 是隱式物件（左邊），f2 對應 other。

2.2 計算分子與分母

數學上：

[

$$\frac{a}{b} + \frac{c}{d} = \frac{a d + c b}{b d}$$

]

對應程式：

- 左邊 f1：
 - `numerator = (a)`
 - `denominator = (b)`
- 右邊 other：
 - `other.numerator = (c)`
 - `other.denominator = (d)`

因此：

```
int n = numerator * other.denominator +
      other.numerator * denominator;  // a*d + c*b
```

```
int d = denominator * other.denominator; // b*d
```

- n：新分子；
- d：新分母。

2.3 建立結果物件 r

```
fraction r(n, d); // Note that r is a local variable !!
```

- 用之前定義的建構子 `fraction(int n, int d)` 建立一個暫時物件 r。
- 建構子會：
 - 檢查分母是否為 0；
 - 若分子為 0，統一成 0/1；
 - 一般情況下會約簡、調整正負號。
- 這樣 r 就是一個合法且已約簡的分數。

2.4 為甚麼可以 return r (local 變數) ？

```
return r; // r ceases to exist after function return.  
// It is OK if the function returns by-value.
```

- r 是區域變數，生命週期只存在於 `operator+` 裡。
- 但 `return r;` 的語意是「以 值傳回 (by-value)」：
 - 編譯器會先用 r 建立一個回傳值的拷貝；
 - 然後函式結束，r 自己被銷毀；
 - 呼叫者拿到的是拷貝，不會受影響。

在現代 C++ 中，多數情況還會做 **return value optimization (RVO)**，避免多餘的拷貝，但對你來說只要記住：

「回傳 local 物件 by-value 是安全的，因為回傳的是它的拷貝，不是那個變數本身。」

3. print()：簡單輸出分數

```
void print()
{
    cout << numerator << "/" << denominator;
}
```

- 這是一個 public 成員函式。
- 用法：

```
fraction f(1, 2);
f.print(); // 輸出：1/2
```

- 它直接把 numerator、denominator 透過 cout 輸出為 a/b 的形式。
- 注意：
 - print()
本身沒有輸出換行（newline），所以呼叫者如果要換行，應該再寫 cout << endl; 或 "\n"。
 - 和前面宣告的 friend operator<< 不一樣：
 - print() 用法是 f.print();
 - operator<< 則是 cout << f;，更符合 C++ 流式輸出的風格。

4. 為甚麼「不需要 destructor」？

```
// Other operators and functions in the class (not shown).
// Remark: No need to define a destructor for this class
// because creation of object instance does not involve
// dynamic memory allocation.
```

```
}; // end of class definition
```

這段註解在說：

1. 這個 **class** 其實還可以有更多 **operator**（如 -, *, /）和函式（如 `reduce()` 等），只是沒列出。
2. 這個 **class** 不需要自訂 **destructor**，原因是：
 - a. 類別裡只有兩個 `int` 成員：

```
int numerator;
```

```
int denominator;
```

- b. 建立物件時並沒有用到 `new`、`malloc`、開檔案、開 `socket` 等「外部資源」；
- c. 所以只要用編譯器自動產生的 **default destructor** 就足夠：
 - i. 它會自動解除這兩個 `int`，沒有額外要釋放的東西；
 - ii. 不會有 `memory leak` 的問題。

只有在 **class** 內部持有「動態配置資源」時（例如 `int* data;` 指向 `heap` 上的陣列），你才「必須」寫 **destructor** 負責 `delete[] data;`。

5. 你可以怎樣在筆記裡總結這一頁？

建議你寫成幾個重點句，方便之後溫習：

1. `bool fraction::operator==(const fraction& other) :`
 - a. 成員版 `==`，左邊物件透過隱含指標 `this` 存取；
 - b. 直接比較 `numerator`、`denominator` 是否相同（假設建構子已保證約簡與規則）。
2. `fraction fraction::operator+(const fraction& other) :`
 - a. 成員版 `+`，實作 $((a/b) + (c/d) = (ad + cb)/(bd))$ ；

- b. 用 `fraction(n, d)` 建立暫時物件 `r`，建構子負責約簡；
 - c. `return r`; 以值傳回，雖然 `r` 是 `local`，但回傳的是拷貝，安全。
- 3. `void fraction::print()`：
 - a. 簡單輸出 `"numerator/denominator"` 到 `cout`。
- 4. 由於 `class` 內沒有動態記憶體或其它資源，不需要自訂 **destructor**，預設 **destructor** 已足夠。

1. **this** 是甚麼？

在「每一個非 **static** 成員函式」裡面，C++ 會自動幫你準備一個隱藏的指標，名字就叫 **this**。

以這個成員函式為例：

```
bool operator==(const fraction& other)
{
    return (this->numerator == other.numerator) &&
           (this->denominator == other.denominator);
}
```

假設你在外面寫：

```
fraction f1(1, 2);
```

```
fraction f2(1, 2);
```

```
bool same = (f1 == f2);
```

編譯器實際上會把 `f1 == f2` 翻譯成：

```
bool same = f1.operator==(f2);
```

在這個呼叫裡：

- `f1` 就是「呼叫這個函式的物件」。
- 在函式內部，**this** 指向的就是 `f1`，型別是 `fraction*`。
- 所以：
 - `this->numerator` 等價於 `f1.numerator`
 - `this->denominator` 等價於 `f1.denominator`

也就是說：

```
return (this->numerator == other.numerator) &&  
       (this->denominator == other.denominator);
```

其實就是在做：

```
return (f1.numerator == f2.numerator) &&  
       (f1.denominator == f2.denominator);
```

你也可以省略 **this->**，直接寫：

```
return (numerator == other.numerator) &&  
       (denominator == other.denominator);
```

效果一樣，只是 `slide` 特地加上 `this->`
讓你知道「這邊用到的是左邊那個物件的成員」。

2. other 是甚麼？

`other` 只是這個成員函式的參數名字：

```
bool operator==(const fraction& other)
```

- 型別：`const fraction&`
 - `reference`（參考），代表「右邊那個 `fraction`」，不會複製整個物件，比較有效率；
 - 前面有 `const`，表示在函式內不能改 `other`。

還是用剛才的例子：

```
fraction f1(1, 2);  
fraction f2(1, 2);  
bool same = (f1 == f2);
```

呼叫對應成：

```
same = f1.operator==(f2);
```

對應關係：

- `this` → 指向 `f1`（左邊那個）
- `other` → 參考到 `f2`（右邊那個）

所以你可以記：

在成員運算子 `bool fraction::operator==(const fraction& other)` 裡面，

`this` 代表左邊那個物件，`other` 代表右邊那個物件。

在 `operator+` 也是完全同一個概念。

3. `operator+` 是不是函式名？

是的，`operator+` 就是一個函式名稱，只是這個名稱有特別意義，編譯器會把「使用 + 運算子」對應到這個函式。

成員版宣告長這樣：

```
fraction operator+(const fraction& other);
```

可以解讀成：

- 回傳型別：fraction
- 函式名：operator+
- 參數：const fraction& other

呼叫方式有兩種你可以這樣理解：

1. 平常寫法（人類看）：

```
fraction f3 = f1 + f2;
```

2. 編譯器看的寫法：

```
fraction f3 = f1.operator+(f2);
```

所以在 operator+ 裡面：

```
fraction operator+(const fraction& other)
{
    int n = numerator * other.denominator +
            other.numerator * denominator;
    int d = denominator * other.denominator;
    fraction r(n, d);
    return r;
}
```

對於 f1 + f2：

- this → 指向 f1（左邊）；
- other → 就是 f2（右邊）；
- numerator/denominator 其實就是 this->numerator / this->denominator，也就是 f1

的分子分母；

- `other.numerator / other.denominator` 就是 `f2` 的分子分母。

我幫你把整個 `main()` 由上到下說一次，你可以直接把重點抄下來。

1. 檔案包含

```
//Example program that makes use of arrayListType
```

```
#include "arrayListType.h"
```

- 註解：這是一個「示範如何使用 `arrayListType`」的程式。
- `#include "arrayListType.h"`：把之前定義好的範本類別 `arrayListType` 的宣告載入進來，這樣 `main()` 才能使用 `arrayListType<...>`。

2. 建立兩個 list 物件

```
int main()
{
    arrayListType<int> intList(20);

    arrayListType<fraction> fractionList(50);
```

2.1 `arrayListType<int> intList(20);`

- `arrayListType` 是一個 **template class**。
- `<int>` 表示：把裡面的型別參數 `Type` 代入 `int`，產生「整數版」的陣列 `list`。

- `intList(20)` 呼叫建構子，傳入 20：
 - 通常意思是：這個 **list** 的最大容量 **maxSize** 是 **20**；
 - 內部會動態配置一塊 `new int[20]` 的記憶體來存放元素。

2.2 `arrayListType<fraction> fractionList(50);`

- `<fraction>` 表示：把 `Type` 代入你之前定義的 `class fraction`，產生「fraction 版」的 `list`。
- `fractionList(50)`：容量 50，內部會配置 `new fraction[50]`。

所以到這裡為止：

- `intList`：可以存最多 20 個 `int`。
- `fractionList`：可以存最多 50 個 `fraction` 物件。

3. 填入整數 1..19 到 `intList`

```
for (int i = 1; i < 20; i++)  
    intList.insert(i);
```

```
intList.print();
```

3.1 for 迴圈

- `i` 從 1 到 19 (`i < 20`)：
 - 每次呼叫 `intList.insert(i)`;
- 這會把整數 1, 2, 3, ..., 19 依次插入 `intList`。
- 插入的實際行為（塞尾、或是保持排序等）取決於 `arrayListType` 裡 `insert` 的實作；

這裡只是示範「`template` 一樣可以放 `int`」。

3.2 intList.print();

- 呼叫 intList 的 print() 成員函式，把目前 list 中的所有整數輸出到螢幕。
- 效果類似印出整個陣列內容，例如：

1 2 3 ... 19

（實際格式要看 print() 怎麼寫。）

4. 填入 fraction 到 fractionList

```
for (int t = 1; t < 50; t++)  
{  
    fraction f(t, t+1);  
    fractionList.insert(f);  
}  
  
fractionList.print();  
  
...  
}
```

4.1 產生一批 fraction 物件並插入

```
for (int t = 1; t < 50; t++)  
{  
    fraction f(t, t+1);  
    fractionList.insert(f);  
}
```

- 這個迴圈把 t 由 1 跑到 49。
- 每一次：
 - `fraction f(t, t+1);`
 - 呼叫你之前定義的 `fraction(int n, int d)` 建構子；
 - 建立一個分數 $f = t / (t+1)$ ，例如：
 - $t = 1 \rightarrow f = 1/2$
 - $t = 2 \rightarrow f = 2/3$
 - ...
 - $t = 49 \rightarrow f = 49/50$
 - 建構子會負責檢查分母、約簡等（前面 slide 已說）。
 - `fractionList.insert(f);`
 - 把這個 `fraction` 物件插入到 `fractionList` 這個 list 裡。

所以最後 `fractionList` 會包含一堆分數 $(1/2, 2/3, \dots, 49/50)$ 。

4.2 印出 fraction list

`fractionList.print();`

- 呼叫 `fractionList` 的 `print()`，依次把 list 中的所有 `fraction` 印出。
- `arrayListType<fraction>::print()` 的實作裡，應該會用 `cout << list[i];`，

這又會呼叫你寫的 `operator<<(ostream&, const fraction&)`，把每個分數以 a/b 形式輸出。

我先講整體想法，再按程式區塊解釋，會盡量短和清楚。

0. 這兩頁在做甚麼？

前面 `arrayListType<Type>` 是「無序陣列 list」。

p.8-9 是用 **繼承 (inheritance)** 做出一個「有序陣列 list」：

```
class ordered_arrayListType : public arrayListType<Type>
```

然後把 base class 的 `search / insert / remove / removeAt`

改寫 (override) 成「適合有序陣列」的版本，`search` 就改用 **binary search**。

1. 檔頭部分：guard + include + 類別宣告

```
// filename: orderedArrayListType.h
#ifndef ORDERED_ARRAY_LIST_TYPE_H
#define ORDERED_ARRAY_LIST_TYPE_H

#include "arrayListType.h"
```

- `#ifndef / #define / #endif`：**header guard**，防止這個 .h 被重複 include。
- `#include "arrayListType.h"`：因為要繼承 `arrayListType<Type>`，所以先把它宣告包含進來。

2. 類別定義與建構子

```
template <class Type>
class ordered_arrayListType : public arrayListType<Type>
{
    // no additional variables required in this example
```

public:

```
ordered_arrayListType() : arrayListType<Type>() { }
```

```
ordered_arrayListType(int n) : arrayListType<Type>(n) { }
```

```
// override the virtual functions in base class
```

```
int search(const Type& item);
```

```
void insert(const Type& item);
```

```
void remove(const Type& item);
```

```
void removeAt(int loc);
```

```
};
```

重點：

1. `template <class Type>`：這還是個 **template 類別**，內部元素型別叫 `Type`。
2. `: public arrayListType<Type>`：用 **public 繼承**，表示：
 - a. `ordered_arrayListType<Type>` 有 `arrayListType<Type>` 的所有資料成員（如 `list`, `length`, `maxSize`）和函式。
 - b. 在 `base class` 裡那些 `virtual` 函式，這裡會「覆寫」成新的版本。
3. 兩個建構子：
 - a. `ordered_arrayListType() : arrayListType<Type>() { }`
 - i. 呼叫 `base class` 的 `default constructor`，初始化 `list` 等。
 - b. `ordered_arrayListType(int n) : arrayListType<Type>(n) { }`
 - i. 把 `n` 傳給 `base class` 建構子，通常是設定容量 `maxSize = n`、`list = new Type[n]`。
4. 後面四條宣告：
 - a. `search / insert / remove / removeAt`：

在這裡宣告要覆寫 `base class` 的四個 **virtual 函式**，實作會放在後面（p.8–9）。

3. search 的實作：用 binary search

接下來是 p.8 底至 p.9 的程式：

```
template<class Type>
int ordered_arrayListType<Type>::search(const Type& item)
{
    // use binary search
    int low = 0;
    int high = length - 1;
    int mid;
    bool found = false;

    while (low <= high && !found)
    {
        mid = (low + high) / 2;

        if (list[mid] == item)
            found = true;
        else if (list[mid] > item)
            high = mid - 1;
        else
            low = mid + 1;
    }

    if (found)
        return mid;
    else
        return -1;
}
```

逐行看：

1. `template<class Type> + ordered_arrayListType<Type>::search`
 - a. 這是 `template` 類別成員函式的定義。
2. `int low = 0;`
 - a. `low`：目前搜尋區間的「左邊界」，初值是第一個元素 `index 0`。
3. `int high = length - 1;`
 - a. `high`：目前搜尋區間的「右邊界」，初值是最後一個元素 `index length - 1`。
 - b. `length` 是從 `base class` 繼承來的，代表 `list` 現在有多少元素。
4. `bool found = false;`
 - a. 記錄「是否已找到 `item`」。
5. `while (low <= high && !found) :`
 - a. 當前搜尋區間還有效 (`low <= high`)，而且還沒找到，就繼續 `binary search`。
6. `mid = (low + high) / 2;`
 - a. 取中間位置 `index mid`。
7. 三種情況：

`if (list[mid] == item)`

`found = true;`

`else if (list[mid] > item)`

`high = mid - 1;`

`else`

`low = mid + 1;`

- a. `list[mid] == item`：剛好找到目標，`found = true`。
- b. `list[mid] > item`：因為 `list` 是 **升序有序**，所以 `item` 如果存在，一定在左半邊 → 更新 `high = mid - 1`。
- c. 否則 `list[mid] < item`：`item` 若存在，一定在右半邊 → 更新 `low = mid + 1`。

這就是標準的 **二分搜尋 (binary search)**。

8. 迴圈結束後：

```
if (found)
    return mid;
else
    return -1;
```

- a. 若 `found == true`，傳回找到的位置 `mid`。
- b. 否則傳回 `-1` 表示「不存在」。

4. 為甚麼這樣就「**override base class 的 virtual search**」？

在 base class `arrayListType<Type>` 中，`search` 是這樣宣告的（大約）：

```
virtual int search(const Type& item);
```

在 derived class 中，我們定義：

```
int ordered_arrayListType<Type>::search(const Type& item) { ... }
```

- 函式名稱、參數型別、回傳型別都一樣 → 自然就覆寫了 base class 的那個 virtual 函式。
- 所以之後不論你用的是：

```
ordered_arrayListType<int> oList;
arrayListType<int>* p = &oList;
p->search(10);    // 會呼叫「binary search」這個版本
```

因為 base class 的 `search` 是 virtual，所以會在「執行期」選擇 derived class 的實作。

好的，我直接針對你畫面上的三個函式 `insert`、`remove`、`removeAt` 做「短而清楚」的說明。

1. `insert(const Type& item)`：插入新元素，保持有序

```
template<class Type>
void ordered_arrayListType<Type>::insert(const Type& item)
{
    if (isFull())
        return;

    int p = search(item);    // determine insertion point
    if (p < 0)                // O(log n) key comparison
        p = -(p+1);

    for (i = length-1; i >= p; i--)    // Shifting takes O(n) time
        list[i+1] = list[i];

    list[p] = item;
    length++;
}
```

逐行解釋：

1. `if (isFull()) return;`
 - a. 如果 `list` 已經滿了，就直接離開，不插入。
2. `int p = search(item);`
 - a. 用「二分搜尋」找 `item` 應該在的位置。
 - b. `search` 的設計是：
 - i. 找到：回傳元素所在 `index` (≥ 0) ；
 - ii. 找不到：回傳一個負值 $-(\text{插入位置} + 1)$ 。

3. `if (p < 0) p = -(p+1);`
 - a. 若 `<0` 表示原本不存在，這一行把它「還原」成真正的插入位置 **index**。
 - b. 例如 `search` 回 `-3` $\rightarrow p = -(-3+1)=2$ ，表示應插在 `index 2`。
4. `for (i = length-1; i >= p; i--) list[i+1] = list[i];`
 - a. 從最後一個元素往前搬，到位置 `p` 為止。
 - b. 把 `[p..length-1]` 所有元素整體往右移一格，騰出 `p` 這個空位。
5. `list[p] = item;`
 - a. 把新元素放到位置 `p` 裡。
 - b. 因為 `p` 是按大小算出來，所以放進去後整個 `list` 仍然「由小到大」有序。
6. `length++;`
 - a. 元素數量加一。

小結：

`insert`：用 `binary search` 找插入位置 ($O(\log n)$)，再搬動元素騰位 ($O(n)$)，最後把新值塞入，確保陣列仍然保持排序。

2. `remove(const Type& item)`：刪除「值等於 `item` 的第一個元素」

```
template<class Type>
void ordered_arrayListType<Type>::remove(const Type& item)
{
    int k = search(item);
    if (k >= 0)
    {
        for (int i = k; i < length-1; i++)
            list[i] = list[i+1];
        length--;
    }
}
```

```
}
```

1. `int k = search(item);`
 - a. 用二分搜尋找 `item` 的 `index`。
 - b. 找到 $\rightarrow k \geq 0$ ；找不到 $\rightarrow k < 0$ 。
2. `if (k >= 0) { ... }`
 - a. 只在找到時才做刪除。
3. `for (int i = k; i < length-1; i++) list[i] = list[i+1];`
 - a. 從位置 `k` 開始，把後面的元素往左搬一格，覆蓋掉 `k` 的元素。
 - b. 等於刪除 `list[k]`，其後所有元素往前補位。
4. `length--;`
 - a. 長度減一。

小結：

`remove(item)`：先用 `binary search` 找到 `item` 的位置，如果存在，就把後面的元素往左移，覆蓋掉它。

3. `removeAt(int loc)`：刪除「指定 `index` 位置」的元素

```
template<class Type>
void ordered_arrayListType<Type>::removeAt(int loc)
{
    if (loc >= 0 && loc < length)
    {
        for (int i = loc; i < length-1; i++)
            list[i] = list[i+1];
        length--;
    }
}
```

1. `if (loc >= 0 && loc < length)`
 - a. 先檢查 `loc` 是否是合法 `index`（在 `0..length-1` 之間）。
 - b. 不合法就什麼都不做。
2. `for (int i = loc; i < length-1; i++) list[i] = list[i+1];`
 - a. 和前一個 `remove` 很像，只是起點改成指定的 `loc`。
 - b. 從 `loc` 開始，把後面所有元素往左搬。
3. `length--;`
 - a. 長度減一。

很好，這段 `code` 我用很短、一步一步幫你看：

```
template<class Type>
void ordered_arrayListType<Type>::removeAt(int loc)
{
    if (loc >= 0 && loc < length)
    {
        for (int i = loc; i < length-1; i++)
            list[i] = list[i+1];
        length--;
    }
}
#endif
```

1. 函式是甚麼？

- `template<class Type>`
- 這個函式是模板函式，`Type` 可以是 `int`、`double`、`fraction` 等任何型別。
- `void ordered_arrayListType<Type>::removeAt(int loc)`

這是 `ordered_arrayListType` 類別裡的一個成員函式，名字叫 `removeAt`，

功能：刪除位置在 `loc` 這個 `index` 的元素。

2. 檢查 loc 合不合法

```
if (loc >= 0 && loc < length)
```

- 只在 loc 落在 $[0, \text{length}-1]$ 之間時才刪除。
- 如果 $\text{loc} < 0$ 或 $\text{loc} \geq \text{length}$ ，就什麼都不做，避免 out-of-range。

3. 把後面的元素往前搬一格

```
for (int i = loc; i < length-1; i++)  
    list[i] = list[i+1];
```

假設原本是：

$[a_0, a_1, a_2, a_3, a_4]$ 且 $\text{length} = 5$ ，你要 `removeAt(2)`（刪掉 a_2 ）。

迴圈做的事是：

- $i = 2$: `list[2] = list[3]`; \rightarrow 變成 $[a_0, a_1, a_3, a_3, a_4]$
- $i = 3$: `list[3] = list[4]`; \rightarrow 變成 $[a_0, a_1, a_3, a_4, a_4]$

搬完後，前四個變成 $[a_0, a_1, a_3, a_4]$ ，最後一格多出來但不要了。

4. 更新長度

```
length--;
```

- 把 `length` 減一，現在有效元素是前 `length` 個。

- 物理上陣列仍然有那麼多格，但邏輯上最後那一格已經「被丟掉」。

5. #endif

- 這是對應檔案開頭的 #ifndef ... #define ... ，

表示「**header guard 結束**」，防止這個 .h 被重複 include 。

一句話總結這個函式：

`removeAt(loc)`：如果 `loc` 是合法 `index`，就從 `loc` 開始，把後面所有元素往前移一格，然後把 `length` 減一，等於刪除了位置 `loc` 的那個元素。

```
// 1. search for the node holding x with a while-loop
```

```
// 2. if x is found, remove the node
```

```
//    (need to know its predecessor)
```

```
bool remove(node*& list, int x) // setter function
```

```
{
```

```
    node* cur = list;
```

```
    node* prev = nullptr;    // prev 指向 cur 的前一個節點
```

```
    while (cur != nullptr && cur->info != x) // 順序搜尋
```

```
    {
```

```

    prev = cur;           // 向前推進：prev 落後一格

    cur = cur->link;       // cur 指向下一個節點
}

// 走完 while 之後：

// - 如果 cur == nullptr：代表整條 list 都找完，沒有人 info == x

// - 如果 cur != nullptr：代表 cur 指向的節點就是要刪除的那個（info == x）

if (cur != nullptr)      // 找到節點，cur->info == x
{
    if (prev != nullptr)  // 情況 1：要刪的不是第一個節點

        prev->link = cur->link; // 跳過 cur，讓 prev 直接連到 cur 之後的節點

    else                  // 情況 2：要刪的是第一個節點（head）

        list = cur->link; // 把 head（list）改成第二個節點

    delete cur;          // 釋放被刪節點的動態記憶體

    return true;          // 成功刪除，回傳 true
}

return false;            // 沒找到 x，所以沒有刪除，回傳 false
}

```

