**Ch9. 堆疊 Stack**

接下來要進入堆疊 Stack 的部分。

課程大綱

A. 堆疊 Stack 與佇列 Queue 簡介

B. 堆疊 Stack 簡介

C. 堆疊 Stack 實作

D. C++ STL 中的堆疊

E. 堆疊 Stack 應用

首先會簡介堆疊 Stack 和佇列 Queue ，這兩種結構十分類似，差別只在新增資料和刪除資料方向上的不同。

再來，會介紹堆疊 Stack 有哪些實際的應用與其架構，緊接著，來看如何實作出一個堆疊，以及在 C++ STL 裡面要怎麼呼叫出一個堆疊來使用，最後，再來看幾個堆疊的實際應用。

**第一節：堆疊與佇列簡介**

1. 堆疊與佇列的特色

A. 堆疊 Stack 與佇列 Queue 相同處

a. 都只能操作兩端的值

b. 不支援搜索

B. 堆疊 Stack

插入、刪除在同側：Last-in-first-out（LIFO）

C. 佇列 Queue：

插入、刪除在不同側：First-in-first-out（FIFO）

堆疊和佇列這兩種結構都限制了新增資料的方向，在取用資料時，只能操作「兩端的值」。

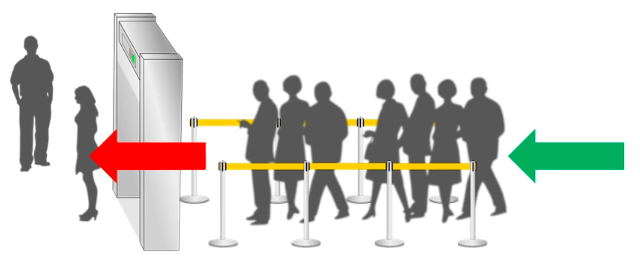
堆疊的操作方式很像洗完盤子後，把盤子一個個堆在一起，每個盤子都會在洗好的時候被放到最上面，下次要用的時候，也是從最上面把盤子拿出來：放盤子的時候，是從「最上面放」，拿的時候也是從「最上面拿」。

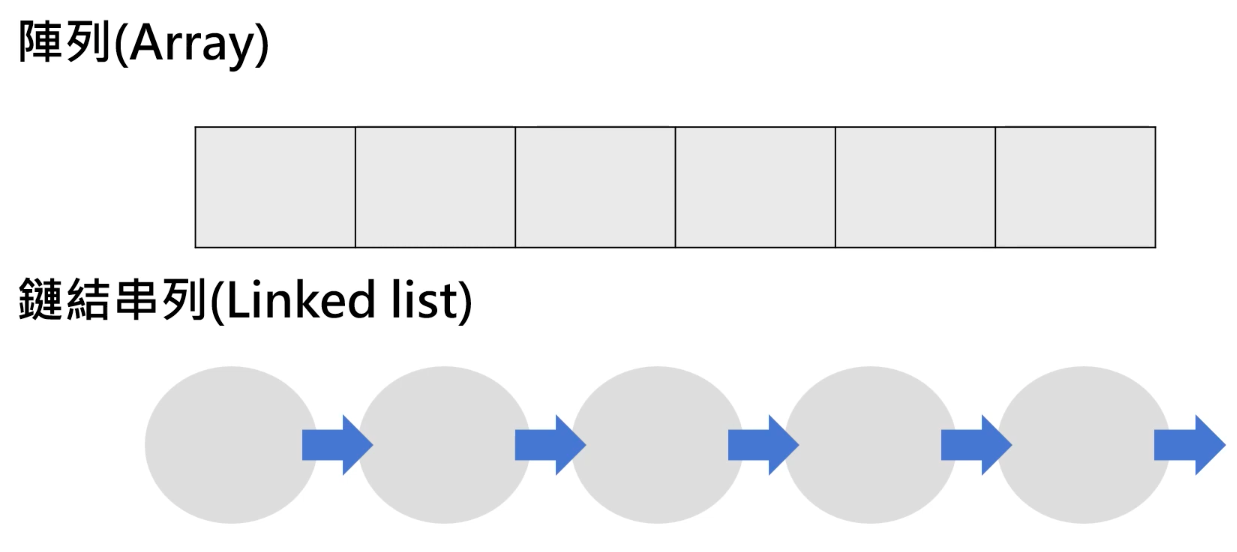
與此類似，新增資料和刪除資料都是同一側的話，我們就把它叫做堆疊。

另一個常見的例子是品客洋芋片，平常在吃的時候，都是從圓筒的上方把洋芋片拿出來，如果吃不完，需要放回去，也是從上方放回。

堆疊就是這樣一個「新增資料和刪除資料在同一側」的結構，這種資料結構會有什麼特性？想像它像洗衣服一樣，剛洗完的衣服放在最上面，每次要穿新衣服也從上面拿出來，因為放和拿都是從最上面，就導致「最上面的資料會被頻繁更新」，下面的資料則久久才更新一次。

插入和刪除在同側時，又叫做 last-in-first-out（LIFO），根據文意，也就是最後進去結構的資料最先出來，就如最後洗完的衣服放在最上面，下次會第一個穿到，它很適合拿來實現「回復到之前的狀態」。

再來是佇列，佇列的插入和刪除在不同側，這像是排隊的時候，先排進去隊伍中的，會最先排完隊而從隊伍出來：從右邊開始排隊，左邊進場時，先排到佇列裡面的就會先出來，又叫做 first-in-first-out（FIFO）。

(1) 堆疊與佇列的實作方式

實作堆疊和佇列的時候，都各有兩種方式：一個是用「陣列」來實作，另一個則使用「鏈結串列」，但因這兩種資料結構本身特性上的不同，所以本書在實作上，選擇用陣列練習 Stack 的實作、鏈結串列練習 Queue 的實作。

(2) 操作堆疊與佇列的複雜度

堆疊和佇列都只支援在特定的位置新增和刪除資料，查詢也是一樣。

新增、刪除、查詢這三種操作的複雜度都是 ，不管資料的筆數多寡，取出所需要的時間都是一個「定值」。

新增、刪除、查詢只能在特定位置進行的特性，實務上更像是一種「保護機制」，在取用時，只能照順序一筆筆取出來，不會也不能跳過某幾筆資料，這和「印出資料」或者「函式與函式間呼叫」需要「照順序處理」的特性呼應。

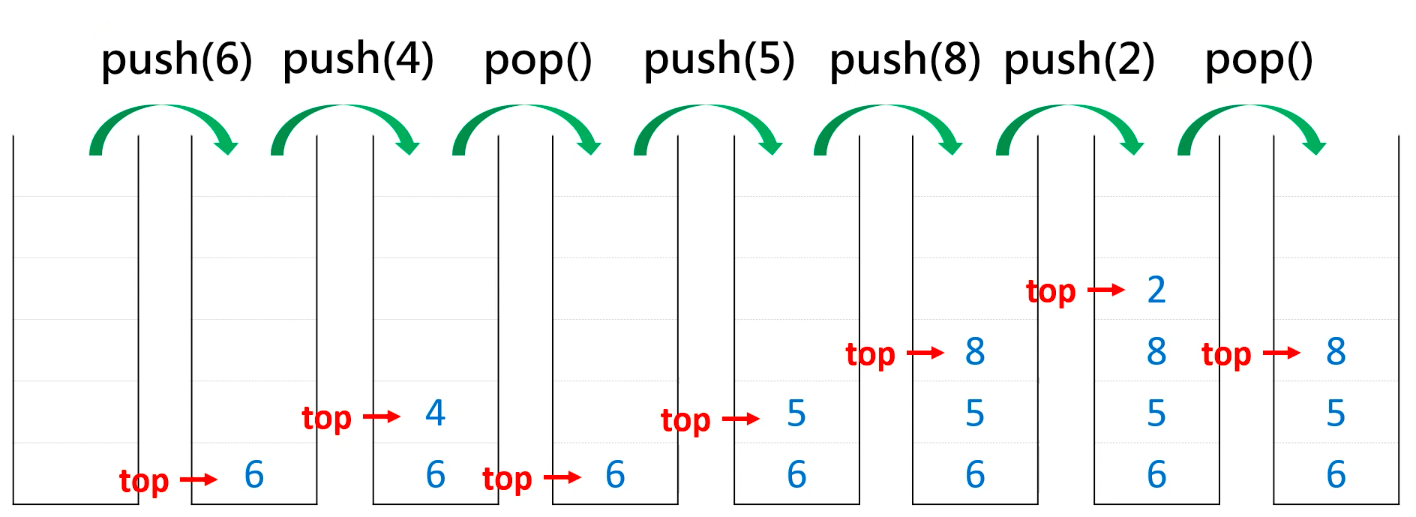
之後介紹應用時，就會看到它為什麼要做這樣的限制。

2. 堆疊 Stack 簡介

還有一種類比，是把堆疊想像成一個書堆，每次放新的一本書都是從上面放，每次拿書也從上面拿。

堆疊的操作因為受到限制，比較單純，沒有 erase、insert 等操作，基本的操作只有下列 5 種：

|  |  |
| --- | --- |
| 常見的堆疊操作 | |
| push  pop  top  empty  size | 新增一筆資料  刪除一筆資料  回傳最末端（上面）的資料  確認 stack 裡是否有資料  回傳 stack 內的資料個數 |

(1) 堆疊操作的實例

實際來看一個例子，表示堆疊時，通常會畫成上圖這樣像品客洋芋片的圓筒形狀，上面是開口，下面則是封閉的，每次新增資料都是從上面放進去，要取出資料時也必須從上面取出來。

假設一開始執行 push(6)，代表新增一筆資料 6 到這個堆疊裡面，堆疊中多出 6 這筆資料，此時會有一個 top 指標指向「最上面那筆資料」的位置。再來，執行 push(4)，把 4 這筆資料放進堆疊裡，這樣 4 就在 6 的上面。

pop() 是刪除一筆資料，要刪除時一樣從上面拿走，所以會把最上面的 4 拿掉，剩下 6。

接下來是 push(5)，把 5 放到堆疊裡，變成 5 和 6，push(8) 和 push(2) 依序把 8 和 2 放到堆疊裡，這時 pop() 把最上面的 2 取出來，堆疊從上而下剩下 8 5 6 這三筆資料。

(2) 練習判斷堆疊的新增與刪除

給定 stack = {1,2,3}，方向為右進右出，經過以下操作後，該 stack 的最後內容為何？

A. push(4)

B. pop()

C. push(5)

D. push(6)

E. push(7)

F. pop()

G. pop()

先試著動手算一次，稍後再來寫程式實作。

A. push(4)：{1,2,3,4}

B. pop()：{1,2,3}

C. push(5)：{1,2,3,5}

D. push(6)：{1,2,3,5,6}

E. push(7)：{1,2,3,5,6,7}

F. pop()：{1,2,3,5,6}

G. pop()：{1,2,3,5}

經過這些操作之後，最後 stack 裡的資料是 {1,2,3,5}。

3. 堆疊 Stack 的用途

(1) 堆疊最常見的用途：依序紀錄先前的資訊

A. 常用來回復到先前的狀態

a. 瀏覽器回到上一頁

b. 編輯器復原：在 word 中打錯字，想回復先前的狀態時按「復原」

B. 編譯器的解析 parse

C. 函式呼叫（遞迴）

D. 迷宮探索、河內塔、發牌：Depth-First Search（演算法內容）

E. 無法得知 stack 裡有哪些資料：只能以 pop () 一個個把資料拿出來

在文書軟體中依序做一些操作：剪下、貼上、斷行、輸入、調整字體、貼上、刪除，完成這些操作後，若想要復原應如何進行？要從最後進行的「刪除」開始，依序檢視做過哪些動作，並進行反向操作。

所以每次使用一個指令，或者瀏覽一個網頁的時候，都是把這些操作放到一個 堆疊 stack 裡面，最後放入的資料在最上面，與「復原」時最後做的操作應最先復原的特性相同，也就是 last-in-first-out，類似的，最後造訪的網頁要最先還原。

就像這樣，常見的「文件還原」和瀏覽網頁時使用的「回到上一頁」功能都是透過遞迴來實作的。

stack 和 queue 有一個共通問題：無法直接知道其中有哪些資料，只能一個一個依序把資料取出來，而不能跳過次序將中間的某筆資料取出，這種操作是不被保護機制允許的。

(2) Stackoverflow

你或許聽過「Stackoverflow」，這是一個著名「解答程式問題」網頁的名稱。



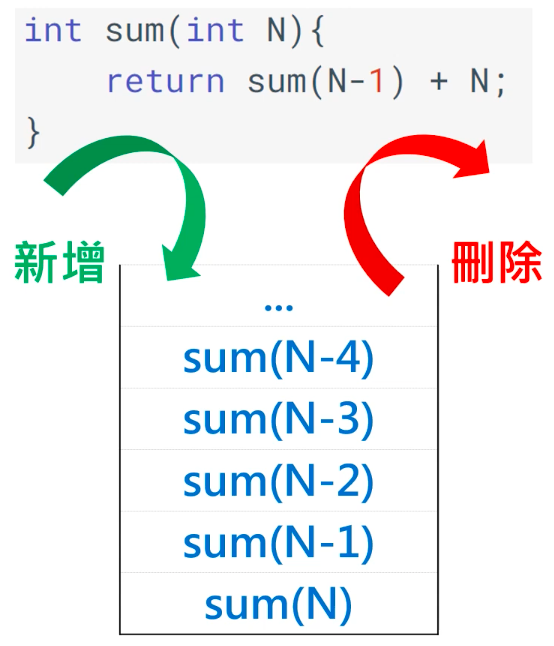
Stackoverflow 的名稱和 logo 是怎麼來的呢？這牽涉到記憶體配置的問題：遞迴呼叫一個函式的時候，每次呼叫都要紀錄當下的位置，以便之後可以返回。

|  |  |
| --- | --- |
| 遞迴呼叫的函式 | |
| 1  2  3 | int sum (int N){  return sum(N-1) + N;  } |

比如可以把「1 加到 N 的和」拆解成「1 加到 N-1」再「加 N」：

sum(N) = 1 + 2 + … + N = (1 + 2 + … + N-1) + (N) = sum(N-1) + N

所以 sum(N) 也可以表示成 sum(N-1) + N。



第一次呼叫執行 sum(N)，將 sum(N) 放到某個特定的 stack 裡面，接著因為 sum(N) 裡呼叫了 sum(N-1)，所以又把 sum(N-1) 也放到 stack 裡面，sum(N-1)

又會呼叫 sum(N-2)、sum(N-2) 會呼叫 sum(N-3)、...，依此類推，把所有被呼叫的函式不斷的放到 stack 裡。

在呼叫 sum(N-4) 之後，函式裡的「return」究竟該「回歸」到哪裡呢？

答案是 sum(N-3)，這個遞迴產生的 stack 會指出目前執行的函式是被誰呼叫的，就像 word 會把執行的每個步驟都用 stack 記錄方便還原一樣，只要依序把資料從 stack 裡面取出來就可以回到上一個步驟了。

呼叫函式要紀錄當下位置以便之後返回

A. 最後呼叫的函式會優先返回

B. 作業系統會以 stack 處理函式呼叫

呼叫太多層函式時，產生「Stack Overflow」錯誤。

stack 本身的容量是有上限的，如果遞迴的層數太多，把 stack 的整個容量都占滿了，程式就會崩潰，發生 Stack Overflow （堆疊溢位），這也是 stackoverflow 網站命名的由來。

**第二節：堆疊的實作**

接下來要示範如何實作一個堆疊 stack。

堆疊 Stack

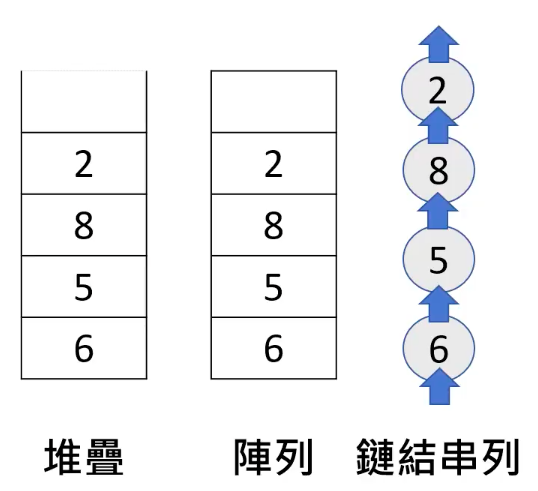
A. 可以用陣列或鏈結串列來實作

a. 以陣列示範堆疊 Stack

b. 再以鏈結串列練習佇列 Queue

B. 堆疊有大小限制

a. 解決方式：realloc

 b. 或是直接使用 vector

選擇以「陣列」實作堆疊，是因為堆疊「新增和刪除資料」在同側，從上面把資料取走後，馬上又可以從上面把資料放進去，與佇列相比，較不容易發生「放滿」的情形；佇列是從上面把資料放進去，從下面把資料拿出來（拿出資料空出的空間不能直接用來放新的資料），相對更容易遇到「放滿」的問題，在佇列的章節中會說明如何解決。

然而陣列仍然會有容量的限制，所以使用「動態陣列」來控制記憶體空間的大小，比如使用 realloc，或者 vector，因為目的是練習實作，所以下列示範將引入函式庫：include <stdlib>，想辦法以動態陣列實作出堆疊來。

1. 堆疊類別內的成員

(1) 資料成員

A. 空間大小 Capacity：容器（底層的動態陣列）能容納的上限

B. 最上層位置 Top\_Index：容器內最上層資料的位置，

代表目前使用到這個 Array 的哪個位置

C. 指標 Pointer：指向儲放資料的空間（開頭的記憶體位置）

(2) 函式成員

A. 新增 Push

B. 刪除 Pop

C. 大小 Size

D. 空 Empty

E. 取值 Top

(3) 以陣列實作堆疊的步驟：

A. 先用 malloc 準備好一個陣列

B. Top\_Index（即索引值）一開始沒有任何資料，初始化為 -1

C. 再分別完成下列函式

a. Top

b. Empty

c. Size

d. Double\_Capacity

e. Push

f. Pop

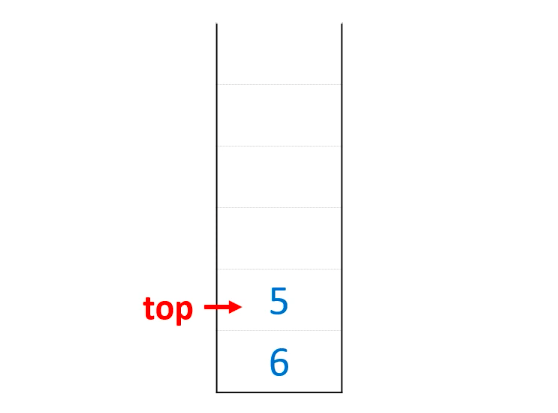
g. Print\_Stack

其中，Double\_Capacity 的功能是把目前的大小 Capacity 變成兩倍，用於在容量 Capacity 不夠時擴充空間；Print\_Stack是印出陣列中的所有資料，這個函式 STL 中沒有，僅方便測試使用。

為了表明我們的堆疊由陣列實作，將其命名為 Stack\_Array。

|  |  |
| --- | --- |
| class Stack\_Array | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | template<typename T>  class Stack\_Array{  private: // 屬性都設定成 private  int Capacity;  int Top\_Index;  T\* Pointer;  // 理論上只有類別內的函式會呼叫  void Double\_Capacity();  public: // 函式要被外界取用  Stack\_Array(int=0);  bool Empty;  int Size();  T Top();  void Push(T);  void Pop();  void Print\_Stack();  }; |

2. 函式的實作邏輯

(1) Top：回傳最末端的資料

A. 確認 Stack 不為空

* 如果 Top\_Index 值是 -1，就代表裡面沒有任何資料

B. 回傳 Top\_Index

* 指向目前 stack 中「最上面那筆資料」

(2) Empty：確認 Stack 裡是否有資料

A. 確認 Top\_Index 是否 >= 0

B. 如果 Top\_Index >= 0，回傳 false

C. 如果 Top\_Index 是 -1，回傳 true，代表堆疊是空的

(3) Size：回傳 Stack 內的資料個數

A. 回傳 Top\_Index + 1

說明：索引值是 3 代表裡面有 4 筆資料，索引值是 4 代表裡面有 5 筆

資料，索引值是 -1 則代表裡面沒有任何資料，Top\_Index + 1 正好就等於

目前 stack 裡面的資料筆數。

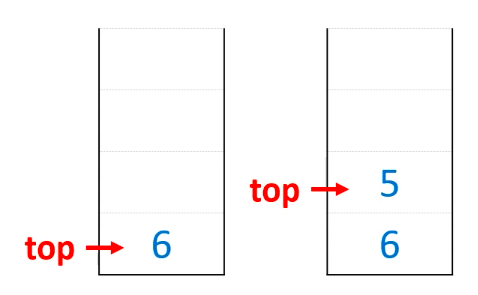
(4) Push：新增一筆資料

A. 確認空間足夠，空間不夠時用 realloc 重新配置空間

* Capacity = Capacity\*2

B. Top\_Index 往後移一格（Top\_Index++），等於多給出一個空間

C. 把資料 assign 給 Top\_Index 指到的位置



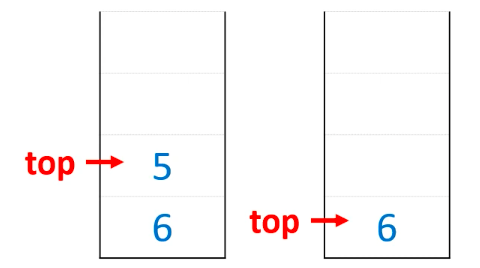
說明：想要在 6 這筆資料上面新增一筆 5，就把 top 往上移一格，再把

5 這個值賦予給 top 新指到的位置。

(5) Pop：刪除一筆資料

A. 確認 Stack 不為空

B. Top\_Index 往前移一格（Top\_Index--）



說明：因為 top 代表目前 stack 裡面最上面那筆資料的位置，取用時不可

以超過這個位置，所以資料「不需要」真的刪除，在上面的例子裡，之後

讀取只會讀到 6，再新增一筆資料時自動就會把已經刪除的 5 覆蓋掉。

3. 初始化一個堆疊，並完成

A. 建構式

B. Print\_Stack()

|  |  |
| --- | --- |
| Stack.h | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #ifndef STACK\_H\_INCLUDED  #define STACK\_H\_INCLUDED  #include <stdlib.h> // for malloc  #include <iostream> // for cout  using namespace std;  template<typename T>  class Stack\_Array{  private: // 屬性都設定成 private  int Capacity;  int Top\_Index;  T\* Pointer;  // 理論上只有類別內的函式會呼叫  void Double\_Capacity();  public: // 函式要被外界取用  Stack\_Array(int=0);  bool Empty;  int Size();  T Top();  void Push(T);  void Pop();  void Print\_Stack();  };  …  #endif // STACK\_H\_INCLUDED |

|  |  |
| --- | --- |
| 建構式 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template<typename T>  Stack\_Array<T>::Stack\_Array(int len){  // 容量設定為與 len 相同  Capacity = len;  // 如果使用者有給出非 0 的初始化大小  if (len>0){  // 用 malloc 開出空間，Pointer 指向這個動態陣列的開頭  Pointer = (T\*) malloc(sizeof(T)\*len);  }  // 沒有給出引數時，把 Pointer 設定為空指標  else {  Pointer = nullptr;  }  // 目前還沒有任何資料，因此 Top\_Index 要設定成 -1  Top\_Index = -1;  } |

|  |  |
| --- | --- |
| 印出堆疊中所有資料：Print\_Stack | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | template<typename T>  void Stack\_Array<T>::Print\_Stack(){  cout << "Data:";  for(int i=0;i<=Top\_Index;i++){  cout << \*(Pointer+i) << " ";  }  cout << endl;  } |

|  |  |
| --- | --- |
| 測試堆疊的建構式與 Print\_Stack | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <iostream>  #include "Stack.h"  using namespace std;  int main()  {  // 注意類別名稱取名叫「Stack\_Array」以和 STL 中的 Array 區別  Stack\_Array<int> data;  data.Print\_Stack();  return 0;  } |
| 執行結果（目前堆疊中還沒有資料） | |
| Data： | |

4. 在 Stack.h中完成以下函式

A. Empty()

B. Size()

C. Double\_Capacity()

D. Top()

E. Push()

F. Pop()

|  |  |
| --- | --- |
| 判斷是否為空：Empty | |
| 1  2  3  4 | template<typename T>  bool Stack\_Array<T>::Empty(){  return (Top\_Index == -1);  } |

|  |  |
| --- | --- |
| 回傳堆疊大小：Size | |
| 1  2  3  4 | template<typename T>  int Stack\_Array<T>::Size(){  return Top\_Index + 1;  } |

|  |  |
| --- | --- |
| 將容量 Capacity 變為兩倍：Double\_Capacity | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | template<typename T>  void Stack\_Array<T>::Double\_Capacity(){  // 例外處理：原本為空堆疊（Capacity == 0）  if(Capacity == 0){  Capacity = 1;  Pointer = (T\*) malloc(sizeof(T));  }  // 一般情形  else{  Capacity \*= 2;  // 先儲存 Pointer 位置以便釋放  T\* tmp = Pointer;  // 開一個新的兩倍大空間給 Pointer  Pointer = (T\*) malloc(sizeof(T)\*Capacity);  // 把原先的資料搬遷到新空間  for(int i=0;i<=Top\_Index;i++){  Pointer[i] = tmp[i];  }  // 釋放原空間  free(tmp);  }  } |

|  |  |
| --- | --- |
| 回傳最上面一筆資料：Top | |
| 1  2  3  4  5  6  7  8  9  10  11 | template<typename T>  T Stack\_Array<T>::Top(){  // 例外處理：空 Stack  if (Empty()){  cout << "Error! This stack is empty!" << endl;  }  // 一般情形  else {  return Pointer[Top\_Index];  }  } |

|  |  |
| --- | --- |
| 新增資料進堆疊：Push | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | template<typename T>  void Stack\_Array<T>::Push(T value){  // 空間已滿時擴充  if (Top\_Index == Capacity-1){  Double\_Capacity();  }  // 移動最上面一筆資料的 index，使資料要放入的位置變成可以取用  Top\_Index++;  // 放入資料  Pointer[Top\_Index] = value;  // 上面兩行也可合併成  // Pointer[++Top\_Index] = value;  // 會先把 Top\_Index +1 後再傳回 Top\_Index 的值  } |

|  |  |
| --- | --- |
| 從堆疊中去除一筆資料：Pop | |
| 1  2  3  4  5  6  7  8  9  10  11 | template<typename T>  void Stack\_Array<T>::Pop(){  // 例外處理：空 Stack 時不處理  if(Empty())  return ;  // 一般情形：Top\_Index 減一，使最後一筆資料不會被取用到  Top\_Index--;  } |

|  |  |
| --- | --- |
| 測試堆疊 Stack 的各項功能 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <iostream>  #include "Stack.h"  using namespace std;  int main()  {  Stack\_Array<int> data;    // 新增資料到 stack 中  for(int i=0;i<5;i++){  data.Push(i);  data.Print\_Stack();  }  // 印出 stack 中的資料  for (int i=0;i<5;i++){  cout << data.Top() << " " << endl;;  // 刪除 stack 最上端的資料  data.Print\_Stack();  data.Pop();  }  return 0;  } |
| 執行結果 | |
| Data:0  Data:0 1  Data:0 1 2  Data:0 1 2 3  Data:0 1 2 3 4  4  Data:0 1 2 3 4  3  Data:0 1 2 3  2  Data:0 1 2  1  Data:0 1  0  Data:0 | |

**第三節：C++ STL 裡面的堆疊**

接下來，來看 C++ STL 裡面的堆疊 stack。

A. C++ STL 裡面常見的容器：

a. Container adapter：提供特殊的介面/資料存取順序

* stack：新增刪除資料在同側
* queue：新增刪除資料在異側
* priority\_queue

B. C++ 中：

a. stack、queue、priority\_queue 各有特性與適用情形

b. deques 是 double-ends queue

* 插入、搜尋、刪除：
* b. 只能在特定位置進行

C. Python：沒有引用特定函式庫的情況下，一般都使用 list

這也是學資料結構時常使用 C++ 的原因，因為 C++ STL 有把這些不同的資料結構各自封裝起來，且可以直接使用。

1. STL 中的 stack

(1) stack 和 queue 的使用

A. 引用函式庫

#include <stack>

#include <queue>

B. 宣告時，以要使用的資料結構名稱開頭，<> 中放的是裡面資料的型

別，如果堆疊中要放 int，就寫 <int>，並在後方取一個名稱

stack<datatype> stack\_name;

queue<datatype> queue\_name;

注意 STL 中的 stack 與 queue 沒有迭代器 iterator！因為這兩種結構都只能從固定的方向一筆一筆取用資料，所以迭代器沒有意義。

|  |  |
| --- | --- |
| STL 中 stack 的操作 | |
| stack.push(value)  stack.pop();  stack.top();  stack.empty();  stack.size(); | 新增一筆資料  刪除一筆資料  回傳一筆資料  判斷 stack 是否為空  回傳 stack 的長度 |

|  |  |
| --- | --- |
| 測試 STL 中的 Stack | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <iostream>  #include <stack>  using namespace std;  int main(){  // data 中放的是 int 資料  stack<int> data;  // 新增資料  for (int i=0;i<10;i++) // 0 1 2 3 4 5 6 7 8 9  data.push(i);  cout << data.top() << endl; // 9  data.pop(); // 0 1 2 3 4 5 6 7 ~~9~~  data.pop(); // 0 1 2 3 4 5 6 7 ~~8~~  cout << data.top << endl; // 7  return 0;  } |
| 執行結果 | |
| 9  7 | |

2. print\_stack

Stack 的資料只能照順序取用，不能跳過部分資料直接取用中間的資料，唯一使用中間資料的方法就是把在該筆資料上方的資料全部先 pop 掉。

|  |  |
| --- | --- |
| 印出 stack 內的資料： | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | void print\_stack(stack<int>& s){  // 例外處理：空 stack  if(s.empty())  return ;    // 一般情形  // 把最上面的資料存進 data 裡  int data = s.top();  // 刪去最上面的資料後重新丟回函式中  // 相當於處理去除最上面一筆資料的 stack  s.pop();  print\_stack(s);  // 從最底層的資料開始處理  // 只剩下一筆資料的時候會第一次運行到這裡  cout << data << " ";  // 為了不改動 stack，把資料放回 stack 中  s.push(data);  } |

3. Example：LeetCode#155 最小值堆疊 Min Stack

A. 題目

設計一個堆疊，支援下列操作，且可以在 時間內回傳最小元素

a. push(x)：加入一筆資料 x 到堆疊中

b. pop()：把堆疊最上方的元素移除

c. top()：取得堆疊最上方的元素

d. getMin()：取得堆疊中值最小的元素

B. 出處：https://leetcode.com/problems/min-stack/

C. 說明

min stack 是要實作出一個最小堆疊，這個最小堆疊可以在 的時間（固定時間）內取出 stack 中的最小資料。

D. 解題邏輯

如果一個個去看哪筆資料最小，會需要 ，所以我們需要使用兩個 stack

A. 第一個 stack 叫做 data，記錄了原本的資料

B. 第二個 stack 叫做 min，它紀錄了到目前為止 stack 裡面的最小值

用下面的例子說明，從最底端的 6 開始新增

|  |  |
| --- | --- |
|  | // data 中不高於此的資料，最小值是 1  // data 中不高於此的資料，最小值是 4  // data 中不高於此的資料，最小值是 4  // data 中不高於此的資料，最小值是 4  // data 中不高於此的資料，最小值是 6  // data 中不高於此的資料，最小值是 6 |

min 中記載了對應到 data 堆疊中，目前高度之下出現過的最小值，要取出資料時，從 data 堆疊取用，只有在取出最小值時，改取用 min。

|  |  |
| --- | --- |
| 最小值堆疊 Min Stack | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | class MinStack{  stack<int> data;  stack<int> min;  public:  // 建構式，不需實作  MinStack(){};    // 新增資料到兩個 stack 中  void push(int val){  // data 這個 stack 中，val 按照一般規則直接新增在最上端  data.push(val);  // min 的例外處理：目前沒有資料時，min 中要放 val  if (min.empty()){  min.push(val);  return ;  }  // min 的一般情形  // 取出新增目前 val 之前下面有最小值的資料  int current\_min = min.top();  // 如果新的資料 val 比之前的最小值還小，才改放 val  if (val < current\_min){  min.push(val);  // 否則繼續放 current\_min  } else {  min.push(current\_min);  }  }  // 刪除最上端資料  void pop(){  // 例外處理：空 stack  if(data.empty())  return ;  // 一般情形：兩個 stack 都去掉最上端資料  data.pop();  min.pop();  }    // 回傳最上端的資料  int top(){  return data.top();  }  // 回傳目前資料中最小值  int getMin(){  // 從 min 中取最上端的值  return min.top();  }  } |

4. LeetCode#20. 判斷括號合法性 Valid Parentheses

A. 題目

給定一個字串 s，其中只含有以下六種字元：「(、)、{、}、[、]」，根據一定的法則，判斷該字串是否合法。

一個字串合法的條件為：

a. 左括號必須被對應的右括號關閉

b. 右括號的順序要正確對應左括號出現的順序

B. 出處：https://leetcode.com/problems/min-stack/

C. 說明

利用堆疊 Stack 撰寫除錯工具，除錯工具可以確認某字串中的括號是否有成對，例如：

A. -> True

B. -> 小括號不成對，False

C. -> 中括號不成對，False

撰寫程式時，編譯器都會幫我們檢查括號是否有成對，這個應用很常見。

D. 解題邏輯

先準備一個 stack，裡面的資料代表上一個遇到的左括號類型，從左邊開始依序取出字元，如果是左括號 {、[、( ，把它放到 stack 裡面，如果是右括號 )、]、}，則檢查是否和 stack 最上方的左括號正確對應。

中，左括號依序是 {、[、(，因此 stack 從下到上也依序是 {、[、(。

往右處理的過程中，數字與加減乘除符號可以忽略，遇到第一個右括號的時候（上例中是右小括號），把之前最後遇到的左括號拿出來看，因為正好是左小括號，代表這組兩個括號有成對，接下來，把這個左小括號拿掉，繼續向右比對，發現下一個右括號是右中括號時，檢查 stack 此刻最上面的資料是不是對應的左中括號。

依此繼續進行，若 stack 可以剛好被完全消完，就代表字串中的括號有成對。

|  |  |
| --- | --- |
| 判斷括號合法性 Valid Parentheses | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | class Solution{  // 建立可以存放字元的堆疊 data  stack<char> data;  public:  bool isValid(string s){  // 把字串 s 中的字元一個個取出來放到 c 裡面  for (char c:s){  // 遇到左括號時，放到陣列 data 當中  if (c=='(' || c =='[' || c='{'){  // 字元 c 加到陣列當中  data.push(c);  // 繼續執行下一輪迴圈，處理下個字元  continue;  }  // 例外處理：  // 出現右括號，但堆疊中還沒有加入過小括號  // 或者堆疊中的左括號已經全部被消掉了  if ((c==')' || c==']' || c=='}')&& data.empty())  // 沒有成對  return false;  // 一般情形  // 遇到右小括號  if (c==')') {  // stack 頂端要是左小括號才有成對  if (data.top=='('){  data.pop();  }  // 否則一定不成對  else {  return false;  }  }  // 遇到右中括號  else if (c==']') {  // stack 頂端要是左中括號才有成對  if (data.top=='['){  data.pop();  }  // 否則一定不成對  else{  return false;  }  }  // 遇到右大括號  else if (c=='}') {  // stack 頂端要是左大括號才有成對  if (data.top=='{'){  data.pop();  }  // 否則一定不成對  else{  return false;  }  }  }  // 右括號使用完後，檢查結果  // stack 中的左括號都被使用完，代表成對  if (data.empty())  return true;  // 左括號沒有全部消完，代表沒有成對  else  return false;  }  } |

5. Leetcode#143. 改變鏈結串列順序 Reorder List

A. 題目

給定一個單向鏈結串列，可以表示為：

把該串列的順序改為：

本題規定不能透過修改節點內的值完成，只能修改節點間指標的指向。

B. 出處：https://leetcode.com/problems/reorder-list/

C. 說明

這題把 stack 跟之前的 linked list 做結合，題目的要求是把給定的順序重新做排序，如果是 1 2 3 4，要變成 1 4 2 3，如果是 1 2 3 4 5，要變成 1 5 2 4 3。

D. 解題邏輯

準備一個 stack，依序把想要的順序放到 stack 當中，思考一下如何放進 stack 中才會符合題目要求的順序。

因為其中 、、、... 這部分與原本的順序是倒過來的，所以要有一種方法來「逆著順序取用」。剛好堆疊就有這個特性，如果放資料進去的順序是 1 2 3 4 5，取出的順序會是 5 4 3 2 1（右進右出），可以把資料逆向取用。

|  |  |
| --- | --- |
| 改變鏈結串列順序 Reorder List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | class Solution {  // 利用一個堆疊 data 來儲存所有節點的記憶體資料  stack<ListNode\*> data;  public:  void reorderList(ListNode\* head){  // current 從 list 的開頭出發  ListNode\* current = head;  // 用整數 len 來計算list 資料的總筆數  int len = 0;  // 把 list 中的資料依序放到堆疊中  // 注意取用時會是反向的（從尾項到頭項）  while(current!=nullptr){  // 在 data 中儲存當前 current 節點的記憶體位置  data.push(current);  current = current->next;  len++;  }  // 重新從開頭處理，開始改變節點順序  current = head;  // 要把後半的節點插入前半，  // 迴圈只需要處理到長度 len 的一半即可  for (int i=0 ; i<len/2 ; i++){  // 從剛才的堆疊中取出一筆資料  // 注意取到的順序會是從尾端節點開始  ListNode\* last\_node = data.top();  data.pop();  // 要插入資料到兩筆資料之間  // 比如在 A->B 中插入一筆資料 C，變成 A->C->B  // 步驟如下（A: current, B: current->next, C: last\_node）：  // 把 C 的 next 指向 A 的 next（也就是B）  last\_node->next = current->next;  // 把 A 的 next 改指向 C  current->next = last\_node;  // current 往後一格從 A 到 B  // 因為中間已經插入了 C，要移動到 current->next->next  current = current->next->next;  }  // current 處理完成後，尾端資料的 next 指向空指標  current->next = nullptr;  }  }; |

觀察一下上面的程式碼，如果輸入是一個空的 list 時也可以處理：因為一開始 current 會被設定成 head，在空的 list 中也就是空指標，因此 len 會被設定為 0，第二個迴圈不會被執行，最後處理的結果仍然是一個空的 list，所以上面的程式碼也能夠正確處理這個邊界狀況。

**第四節：堆疊常見的應用**

接下來，我們會來介紹三個堆疊常見的應用

A. 河內塔：一個經典的古老遊戲

B. 深度優先搜尋：老鼠走迷宮

C. 中序轉後序運算

1. 河內塔

A. 河內塔遊戲的規則可以表示為：

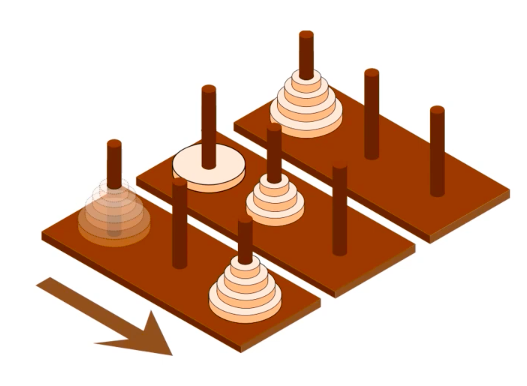
a. 讓使用者輸入一正整數 N，輸出將 N 層河內塔從一根棍子移到另一根

棍子的所有過程。

b. 共有三根棍子，每根棍子上可以擺放盤子。開始至完成間，每次只能移

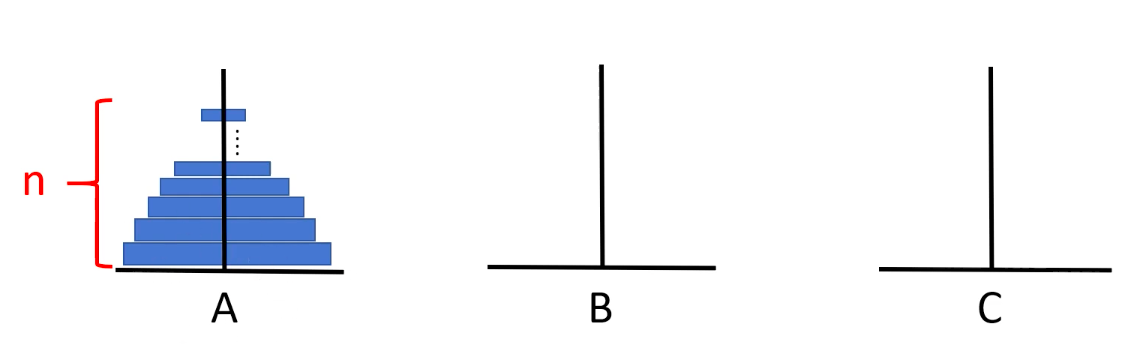
動一個盤子，且移動前後，所有棍子上，較上方的盤子都必須較下方的

「所有盤子」來得小，也就是說，對於每根棍子而言，只要上面有盤子，

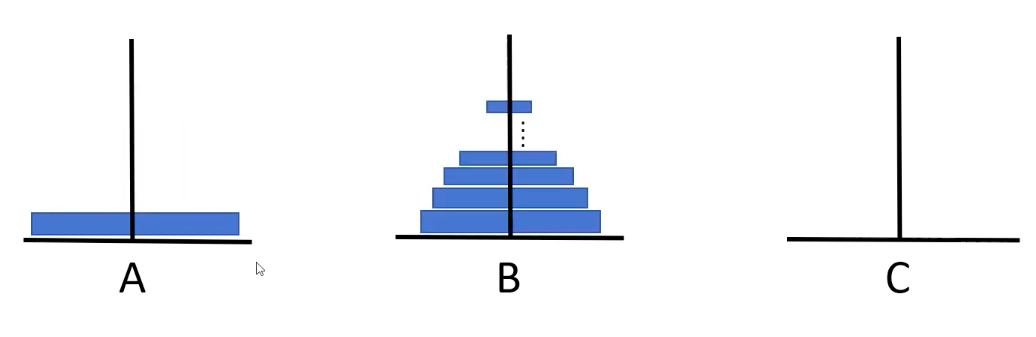
盤子就必須是從底層開始由大到小依序擺放。

c. 提示：因為大的盤子一定要放在小盤子的下面，所以最下面 / 最大的盤

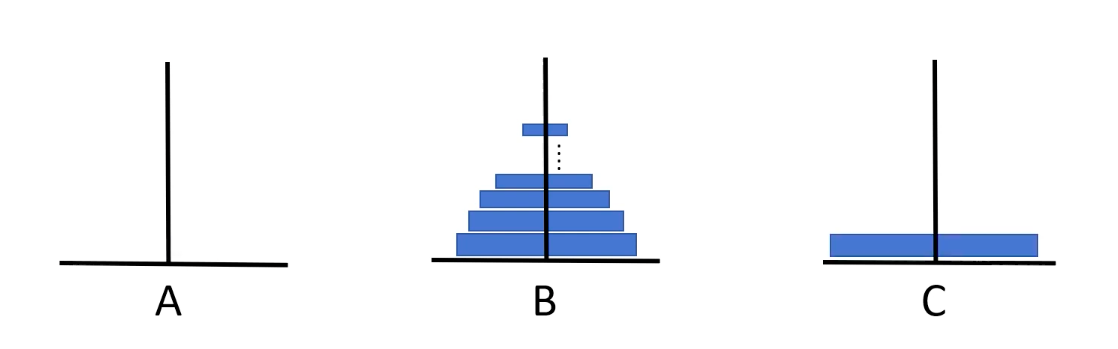
子必須先移動到目標的棍子上。

B. 解題邏輯

目標是要把所有盤子從 A 移到 C，那麼「最大的盤子」必定要最先從 A 移到 C。然而如果想要移動這個目前位於 A 底部的最大的盤子，在它上方的 n-1 個盤子都必須先移動到其他兩根棍子之一上。



假設找到一個方法，把上面的 n-1 個盤子都移動到 B 了，那麼緊接著，就可以用「一次」移動把最大的盤子從 A 棍子移動到 C 棍子上。



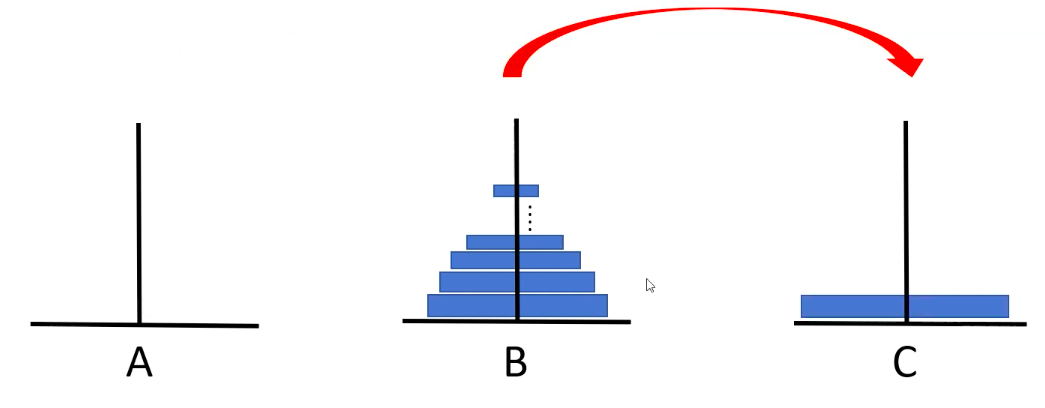
經過上面的處理，目前的情況是：

A：沒有盤子

B：除了最大盤子外的 n-1 個盤子

C：最大的盤子

從目前的情況開始，繼續把 B 上的 n-1 個盤子移動到 C 棍子，就可以達成遊戲的目標。



然而這同樣需要經過「把 n-1 個盤子移動到另一個棍子」的步驟，因此我們可以列出下面的等式：

// 把 n 個盤子從 A 移到 C =

// 把 n-1 個盤子從 A 移到 B +

// 把最大的盤子從 A 移到 C

// 把 n-1 個盤子從 B 移到 C

這也就是說：n 層河內塔的問題可以被拆分表示為「n-1 層河內塔的問題」與「1 層河內塔的問題」與另一個「n-1 層河內塔的問題」。

河內塔的問題與堆疊有什麼關係呢？當要從棍子上把盤子取走的時候，是從上方取走盤子，要把盤子放下時，也是從棍子上方放下，因為「新增盤子」與「刪除盤子」是同向，所以三個棍子就等同於三個堆疊，在進行河內塔遊戲的過程，也是對三個 stack 進行操作。

C. 目標輸出

A B C 代表三根棍子，每次移動後，分別輸出 A B C 上的每個盤子。

一開始 A 從下而上有三個盤子 3、2、1 時的目標輸出：

A:3 2

B:

C:1

--------------

A:3

B:2

C:1

--------------

A:3

B:2 1

C:

--------------

A:

B:2 1

C:3

--------------

A:1

B:2

C:3

--------------

A:1

B:

C:3 2

--------------

A:

B:

C:3 2 1

--------------

|  |  |
| --- | --- |
| 河內塔 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | #include <iostream>  #include <stack>  using namespace std;  // 宣告三個 stack 為全域變數，可使用的空間會較大  stack<int> A;  stack<int> B;  stack<int> C;  // 仿照上面的邏輯，寫出搬動河內塔的函式  // 把代表三根棍子的三個堆疊分為 from、to 和 others  // 每次呼叫函式，藉由傳入時的順序表示要從哪個堆疊移動到哪個堆疊  void Hanoi(int N, stack<int>& from, stack<int>& to, stack<int>& others){  // 邊界條件：只有一個盤子（最大的盤子）要搬動  if (N==1){  // 把這個盤子由 from 搬到 to  to.push(from.top());  from.pop();  // 每次移動完都印出目前狀態  print\_all();  return;  }  // 一般情形  else {  // 把 N-1 個盤子從 from 移動到 others  // 在剛才的說明裡，為 A->B  Hanoi(N-1, from, others, to);  // 把 1 個盤子從 from 移動到 to  // 剛才的說明裡，為 A->C  to.push(from.top());  from.pop();  print\_all(); // 移動完要印出目前狀態  // 把 N-1 個盤子從 others 移動到 to  // 剛才的說明裡，為 B->C  Hanoi(N-1, others, to, from);    }  }  // 寫兩個印出資料的函式  // print\_stack 印出任一個堆疊中的資料  // print\_all 利用 print\_stack 來印出 A B C 三個堆疊全部的資料  // 與先前介紹的 print\_stack 做法相同  void print\_stack(stack<int>& s){  if (s.empty())  return ;  int data = s.top();  s.pop();  print\_stack(s);  cout << data << " ";  s.push(data);  }  void print\_all(){  cout << "\nA:";  print\_stack(A);  cout << "\nB:";  print\_stack(B);  cout << "\nC:";  print\_stack(C);  cout << "\n--------------";  }  int main(){  // 輸入 N，代表要移動 N 層河內塔  int N;  cout << "Please enter N:" << endl;  cin >> N;  // 設定初始狀態  // 堆疊 A 從底層開始，有編號 N 到 1 的盤子的 N 個盤子  for (int i=N;i>0;i--)  A.push(i);  // 第一次呼叫 Hanoi 函式，其中 from 是 A，to 是 C  Hanoi(N, A, C, B);  return 0;  } |
| 執行結果 | |
| Please enter N:  >> 3  A:3 2  B:  C:1  --------------  A:3  B:2  C:1  --------------  ... | |

2. 老鼠走迷宮

A. 題目

給定一個二維陣列，二維陣列中 1 代表牆壁，0 代表可以走的路徑，起點是 (1,1)，終點是 (8,10)，請找一條可以從起點通到終點的路徑，並把該可行路徑在陣列中用 2 這個值標出。

B. 初始狀態與目標輸出

int MAZE[10][12] = { // 初始狀態

{1,1,1,1,1,1,1,1,1,1,1,1},

{1,0,0,0,1,1,1,1,1,1,1,1},

{1,1,1,0,1,1,0,0,0,0,1,1},

{1,1,1,0,1,1,0,1,1,0,1,1},

{1,1,1,0,0,0,0,1,1,0,1,1},

{1,1,1,0,1,1,0,1,1,0,1,1},

{1,1,1,0,1,1,0,1,1,0,1,1},

{1,1,1,1,1,1,0,1,1,0,1,1},

{1,1,0,0,0,0,0,0,1,0,0,1},

{1,1,1,1,1,1,1,1,1,1,1,1},

};

int MAZE[10][12] = { // 目標輸出

{1,1,1,1,1,1,1,1,1,1,1,1},

{1,2,2,2,1,1,1,1,1,1,1,1},

{1,1,1,2,1,1,2,2,2,2,1,1},

{1,1,1,2,1,1,2,1,1,2,1,1},

{1,1,1,2,2,2,2,1,1,2,1,1},

{1,1,1,0,1,1,2,1,1,2,1,1},

{1,1,1,0,1,1,2,1,1,2,1,1},

{1,1,1,1,1,1,0,1,1,2,1,1},

{1,1,0,0,0,0,0,0,1,2,2,1},

{1,1,1,1,1,1,1,1,1,1,1,1},

};

C. 解題邏輯

利用一個堆疊 stack 儲存目前走過的路徑，並開另外一個二維陣列紀錄已經嘗試過的路徑（以使之後走時不要重複），不斷往下搜索直到無路可走的時候（代表目前路徑的尾端若干個不在正確路徑上），回頭退到分岔口找別的路徑。

堆疊的重要功能之一就是「復原狀態」，當發現目前走的路徑不是正確的，可以從堆疊取出資料以找到退回上一個路口（有其他方向可以選擇）的路徑。

程式的結束條件是已經走到終點，或者 stack 是空的，該題沒有可行的路徑。

|  |  |
| --- | --- |
| 老鼠走迷宮 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | #include <iostream>  #include <stack>  #include <vector>  using namespace std;  int main(){  int MAZE[10][12] = {  {1,1,1,1,1,1,1,1,1,1,1,1},  {1,0,0,0,1,1,1,1,1,1,1,1},  {1,1,1,0,1,1,0,0,0,0,1,1},  {1,1,1,0,1,1,0,1,1,0,1,1},  {1,1,1,0,0,0,0,1,1,0,1,1},  {1,1,1,0,1,1,0,1,1,0,1,1},  {1,1,1,0,1,1,0,1,1,0,1,1},  {1,1,1,1,1,1,0,1,1,0,1,1},  {1,1,0,0,0,0,0,0,1,0,0,1},  {1,1,1,1,1,1,1,1,1,1,1,1},  };  // 需要二維陣列（這裡使用二維 vector）來記錄哪些點走過  // 哪些點還沒有走過，以避免陷入無窮迴圈  // 10 個 row，12 個 column，初始值都是 false 的布林陣列  vector<vector<bool>> visited(10, vector<bool>(12,false));  // 記錄走過的路徑，每個點用 x 和 y 座標組成的 pair 來表示  stack<pair<int, int>> path;  // 起點 (1,1) 放到堆疊中，make\_pair 函式可以用來創造出 pair  path.push(make\_pair(1,1));  // 記錄起點 (1,1) 這個點已經走過  visited[1][1] = true;  // 當 path 中還有可行路徑的時候繼續進行  while(!path.empty()){    // 取出當前座標 (x,y)  pair<int, int>current = path.top();  int x = current.first;  int y = current.second;  // 檢查是否已經到達終點，本題中為 (8,10)  if(x == 8 && y == 10)  break;  // 找尋可以走的路徑  // 試試看能不能往「下方」走一格  // 兩個條件：一是下面一格在迷宮裡是可以走的數字 0  // 二是那一格不是已經走過的路徑  //（因為我們希望產生的路徑不會有回頭走的情形）  if (MAZE[x+1][y]==0 && visited[x+1][y] == false){  // 往下走一格並記錄到 path 當中  path.push(make\_pair(x+1),y);  // 把 visited 設定成 true，避免重複走到  visited[x+1][y] = true;  }  // 如果「下方」不能走，試試往「上方」走  else if (MAZE[x+1][y]==0 && visited[x-1][y] == false){  path.push(make\_pair(x-1),y);  visited[x-1][y] = true;  }  // 如果前幾個方向不能走，試試往「右方」走  else if (MAZE[x][y+1]==0 && visited[x][y+1] == false){  path.push(make\_pair(x),y+1);  visited[x][y+1] = true;  }  // 如果前幾個方向不能走，試試往「左方」走  else if (MAZE[x][y-1]==0 && visited[x][y-1] == false){  path.push(make\_pair(x),y+1);  visited[x][y-1] = true;  }  // 發現上下左右都不是可行的方向  // 代表目前的這點不在正確的路徑上  else{  // 把目前這一點從我們想要找到的正確路徑上去除  path.pop();  }  }  // 根據得到的正確 path，把 MAZE 當中路徑文字改成 2  while (!path.empty()){  pair<int, int> current = path.top();  int x = current.first;  int y = current.second;  MAZE[x][y] = 2;  // 繼續修改堆疊的下一筆資料  path.pop();  }  // 印出已經標出路徑的 MAZE 陣列  for(int i=0;i<10;i++){  for(int j=-;j<12;j++){  cout << MAZE[i][j] << " ";  }  cout << endl;  }  return 0;  } |
| 執行結果 | |
| 1 1 1 1 1 1 1 1 1 1 1 1  1 2 2 2 1 1 1 1 1 1 1 1  1 1 1 2 1 1 2 2 2 2 1 1  1 1 1 2 1 1 2 1 1 2 1 1  1 1 1 2 2 2 2 1 1 2 1 1  1 1 1 0 1 1 2 1 1 2 1 1  1 1 1 0 1 1 2 1 1 2 1 1  1 1 1 1 1 1 0 1 1 2 1 1  1 1 0 0 0 0 0 0 1 2 2 1  1 1 1 1 1 1 1 1 1 1 1 1 | |

這題的情況比較單純，因為迷宮邊緣正好都是一，不會有路徑超出邊緣的情況，一般來說，還需要處理邊界條件，如往右走一步後「y 座標加一」會不會導致超出邊界等。

3. 中序轉後序運算

(1) 中序與後序

A. 中序（Infix）表示運算子在需要進行運算的兩個數字的中間（比如 x 乘

號在 b 和 c 的中間，代表 b 和 c 要相乘），是我們慣用的、數學上的運

算順序：

B. 後序（Postfix）表示運算子在後面，是電腦慣用的運算順序：

(2) 後序運算的進行方式

電腦之所以使用後序 postfix 這種表達方式，是有運算上的考量。

後序運算可以無須考慮括號造成的優先次序，每當遇到「兩個數字」加上「一個運算子」時，就直接進行運算。

運算時，自左到右把資料取出，遇到「運算元（數字）」時放入堆疊，遇到「運算子」時，則從堆疊中拿出「最近放入」的兩個運算元，並以該運算子加以運算。

(3) 後序運算的例子：

(3)：把 3 放入堆疊 // 堆疊 = {3}

(5)：把 5 放入堆疊 // 堆疊 = {3,5}

(1)：把 2 放入堆疊 // 堆疊 = {3,5,2}

(x)：自堆疊中取出兩筆資料 5,2，運算後 5x2 放回堆疊 // 堆疊 = {3,10}

(+)：自堆疊中取出兩筆資料 10,3，運算後 10+3 放回堆疊 // 堆疊 = {13}

(6)：把 6 放入堆疊 // 堆疊 = {13,6}

(+)：自堆疊中取出兩筆資料 13,6，運算後 13+6 放回堆疊 // 堆疊 = {19}

( )：結束，運算結果即堆疊中的數字 19

(4) 中序轉後序的實作

A. 題目

給定一個中序運算的字串，請利用堆疊 Stack 把中序運算轉換成後序運算。其中運算元（數字）都只有一位數，且運算子間都以小括號包起來。

注意！本題只是產生後序運算的「運算式」，不需算出後序運算的算式結果。

B. 解題邏輯

開一個堆疊 stack 紀錄運算子（+、-、x、/），當遇到數字時直接輸出，遇到運算子則存入堆疊，遇到右括號時，取出一個運算子後輸出。

C. 目標輸出：

中序運算式：

輸出後序運算式：

|  |  |  |
| --- | --- | --- |
|  | 輸出 | 堆疊 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

(：左括號，忽略

a：輸出 a，輸出 = a

+：把 + 放入堆疊，堆疊 = {+}

(：左括號，忽略

b：輸出 b，輸出 = ab

x：把 + 放入堆疊，堆疊={+,x}

c：輸出 c，輸出 = abc

)：自堆疊中取出最上方的 x 並輸出，輸出 = abcx

)：自堆疊中取出最上方的 + 並輸出，輸出 = abcx+

+：把 + 放入堆疊，堆疊={+}

d：輸出 d，輸出 = abcx+d

+：自堆疊中取出最上方的 + 並輸出，輸出 = abcx+d+

|  |  |
| --- | --- |
| 中序轉後序運算 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <iostream>  #include <stack>  using namespace std;  int main(){  // 可以改成給使用者輸入  string str = "(a+(b\*c)+d)";  // 存放運算子的堆疊  stack<char> sign;  for(char c:str){  // 遇到左括號時不處理  if (c=='(') continue;  // 遇到 a-z（代表數字的變數）時直接輸出  else if (c<='z' && c>='a') cout << c;  // 遇到右括號，從堆疊中取出一個運算子  else if (c==')'){  cout << sign.top();  sign.pop();  }  // 遇到運算子（+、-、\*、/）， 放入堆疊  else sign.push(c);  }  return 0;  } |
| 執行結果 | |
| abc\*+d+ | |

4. Practice：LeetCode#224. 簡單計算機 Basic Calculator

A. 題目

給定一個字串 s，內容為合法的運算式。請寫出一個簡單的計算機來判斷該運算式結果並回傳。

注意：不可使用任何內建的運算式相關函式，如 eval() 來解題。

B. 出處：https://leetcode.com/problems/basic-calculator/

C. 說明

給定一個正確的算式（運算子只會出現加號 + 和減號 -），請算出該算式的結果，這題需要用到堆疊，但是不一定要先轉成後序運算，也可以直接從左到右讀取運算出結果。

D. 解題邏輯

使用兩個 stack，一個用來儲存運算元（數字），一個用來儲存加號和減號（分別用 1 和 -1 表達，避免需要資料轉型）。

|  |  |
| --- | --- |
| 簡單計算機 Basic Calculator | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | class Solution{  public:  int calculate(string s){  stack<int> numbers;  // +1：+ , -1：-  stack<int> signs;  // 為了處理大數，使用 long long  long long int value = 0;  // +1：+ , -1：-  int sign = +1;  // 因為與上一題不一樣，沒有小括號  // a+b+c 時從左到右運算需要儲存中間值  long long int result = 0;  for(char c:s){  // 遇到數字 0 到 9  if (c<='9' && c>='0'){  // 把 value 從 ascii 碼轉為 int，'9'->9  // 26 會變成 2\*10 + 6  // 依序遇到 2、6 時，把 2 乘以 10 倍後再加上 6  value = value\*10 +(c-'0');  }  // 遇到左括號，代表先前的運算結果暫告一段落  // 把 result 放到 numbers 中  else if (c=='('){  numbers.push(result);  // 把括號前出現的 + 或 - 放到 signs 中  signs.push(sign);  // 把 value, result, sign 回復到預設的狀態  value = result = 0;  sign = 1;  }  // 遇到右括號  else if(c==')'){  // sign 是目前這對括號裡面最後一個運算子  // value 是最後一個數字，比如 -6)  result += sign\*value;  // 乘以目前這對括號前面出現的正負號  // 比如 -(8-6)，result 要從 2 變成 -2  result \*= signs.top();  signs.pop();    // 把進入這次括號之前的數字重新加到 result 裡  // 注意遇到左括號的時候我們把 result 歸零了  // 左括號前面的結果是存進 numbers 裡  result += numbers.top();  numbers.pop();  // 把 value 和 sign 回復初始值  value = 0;  sign = 1;  }  // 遇到加號 '+'  else if (c=='+'){  // result 加上或減掉 + 號前面出現的數字  result += sign\*value;  // 重新設定 value 和 value  value = 0;  sign = 1;  }  // 遇到減號 '-'  else if (c=='-'){  // result 加上或減掉 - 號前面出現的數字  result += sign\*value;  // 重新設定 value 和 value  value = 0;  // 注意要把 sign 換成 -1  // 減號後面的數字會受到影響，-2 可以看成 +(-2)  sign = -1;  }  }  // 字串最後面如果是以數字而非右括號結尾  // 則最後一個數字還沒有被處理  // 上面遇到數字只會存到 value 裡，沒有影響到 result  if (value){  // (a+b)+c  result += sign\*value;  }  return result;  } // end of function solution  }; // end of class Calculate |