資料結構與物件導向設計

HW1
TO-DO LIST
REPORT

資工一A 113550193 李品翰

2025 .04 .20

一、資料結構

使用的資料結構：

```
vector<Basic_task2*>* li;
stack<vector<Basic_task2*>*>* undoStack;
stack<vector<Basic_task2*>*>* redoStack;
```

為何使用 stack 實作 undo/redo？

- **LIFO（Last-In-First-Out）特性**

  Undo/Redo 需要依「最後一次操作」反向恢復狀態，LIFO能記錄並回溯每個步驟

為何使用 vector 儲存任務清單？

- **動態配置記憶體**

  vector 能夠自動調整大小，
  方便使用者不斷新增、刪除任務。

- **編號對應 & 隨機存取效率高**

  任務通常會透過「編號」選取，
  vector[i] 支援 O(1) 的隨機存取，比 list（O(n)）快速。

- **排序方便**

  vector 可搭配 std::sort 使用 lambda 排序，
  list 雖有 .sort() 但不支援 lambda，較不方便。

二、基本功能

註：此部分程式碼較基礎，故僅列少部分於報告中

基本功能：

核心程式碼片段：

ADD A TASK

```
// 建立新任務並加入列表
li->push_back(
    new Basic_task2(name, category, deadline, note, imp)
);
```

使用者輸入任務名稱、分類、截止日期、重要度與備註，
程式會呼叫 li->push_back(new Basic_task2(...))，
將新任務放入 vector<Basic_task2*> 容器中

VIEW TASKS

```
// 依目前篩選／排序，走訪列表並列印
int* i = new int(0);
while (*i < li->size()) {
    int* index = new int(*i + 1);
    (*li)[*i]->printTask(index);
    delete index;
    ++(*i);
}
```

基本功能提供「全部任務」、「分類篩選」、「完成狀態篩選」三種模式，
透過迴圈走訪 vector，並呼叫 printTask() 列印每筆任務

基本功能：

核心程式碼片段：

EDIT A TASK

```
// 找到欲編輯的任務物件
Basic_task2* task = (*li)[*index - 1];
// 修改欄位
task->setName(&newName);
```

列出所有任務後，使用者輸入欲編輯的任務編號，取得對應 Basic_task2* 物件，再依使用者選擇呼叫 setName()、setCategory()、setDeadline()、setCompleted() 或 setImportance() 更新值

DELETE A TASK

```
// 釋放記憶體並從列表移除
delete (*li)[*index - 1];
li->erase(li->begin() + (*index - 1));
```

列出所有任務後，使用者輸入欲刪除的任務編號，程式先 delete 對應的 Basic_task2* 物件，再以 li->erase() 從 vector 中移除對應元素

三、加分功能

UNDO/REDO 功能（支援單層/多層）

1. 基本結構：兩個堆疊（stack）

```
stack<vector<Basic_task2*>*>* undoStack;
stack<vector<Basic_task2*>*>* redoStack;
```

- 這兩個堆疊用來記錄歷史快照
- undoStack 回復前一步，redoStack 記錄可以重做的操作
- 進行任何會改動清單內容的操作前（如 add、edit、delete），都會儲存目前狀態進 undoStack，準備給 undo 用

2. cloneState：複製當前任務清單

```
vector<Basic_task2*>* cloneState() {
    vector<Basic_task2*>* state = new vector<Basic_task2*>();
    int* i = new int(0);
    while (*i < li->size()) {
        state->push_back(((*li)[*i])->clone());
        ++(*i);
    }
    delete i;
    return state;
}
```

- 每次使用 saveState() 時呼叫，複製目前清單內容
- 透過 .clone() 複製每個任務，避免後續修改影響到快照內容

3. restoreState：還原任務清單

```
void restoreState(vector<Basic_task2*>* snapshot) {
    int* i = new int(0);
    while (*i < li->size()) {
        delete (*li)[*i];
        ++(*i);
    }
    delete i;
    li->clear();
    li->swap(*snapshot);
    delete snapshot;
}
```

- 每次執行 undoAction() 或 redoAction() 呼叫
- 用來把 snapshot 的內容搬進現在的任務清單 li，達到「復原」的效果。

4. clearSnapshots：清空整個堆疊

```
void clearSnapshots(stack<vector<Basic_task2*>*>* snapshots) {
    while (!snapshots->empty()) {
        vector<Basic_task2*>* snap = snapshots->top();
        int* i = new int(0);
        while (*i < snap->size()) {
            delete (*snap)[*i];
            ++(*i);
        }
        delete i;
        snapshots->pop();
        delete snap;
    }
}
```

- 在程式結束時會被呼叫，用來把 undoStack 和 redoStack 裡的所有記憶體釋放

5. saveState：儲存目前狀態 (用於 undo)

```
void saveState() {
    undoStack->push(cloneState());
    while (!redoStack->empty()) {
        vector<Basic_task2*>* snap = redoStack->top();
        int* i = new int(0);
        while (*i < snap->size()) {
            delete (*snap)[*i];
            ++(*i);
        }
        delete i;
        redoStack->pop();
        delete snap;
    }
}
```

- 每次要修改任務清單前（像是新增、刪除、編輯），都會先呼叫 saveState()
- 會複製一份當下任務清單存進 undoStack，以便日後能還原
- 同時把 redoStack 清空，因為做了新操作後，舊的 redo 歷史就失效了

6. undoAction：回復到前一步

```
void undoAction() {
    if (undoStack->empty()) {
        cout << "No actions to undo.\n";
        return;
    }
    redoStack->push(cloneState());
    vector<Basic_task2*>* snapshot = undoStack->top();
    undoStack->pop();
    restoreState(snapshot);
    cout << "Undo performed.\n";
}
```

- 如果可以 Undo，就先把目前清單存進 redoStack，再從 undoStack 拿出快照還原回去，讓使用者回到上一個狀態。

7. redoAction：重做剛剛被取消的

```
void redoAction() {
    if (redoStack->empty()) {
        cout << "No actions to redo.\n";
        return;
    }
    undoStack->push(cloneState());
    vector<Basic_task2*>* snapshot = redoStack->top();
    redoStack->pop();
    restoreState(snapshot);
    cout << "Redo performed.\n";
}
```

- 如果可以 Redo，就先把目前清單存回 undoStack，再從 redoStack 把快照拿出來還原，重做剛剛被取消的動作。

DEADLINE 功能 & 逾期紅字顯示

```
class Basic_task2 : public Basic_task {
protected:
    string* deadline;       // 截止日期
```

1. 基本結構

- 定義在 Basic_task2 裡
- 存的是 yyyy-mm-dd 格式的日期字串

```
while (true) {
    cout << "Enter task deadline (yyyy-mm-dd): ";
    getline(cin, *deadline);
    bool* valid = isValidDateFormat(deadline);
    if (*valid) {
        delete valid;
        break;
    }
    delete valid;
    cout << "Invalid date format. Please try again.\n";
}
```

2. 建構時設定 deadline

- 新增任務時，會要求使用者輸入 deadline
- 最後這個 *deadline 就會傳給建構子 → 設定進 deadline 欄位
- 做了數字與範圍檢查，不符合需重輸入（之後會提到的加分項目）

```
else if (*iu == '3') {
    string* newDeadline = new string();
    while (true) {
        cout << "Enter new deadline (yyyy-mm-dd): ";
        getline(cin, *newDeadline);
        bool* valid = isValidDateFormat(newDeadline);
        if (*valid) {
            delete valid;
            break;
        }
        delete valid;
        cout << "Invalid date format. Please try again.\n";
    }
    task->setDeadline(newDeadline);
    delete newDeadline;
}
```

3. 編輯時更新 deadline

- 編輯任務時也可以修改 deadline
- 一樣做了數字與範圍檢查

```
bool* isValidDateFormat(string* dateStr) {
    regex* pattern = new regex("^\\d{4}-\\d{2}-\\d{2}$");
    if (!regex_match(*dateStr, *pattern)) {
        delete pattern;
        return new bool(false);
    }
    delete pattern;

    int* year = new int;
    int* month = new int;
    int* day = new int;
    sscanf_s(dateStr->c_str(), "%d-%d-%d", year, month, day);

    if (*month < 1 || *month > 12) {
        delete year; delete month; delete day;
        return new bool(false);
    }

    int* daysInMonth = new int[12] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    bool* isLeap = new bool(*year % 4 == 0 && (*year % 100 != 0 || *year % 400 == 0));
    if (*isLeap && *month == 2)
        daysInMonth[1] = 29;

    bool* result = new bool(*day >= 1 && *day <= daysInMonth[*month - 1]);

    delete[] daysInMonth;
    delete isLeap;
    delete year; delete month; delete day;
    return result;
}
```

4. isValidDateFormat — 檢查格式合法

- 每次新增或修改任務時，會呼叫這個函式
  檢查輸入的 deadline 合不合法

※ 這部分在後面的「錯誤判斷」加分項會提到，
詳細解釋這裡就先不說

```
bool* isOverdue() const {
    time_t* raw = new time_t(time(nullptr));
    tm* nowInfo = new tm;
    localtime_s(nowInfo, raw);
    char* buf = new char[11];
    strftime(buf, 11, "%Y-%m-%d", nowInfo);
    string* today = new string(buf);
    bool* flag = new bool(*deadline < *today);

    delete[] buf;
    delete today;
    delete nowInfo;
    delete raw;
    return flag;
}
```

5. isOverdue() — 判斷是否逾期

- 用 time(nullptr) 取得現在的 timestamp
  (i.e. 自 1970-01-01 00:00:00 UTC 起經過多少秒)
- 再轉成本地日曆時間結構（年/月/日）
- 拿現在日期 today 和任務的 deadline 做字串比較，看是否已經過期
- 最後回傳布林指標

```
cout << *index << ". [" << (*completed ? "V" : "X") << "] "
     << *name << " (" << *category << ") "
     << *deadline << " "
     << string(*importance, '*');
if (*overdue) cout << "  !overdue!";
cout << "\n    note: " << *note << "\n";
```

6. printTask() — 文字提醒逾期任務

如果 *overdue == true，會在行尾加上 !overdue!
顯示逾期提示。

任務顏色提示

```
#define RED     "\033[31m"  // overdue
#define GREEN   "\033[32m"  // completed
#define YELLOW  "\033[33m"  // overdue & completed
#define RESET   "\033[0m"
```

1. ANSI 顏色巨集定義

- RED: 未完成
- GREEN: 已完成
- YELLOW: 逾期但已完成
- RESET: 在每次輸出完成後回復至預設色

```
void printTask(int* index) {
    bool* overdue = isOverdue();
    bool* done = completed;

    // 上色邏輯
    if (*overdue && *done) cout << YELLOW;     // 既逾期又已完成
    else if (*overdue) cout << RED;            // 逾期未完成
    else if (*done) cout << GREEN;             // 已完成且未逾期

    cout << *index << ". [" << (*completed ? "V" : "X") << "] "
        << *name << " (" << *category << ") "
        << *deadline << " "
        << string(*importance, '*');
    if (*overdue) cout << "  !overdue!";
    cout << "\n   note: " << *note << "\n";

    cout << RESET; // 顏色復原
    delete overdue;
}
```

2. printTask() — 印出任務顏色提示

- 依照「逾期 & 完成」、「逾期」、「完成」的優先順序設定顏色
- 輸出任務後，統一呼叫 RESET 還原顏色，避免影響後續輸出

任務重要度 (1-5) &
星號視覺化

```
class Basic_task2 : public Basic_task {
protected:
    int* importance;        // 重要度
```

1. 基本結構

- 定義在 Basic_task2 裡，表示任務的重要程度（數字越大越重要）
- 可在加入任務時輸入，也可以修改和顯示任務重要度
- 在 viewTasks 時，用星號數量代表權重

```
int* imp = new int();
while (true) {
    cout << "Enter importance (1-5, 5 = highest): ";
    if (!(cin >> *imp) || *imp < 1 || *imp > 5) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Please enter 1-5.\n";
        continue;
    }
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    break;
}
```

2. 建構時設定 importance

- 新增任務時，會要求使用者輸入 1 到 5 的數字。
- 最後這個 *imp 就會傳給建構子 → 設定進 importance 欄位
- 做了數字與範圍檢查，不符合需重輸入（之後會提到的加分項目）

```
else if (*iu == '5') {
    int* newImp = new int();
    while (true) {
        cout << "Enter new importance (1~5): ";
        if (!(cin >> *newImp) || *newImp < 1 || *newImp > 5) {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout << "Please enter 1-5.\n";
            continue;
        }
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        break;
    }
    task->setImportance(newImp);
    delete newImp;
```

3. 編輯時更新 importance

- 編輯任務時也可以修改重要度
- 一樣做了數字與範圍檢查

```
cout << *index << ". [" << (*completed ? "V" : "X") << "] "
     << *name << " (" << *category << ") "
     << *deadline << " "
     << string(*importance, '*'); // 用星號顯示重要度（1~5）
if (*overdue) cout << "  !overdue!";
cout << "\n    note: " << *note << "\n";
```

4. 顯示時用星號表示重要度

- 每次 printTask() 列印任務時，會根據 importance 數值，印出相對數量的 *

  e.g.
  importance = 3 → 顯示 ***
  importance = 5 → 顯示 *****

多種篩選／排序清單模式
（截止日期、今日清單、急迫清單）

```
else if (*iu == '4') {
    vector<Basic_task2*>* sortedTasks = new vector<Basic_task2*>();
    // 先把 li 裡的任務指標複製到新的 vector 裡
    sort(sortedTasks->begin(), sortedTasks->end(),
        [](Basic_task2* a, Basic_task2* b) -> bool {
            return *(a->getDeadline()) < *(b->getDeadline());
        });
    // 然後依序印出來
}
```

1. ViewTasks 選項 4：依截止日期排序

- 這段是用 sort() + lambda 排序 deadline，因為是 yyyy-mm-dd 的格式，字串直接比大小就能排序
- 用 printTask() 印出，照日期從最近排到最遠的

```
else if (*iu == '5') {
    char* todayBuf = new char[11];
    time_t* raw = new time_t(time(nullptr));
    tm* now = new tm;
    localtime_s(now, raw);
    strftime(todayBuf, 11, "%Y-%m-%d", now);

    string* today = new string(todayBuf);
    delete[] todayBuf;

    cout << "\nToday's tasks (" << *today << "):\n";
    int* shown = new int(0);
    int* index = new int(0);
    while (*index < li->size()) {
        if (*((*li)[*index]->getDeadline()) == *today) {
            int* idxPrint = new int(*index + 1);
            (*li)[*index]->printTask(idxPrint);
            delete idxPrint;
            ++(*shown);
        }
        ++(*index);
    }
    if (*shown == 0)
        cout << "None for today.\n";
```

2. ViewTasks 選項 5：Today's task (只顯示今天要做的)

取得今天的日期字串：

- 用 time(nullptr) 取得現在 timestamp
- 用 localtime_s 轉成當地時間結構（年/月/日）
- 再用 strftime 格式化成 "yyyy-mm-dd" 字串
- 最後轉成 string* ，跟任務的 deadline 比較

與 isOverdue() 的邏輯類似

跑過整個任務清單 li，逐一比對每筆任務的 deadline，
如果某筆任務的 deadline 等於今天，就印出來。
如果都沒有符合的，就輸出 "None for today."

```
else if (*iu == '6') {
    vector<Basic_task2*>* urgent = new vector<Basic_task2*>();
    int* index = new int(0);
    while (*index < li->size()) {
        urgent->push_back((*li)[*index]);
        ++(*index);
    }
    delete index;
```

3. ViewTasks 選項 6：Urgent List
（逾期 > 最近 > 重要度）

step 1：先把所有任務複製到一個新的 vector urgent 裡面
（因為不想改變原本 li 的排序）

```
bool* result;
if (*dA != *dB) {
    result = new bool(*dA < *dB);  // 天數少者優先
}
else if (*(a->getImportance()) != *(b->getImportance())) {
    result = new bool(*(a->getImportance()) > *(b->getImportance())); // 重要度高者優先
}
else {
    result = new bool(*(a->getDeadline()) < *(b->getDeadline()));     // 日期較早者優先
}
```

step 2：對 urgent 這個清單做 sort 排序
自定義一個排序方法，邏輯分成三層順序：
1. 越早到期越前面（即使已經逾期）
2. 如果兩筆任務剩餘天數一樣，重要度高的排前面
3. 如果一樣重要，再照 deadline 順序排

```
    cout << "\nUrgent list (overdue first, then soonest):\n";
    int* indexPrint = new int(1);
    for (auto* t : *urgent) {
        t->printTask(indexPrint);
        ++(*indexPrint);
    }
```

step 3：印出排序後的 urgent 清單
每個任務照順序呼叫 printTask() 印出，
顯示目前最急迫要處理的項目

一次性密碼系統

```
fstream* file = new fstream("password.txt", ios::in | ios::out | ios::app);

cout << "Welcome to the Todo List Application!\n";
cout << "Is this your first time using it? (y/n): ";
string* firstTime = new string();
cin >> *firstTime;
cin.ignore(numeric_limits<streamsize>::max(), '\n');

if (*firstTime == "y") {
    cout << "Please set up your password: ";
    string* password = new string();
    getline(cin, *password);
    file->close();
    file->open("password.txt", ios::out | ios::trunc);
    *file << *password;
    *file << "\n";
    cout << "Password set successfully!\n";

    delete password;

}
```

1. 第一次使用時（設定密碼）

- 啟動時先詢問使用者是否第一次使用
- 如果回答‘y’，系統會請使用者設定一組新密碼
- 接著會把這組密碼寫入 password.txt 檔案中，並使用 ios::trunc 模式保證只留這一筆密碼
- 為了正確寫入，程式會先把原本開啟的 fstream 關閉，再重新以覆蓋模式開啟
- 最後顯示“Password set successfully!”

```
else if (*firstTime == "n") {
    cout << "Welcome back!\n";

    file->clear();
    file->seekg(0, ios::beg);

    cout << "Please enter your password: ";
    string* password = new string();
    getline(cin, *password);

    string* storedPassword = new string();
    getline(*file, *storedPassword);

    if (*password != *storedPassword) {
        cout << "Incorrect password, exiting.\n";
        delete password;
        delete storedPassword;
        delete firstTime;
        delete file;
        return 0;
    }
    else {
        cout << "Login successful!\n";
    }

    delete password;
    delete storedPassword;

}
```

2. 之後登入時（驗證密碼）

- 系統會請使用者輸入先前設定的密碼
- 然後從 password.txt 中讀出正確密碼，拿來比對
- 如果密碼輸入錯誤，就直接輸出錯誤訊息並 return 0 結束程式，防止未授權操作
- 如果密碼正確，才會進入 run() 執行待辦清單的主功能

支援任務註解

```
class Basic_task2 : public Basic_task {
protected:
    string* note;           // 備註
```

1. 基本結構

- 定義在 Basic_task2 裡，表示任務可以備註
- 在 constructor / destructor / copy constructor 都有處理

```
cout << "Enter task note (optional): ";
getline(cin, *note);
if (note->empty()) {
    *note = "No notes";
}
```

2. 建構時設定 note

- 使用者在新增任務的時候可以輸入註解
- 預設值設定（之後會提到的加分項目）

```
else if (*iu == 6) {
    string* newNote = new string();
    cout << "Enter new note: ";
    getline(cin, *newNote);
    task->setNote(newNote);
    delete newNote;
}
```

3. 編輯時更新 note

- 編輯任務時也可以修改註解

簡化使用者輸入
(預設值、選單、自動忽略空白)

1. 預設值機制：

(a) 任務名稱 name（空白自動變成 "I don't know"）

```
cout << "Enter task name (I don't know): ";
getline(cin, *name);
if (name->empty()) {
    *name = "I don't know";
}
```

(b) 任務備註 note（空白自動變成 "No notes"）

```
cout << "Enter task note (optional): ";
getline(cin, *note);
if (note->empty()) {
    *note = "No notes";
}
```

```
void addTask() {
    // ...
    cout << "Enter task category:\n"
         << "1. School\n"
         << "2. Work\n"
         << "3. Personal\n"
         << "4. Other\n";
```

2. 簡易數字選單：

為了讓使用者輸入更方便，
新增任務時會先列出三個常見分類（School / Work / Personal），
可以直接輸入數字選擇，
如果輸入 4，則再請使用者輸入自訂的類別名稱。

3. 自動忽略空白行或多餘輸入：

```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
getline(cin, *category);
```

所有輸入後面都有搭配 cin.ignore()，
確保接下來用 getline() 不會被之前輸入
留下來的 ‘\n’ 吃掉。

錯誤判斷與防呆
（輸入驗證、重試機制）

1. 數值輸入錯誤（例如選單、類別、重要度輸入）：

```
if (!(cin >> *g)) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "Input must be a number, please try again.\n";
    continue;
}
```

先用 cin >> 嘗試讀入數字，
如果讀入失敗（例如輸入英文字母），
會清除錯誤狀態並跳過整行，再次要求輸入

2. 範圍檢查（例如重要度需為 1~5）：

```
if (!(cin >> *imp) || *imp < 1 || *imp > 5) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "Please enter 1-5.\n";
    continue;
}
```

除了判斷有沒有輸入數字，
還要判斷是否落在 1~5 的範圍內，
不符合就提示錯誤並重來

3. 任務編號輸入錯誤（edit/delete 時）：

```
if (!(cin >> *index)
    || *index < 1
    || *index > li->size()) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "Index must be a number between 1 and "
         << li->size() << ".\n";
    delete index;
    return;
}
```

防止輸入不存在的任務編號，
會提示使用者任務數量的上下限

"Index must be a number between 1 and li->size()"

4. 日期格式驗證：

```
bool* valid = isValidDateFormat(deadline);
if (*valid) {
    delete valid;
    break;
}
delete valid;
cout << "Invalid date format. Please try again.\n";
```

檢查日期是否合法，分別檢查以下兩個部分。
若不合法，要求使用者重新輸入。

```
regex* pattern = new regex("^\\d{4}-\\d{2}-\\d{2}$");
if (!regex_match(*dateStr, *pattern)) {
    delete pattern;
    return new bool(false);
}
delete pattern;
```

(a) 用 regex 檢查使用者輸入的字串是否為
“yyyy-mm-dd”格式 (e.g. 2025-04-20)

```
// 月份檢查
if (*month < 1 || *month > 12) {
    delete year; delete month; delete day;
    return new bool(false);
}
// 每月最大天數
int* daysInMonth = new int[12] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
// 閏年處理
bool* isLeap = new bool(*year % 4 == 0 && (*year % 100 != 0 || *year % 400 == 0));
if (*isLeap && *month == 2)
    daysInMonth[1] = 29;

bool* result = new bool(*day >= 1 && *day <= daysInMonth[*month - 1]);
```

(b) 依序進行以下三種邏輯判斷：

1. 月份範圍是否合法（1~12月）
2. 判斷每個月的最大天數
3. 處理閏年問題（2 月變 29 天）

THANK YOU

報告到此結束，感謝參閱