

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI PHÂN HIỆU TẠI TP. HỒ CHÍ MINH

## BỘ MÔN CÔNG NGHỆ THÔNG TIN



**BÁO CÁO BÀI TẬP LỚN**

MÔN LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

# ĐỀ TÀI: QUẢN LÝ SỰ KIỆN

Giảng viên hướng dẫn: Trần Thị Dung

Sinh viên thực hiện:

-Nguyễn Hồ Hưng(NT)-6151071058

* Dương Văn Hoàng– 6151071050

-Đặng Đình Sang– 6151071092

Lớp: CQ.61.CNTT

TP. Hồ Chí Minh, ngày 17 tháng 11 năm 2021

# 

# LỜI CẢM ƠN

Lời đầu tiên, nhóm xin cảm ơn đến nhà trường, quý thầy cô Bộ môn trường Đại học Giao thông Vận tải – Phân hiệu tại TP. Hồ Chí Minh đã tạo điều kiện cho nhóm có môi trường học tập và nghiên cứu thật tốt. Xin cảm ơn đến quý thầy, cô đã hết lòng quan tâm, dạy dỗ và trang bị cho nhóm những kiến thức và kinh nghiệm quý báu.

Nhóm xin trân trọng cảm ơn cô Trần Thị Dung đã trực tiếp giảng dạy môn Lập trình hướng đối tượng cũng như đã hướng dẫn nhóm trong quá trình nghiên cứu lý thuyết và tiến hành thực hiện đề tài này. Xin cảm ơn thầy đã giúp đỡ nhóm thực hiện tốt đề tài.

Xin gửi lời cảm ơn đến các thành viên trong nhóm, các bạn học đã cố gắng cùng nhau để hoàn thành báo cáo đề tài và góp ý để hoàn thiện bản báo cáo đề tài này.

Trong quá trình nghiên cứu, cũng như là trong quá trình làm bài báo cáo đề tài, nhóm khó tránh khỏi sai sót, rất mong quý thầy bỏ qua. Đồng thời do trình độ lý luận cũng như kinh nghiệm thực tiễn còn hạn chế nên bài báo cáo về mặt nội dung cũng như trình bày không thể tránh khỏi những thiếu sót, nhóm rất mong nhận được ý kiến đóng góp của thầy để nhóm học thêm được nhiều kinh nghiệm và sẽ hoàn thành tốt hơn bài những bài báo cáo sắp tới.

Cuối cùng em kính chúc quý thầy dồi dào sức khỏe và thành công trong sự nghiệp cao quý. Đồng kính chúc các quý thầy cô trong bộ môn, cùng các thành viên trong nhóm và các bạn học trong lớp luôn dồi dào sức khỏe, đạt được nhiều thành công tốt đẹp trong công việc.

*Nhóm xin chân thành cảm ơn!*

*TP. Hồ Chí Minh, ngày 15 tháng 11 năm 2021*

nhóm sinh viên thực hiện

Mục lục

[**LỜI CẢM ƠN 1**](#_Toc88349340)

[**Phần 1 : Dịch Sách 3**](#_Toc88349341)

[Chương 16: Class Basics 3](#_Toc88349342)

[Chương 17: Construction, Cleanup, Copy and Move 6](#_Toc88349343)

[Chương 18. Nạp chồng toán tử 12](#_Toc88349344)

[Chương 19 : Toán tử đặc biệt 15](#_Toc88349346)

[Chương 20 : Các lớp dẫn xuất 20](#_Toc88349347)

[Chương 21: Phân cấp lớp 26](#_Toc88349348)

[**Phần 2 : Chương trình 30**](#_Toc88349349)

[I. Giới thiệu đề tài. 30](#_Toc88349350)

[1.Lí do chọn đề tài 30](#_Toc88349351)

[2.Mô tả bài toán 30](#_Toc88349352)

[3.Phân công công việc và thực hiện 30](#_Toc88349353)

[II. Nội dung cơ sở lý thuyết 31](#_Toc88349354)

[1 .Tính chất hướng đối tượng 31](#_Toc88349355)

[2.Cấu trúc dữ liệu và giải thuật dùng trong bài toán 34](#_Toc88349356)

[3.Đồ họa – Ví dụ 40](#_Toc88349357)

[I. Giao diện chương trình 45](#_Toc88349358)

[IV.Phụ lục 52](#_Toc88349359)

[1.Link github 52](#_Toc88349360)

[2. Hướng dẫn cài đặt chương trình: 53](#_Toc88349361)

[3.Hướng dẫn sử dụng chương trình : 53](#_Toc88349362)

# Phần 1 : Dịch Sách

## Chương 16: Class Basics

* **Lớp là gì? Định nghĩa class**
  + Một class là một kiểu do người dùng đặt
  + Một class bao gồm nhiều thành viên
  + Các hàm thành viên có thể xác định công dụng như khởi tạo, sao chép, di chuyển hay loại bỏ
  + Khi truy cập các thành viên ta dùng dấu chấm hoặc dùng dấu mũi tên cho con trỏ
  + Các toán tử như +,! Và [] có thể dùng để định nghĩa cho một Class
  + Một class là một không gian tên chứa các thành viên của nó
  + Các thành viên công khai cung cấp giao diện và các thành viên riên tư cung cấp các nội dung, chi tiết
  + Một cấu trúc là một class mà các thành viên được mặc định là công khai

**16.2.1: Member Funtion**

Không có một kết nào giữa các kiểu dữ liệu, các liên kết sẽ được thiết lập bằng cách khai báo các chức năng của hàm thành viên.

Các hàm được khai báo trong class (struct là một class) được gọi là thành viên và chỉ có thể được gọi cho một biến cụ thể thích hợp bằng cách sử dụng cú pháp thích hợp để truy cập thành viên:

Các cấu trúc khác nhau có thể có các hàm thành viên cùng tên, chúng ta phái chỉ định tên cấu trúc khi xác định một function member.

Trong một member function, tên member có thế được sử dụng mà không cần tham chiếu rõ ràng đến một đối tượng. Trong trường hợp, tên đề cập đến thành viên của đối tượng hàm được gọi như: khi Date::init () được gọi cho today, m = mm gán cho today.m, khi Date::init () được gọi cho my\_birthday, một hàm thành viên trong lớp “knows” đã được gọi cho một đối tượng.

**16.2.2 Default Copying**

Mặc định các đối tượng có thể sao chép. Đặc biệt một đối tượng class có thể được khởi tạo bằng một bản sao của một đối tượng trong class đó.

Thông thường, bản sao của một đối tượng class là bản sao của mỗi thành viên. Các đối tượng class có thể được sao chép thep phép gán

**16.2.3 Access Control**

Khai báo Date trước các hàm để thao tác được, nó không bắt buộc rằng những chức năng đóphải là những chức năng duy nhất phụ thuộc vào các đối tượng của class Date. Hãy sử dụng một class thay vì struct.

Public phân tách nội dung của class thành hai phần. Đầu tiên là private chỉ được sử dụng bởi các chức năng của hàm thành viên. Thứ hai là Public tạo thành giao diện công khai cho các đối tượng của class. Một cấu trúc đơn giản là một class có các thành viên công khai theo mặc định. Các chức năng của thành viên được định nghĩa và sử dụng bình thường như trước đây.

Tuy nhiên, các chức năng nonmember bị cấm sử dụng các thành viên Private.

Hàm init() rất cần thiết vì việc đặt dữ liệu ở chế độ Private buộc chúng ta phải khởi tạo thành viên.

Nếu ta thay đối đại diện của một lớp, chúng ta chỉ cần thay đổi chức năng của hàm thành viên để tận dụng lợi thế của đại diện mới.

**16.2.4 class and struct**

Xây dựng: class X {…}; gọi là định nghĩa lớp.

Theo định nghĩa, struct là một lớp trong đó các thành viên được mặc định là public.

Hai định nghĩa này của S có thể thay thế cho nhau, và ta có thể sử dụng tuỳ ý phù hợp vào hoàn cảnh và sở thích. Các hàm tạo và hàm truy cập có thể khá hữu ích ngay cả đối với các cấu trúc, nhưng như một cách viết tắt chứ không phải là người bảo đảm bất biến. Mặc định các thành viên của một class là private.

Tuy nhiên, chúng ta cũng có thể sử dụng công cụ xác định quyền truy cập private: để nói rằng các thành viên sau đây là riêng tư, giống như công public: nói rằng các thành viên sau đây là công khai.

Không bắt buộc phải khai báo dữ liệu trước trong một class. Trên thực tế, việc đặt dữ liệu thường có ý nghĩa thành viên cuối cùng nhấn mạnh các chức năng cung cấp giao diện người dùng công khai.

Trong thực tế,cả giao diện public và chi tiết triển khai thường nhiều hơn mở rộng hơn trong các ví dụ hướng dẫn. Các chỉ định truy cập có thể được sử dụng nhiều lần trong một khai báo lớp duy nhất.

**16.2.5 Constructors**

Việc sử dụng các hàm như init () để cung cấp khởi tạo cho các đối tượng lớp là không phù hợp và có lỗi.

Hàm xây dựng các giá trị của một kiểu nhất định, nó được gọi là một hàm tạo. Một phương thức khởi tạo được công nhận bởi có cùng tên với chính lớp đó.

Khi một lớp có một phương thức khởi tạo, tất cả các đối tượng của lớp đó sẽ được khởi tạo bởi một lời gọi khởi tạo.

Khi một giá trị đối số được sử dụng để biểu thị " chọn giá trị mặc định ", thì giá trị được chọn phải nằm ngoài tập hợp các giá trị có thể có cho đối số. Đối với ngày và tháng, điều này rõ ràng là như vậy, nhưng đối với năm, không có thể không phải là một lựa chọn hiển nhiên.

**16.2.6 explicit Constructors**

Theo mặc định, một hàm tạo được gọi bởi một đối số hoạt động như một chuyển đổi ngầm định từ kiểu đối số sang kiểu của nó.

Một khởi tạo với dấu = được coi là một khởi tạo sao chép. Về nguyên tắc, một bản sao của trình khởi tạo được đặt vào đối tượng đã khởi tạo. Tuy nhiên, một bản sao như vậy có thể được tối ưu hóa và hoạt động có thể được sử dụng nếu trình khởi tạo là một value. Khởi tạo rõ ràng được gọi là khởi tạo trực tiếp.

**16.2.7 In-Class Initializers**

Khi chúng ta sử dụng một số hàm tạo, việc khởi tạo thành viên có thể trở nên lặp lại.

**16.2.8 In-Class Function Definitions**

Một thành viên có thể đề cập đến một thành viên khác trong lớp của mình một cách độc lập với vị trí của thành viên đó định nghĩa.

**16.2.9 Mutability**

Chúng ta có thể định nghĩa một đối tượng được đặt tên như một hằng số hoặc một biến. Nói cách khác, một cái tên có thể đề cập đến một đối tượng chứa một giá trị không thể thay đổi hoặc một giá trị có thể thay đổi. Để trở nên hữu ích ngoài định nghĩa về các hằng số đơn giản của các kiểu tích hợp, chúng ta phải có khả năng định nghĩa các hàm hoạt động trên các đối tượng const thuộc kiểu do người dùng định nghĩa. Đối với các lớp, điều đó có nghĩa là chúng ta phải có khả năng xác định các hàm thành viên hoạt động trên các đối tượng const.

**16.2.9.1 Constant Member Functions**

Khi một hàm thành viên const được định nghĩa bên ngoài lớp của nó, hậu tố const là bắt buộc.

Một hàm thành viên const có thể được gọi cho cả đối tượng const và không phải const, trong khi một Hàm thành viên const chỉ có thể được gọi cho các đối tượng không phải const.

**16.2.9.2 Physical and Logical Constness**

Đôi khi, một hàm thành viên là const về mặt logic, nhưng nó vẫn cần thay đổi giá trị của một member. Có nghĩa là, đối với người dùng, hàm dường như không thay đổi trạng thái của đối tượng, mà là một số chi tiết người dùng không thể quan sát trực tiếp được cập nhật.

**16.2.9.3 mutable**

Chúng ta có thể định nghĩa một thành viên của một lớp là có thể thay đổi.

**16.2.9.4 Mutability through Indirection**

Khai báo một thành viên có thể thay đổi là thích hợp nhất khi chỉ một phần nhỏ của biểu diễn của một đối tượng nhỏ được phép thay đổi. Các trường hợp phức tạp hơn thường được xử lý tốt hơn bằng cách đặt thay đổi dữ liệu trong một đối tượng riêng biệt và truy cập nó một cách gián tiếp.

Lưu ý rằng const không áp dụngcho các đối tượng được truy cập thông qua con trỏ hoặc tham chiếu. Người đọc có thể coi một đối tượng như vậy là '' một loại đối tượng subobject '', nhưng trình biên dịch thì không biết các con trỏ hoặc tham chiếu đó có bất kỳ khác biệt nào với bất kỳ con trỏ nào khác hay không. Đó là, một con trỏ thành viên không có bất kỳ ngữ nghĩa đặc biệt nào để phân biệt nó với các con trỏ khác.

**16.2.10 Self-Reference**

Trong một hàm thành viên không tĩnh, từ khóa này là một con trỏ đến đối tượng mà hàm đã được gọi. Trong một hàm thành viên const của lớp X, kiểu này là const X ∗ để ngăn chặn việc sửa đổi chính nó.

**16.2.11 Member Access**

Một thành viên của lớp có thể được truy cập bằng cách áp dụng. (dot) toán tử cho một đối tượng của lớp hoặc toán tử -> (mũi tên) cho một con trỏ đến một đối tượng thuộc lớp.

Từ bên trong một lớp không cần toán tử.

Nếu chúng ta muốn tham chiếu đến một thành viên nói chung, thay vì một thành viên của một đối tượng cụ thể, chúng ta đủ điều kiện bởi tên lớp theo sau là ::.

**16.2.12 [static] Members**

Một biến là một phần của một lớp, nhưng không phải là một phần của một đối tượng của lớp đó, được gọi là thành viên tĩnh.

Một thành viên tĩnh có thể được giới thiệu như bất kỳ thành viên nào khác.

Một thành viên tĩnh có thể được tham chiếu mà không cần đề cập đến một đối tượng. Thay vào đó, tên của nó đủ điều kiện theo tên của lớp của nó.

**16.2.13 Member Types**

Một lớp thành viên (thường được gọi là lớp lồng nhau) có thể tham chiếu đến các kiểu và các thành viên tĩnh trong bao bọc của lớp.

Một lớp lồng nhau có quyền truy cập vào các thành viên của lớp bao quanh nó.

**16.3 Concrete Classes**

**16.3.1 Member Functions**

Khởi tạo là một hoạt động tương đối phức tạp vì nó liên quan đến xác thực dữ liệu.

**16.3.2 Helper Functions**

Một lớp có một số hàm được liên kết với nó mà không cần được định nghĩa trong lớp bởi vì họ không cần quyền truy cập trực tiếp vào phần đại diện.

Việc xác định các chức năng trong bản thân lớp sẽ làm phức tạp giao diện lớp và tăng số lượng các chức năng có thể cần phải được kiểm tra khi một thay đổi đối với biểu diễn được xem xét.

Chúng ta có thể làm cho liên kết rõ ràng bằng cách bao quanh lớp và chức năng trợ giúp trong một không gian tên.

**16.3.3 Overloaded Operators**

Việc sử dụng các toán tử đã định nghĩa của chúng gần như là bắt buộc.

**16.3.4 The Significance of Concrete Classes**

**16.4 Advice**

* Biểu diễn các khái niệm dưới dạng các lớp
* Tách giao diện của một lớp khỏi việc triển khai nó
* Chỉ sử dụng dữ liệu công khai (struct) khi nó thực sự chỉ là dữ liệu và không có bất biến nào có ý nghĩa đối với dữ liệu thành viên
* Định nghĩa một phương thức khởi tạo để xử lý việc khởi tạo các đối tượng
* Theo mặc định, khai báo các hàm tạo đối số đơn là tường minh
* Khai báo một hàm thành viên không sửa đổi trạng thái của đối tượng của nó là const
* Loại concrete là loại đơn giản nhất của lớp. Nếu có thể, hãy thích loại concrete hơn các lớp phức tạp hơn và cấu trúc dữ liệu đơn giản hơn
* Chỉ biến một hàm thành thành viên nếu nó cần truy cập trực tiếp vào biểu diễn của một lớp
* Sử dụng một không gian tên để làm cho mối liên kết giữa một lớp và các chức năng trợ giúp của nó trở nên rõ ràng
* Biến một hàm thành viên không sửa đổi giá trị của đối tượng thành một hàm thành viên const
* Tạo một hàm cần quyền truy cập vào biểu diễn của một lớp nhưng không cần được gọi cho một đối tượng cụ thể một hàm thành viên tĩnh

## Chương 17: Construction, Cleanup, Copy and Move

**17.1 Introduction**

Có năm tình huống trong đó một đối tượng được sao chép hoặc di chuyển:

• Là nguồn gốc của một nhiệm vụ

• Là một bộ khởi tạo đối tượng

• Là một đối số của hàm

• Là một hàm trả về giá trị

• Như một ngoại lệ

**17.2 Constructors and Destructors**

Chúng ta có thể chỉ định cách khởi tạo một đối tượng của một lớp bằng cách xác định một phương thức khởi tạo.

**17.2.1 Constructors and Invariants**

Một thành viên có cùng tên với lớp của nó được gọi là một phương thức khởi tạo.

Một khai báo phương thức khởi tạo chỉ định một danh sách đối số nhưng không có giá trị trả về kiểu.

Công việc của một phương thức khởi tạo là khởi tạo một đối tượng của lớp nó.

Tại sao bạn định nghĩa một bất biến?

• Để tập trung nỗ lực thiết kế cho lớp

• Để làm rõ hành vi của lớp

• Để đơn giản hóa định nghĩa của các hàm thành viên

• Để làm rõ việc quản lý tài nguyên của lớp

• Để đơn giản hóa tài liệu của lớp

**17.2.2 Destructors and Resources**

Một hàm tạo khởi tạo một đối tượng tạo ra môi trường mà thành viên các chức năng hoạt động.

Quản lý tài nguyên dựa trên hàm tạo / hủy này được gọi là Thu thập tài nguyên.

Các vùng chứa thư viện tiêu chuẩn, chẳng hạn như vectơ và bản đồ không có thứ tự, sử dụng các biến thể của kỹ thuật này để cung cấp khả năng lưu trữ cho các phần tử của chúng.

**17.2.3 Base and Member Destructors**

Một hàm xây dựng một đối tượng lớp:

[1] Đầu tiên, hàm tạo gọi các hàm tạo lớp cơ sở của nó

[2] Sau đó, nó gọi các hàm tạo thành viên

[3] Cuối cùng, nó tự thực thi.

Một hàm hủy một đối tượng theo thứ tự ngược lại:

[1] Đầu tiên, trình hủy thực thi phần thân

[2] Sau đó, gọi các trình hủy thành viên

[3] Cuối cùng, nó gọi các hàm hủy lớp cơ sở

**17.2.4 Calling Constructors and Destructors**

Một hàm hủy được gọi ngầm khi thoát khỏi phạm vi bằng cách xóa.

Trình hủy phải được gọi một cách rõ ràng.

Trong số hai lựa chọn thay thế, sử dụng private là linh hoạt hơn.

**17.2.5 virtual Destructors**

Một hàm hủy có thể được khai báo là ảo và thường phải dành cho một lớp có hàm ảo.

Lý do chúng ta cần một trình hủy ảo là một đối tượng thường được thao tác thông qua giao diện được cung cấp bởi một lớp cơ sở cũng thường bị xóa thông qua giao diện đó.

**17.3 Class Object Initialization**

**17.3.1 Initialization Without Constructors**

Chúng ta không thể định nghĩa một hàm tạo cho một kiểu dựng sẵn, nhưng chúng ta có thể khởi tạo nó với một giá trị phù hợp.

Chúng ta có thể khởi tạo các đối tượng của một lớp mà chúng ta chưa xác định hàm tạo bằng cách sử dụng

• Khởi tạo thành viên

• Sao chép khởi tạo

• Khởi tạo mặc định

Trường hợp không có hàm tạo nào yêu cầu đối số được khai báo, thì cũng có thể loại bỏ hoàn toàn bộ khởi tạo.

Bạn có thể sử dụng Buf làm biến cục bộ mà không cần khởi tạo nó trước khi sử dụng nó làm mục tiêu cho đầu vào hoạt động.

Khởi tạo thành viên chỉ hoạt động nếu chúng ta có thể truy cập các thành viên.

Nếu một lớp có một thành viên dữ liệu không tĩnh riêng, nó cần một hàm tạo để khởi tạo nó.

**17.3.2 Initialization Using Constructors**

Trường hợp bản sao thành viên là không đủ hoặc không mong muốn, một phương thức khởi tạo có thể được xác định để khởi tạo một sự vật.

Một hàm tạo thường được sử dụng để thiết lập một bất biến cho lớp của nó và để lấy nguồn lực cần thiết để thực hiện điều đó.

Nếu một hàm tạo được khai báo cho một lớp, một số hàm tạo sẽ được sử dụng cho mọi đối tượng.

**17.3.2.1 Initialization by Constructors**

Bạn cần phân biệt giữa khởi tạo với danh sách các phần tử và danh sách các đối số của phương thức khởi tạo.

**17.3.3 Default Constructors**

Một hàm tạo có thể được gọi mà không cần đối số được gọi là một hàm tạo mặc định.

Một hàm tạo mặc định được sử dụng nếu không có đối số nào được chỉ định hoặc nếu danh sách bộ khởi tạo trống được cung cấp.

Một đối số mặc định có thể tạo một hàm tạo nhận các đối số vào một hàm tạo mặc định.

Các hàm tạo cho các kiểu dựng sẵn thường được sử dụng nhiều nhất cho các đối số mẫu.

**17.3.4 Initializer-List Constructors**

Danh sách bộ khởi tạo có thể có độ dài tùy ý nhưng phải đồng nhất.

**17.3.4.1 initializer\_list Constructor Disambiguation**

Các quy tắc:

• Nếu một phương thức khởi tạo mặc định hoặc một phương thức khởi tạo danh sách khởi tạo có thể được gọi, hãy ưu tiên xây dựng mặc định.

• Nếu có thể gọi cả hàm tạo danh sách bộ khởi tạo và '' hàm tạo thông thường '', hãy ưu tiên phương thức khởi tạo danh sách.

Quy tắc đầu tiên, '' ưu tiên phương thức tạo mặc định hơn '', là cách hiểu thông thường: chọn câu lệnh đơn giản nhất.

Quy tắc thứ hai, ‘‘ ưu tiên phương thức khởi tạo danh sách khởi tạo ’’ là cần thiết để tránh các độ phân giải khác nhau dựa trên số lượng phần tử khác nhau.

**17.3.4.2 Use of initializer\_lists**

Các phần tử của một danh sách khởi tạo là không thể thay đổi.

Một vùng chứa có thể triển khai một phương thức khởi tạo danh sách.

Danh sách trình khởi tạo là một phần của thiết kế khởi tạo phổ biến và thống nhất.

**17.3.4.3 Direct and Copy Initialization**

Đối với một vùng chứa, sự phân biệt được áp dụng cho cả vùng chứa và các yếu tố của nó:

• Hàm khởi tạo danh sách bộ khởi tạo của vùng chứa có thể rõ ràng hoặc không.

• Hàm tạo của kiểu phần tử của danh sách bộ khởi tạo có thể rõ ràng hoặc không.\

Một vùng chứa có thể có một số hàm tạo rõ ràng và một số thì không.

**17.4 Member and Base Initialization**

Các nhà xây dựng có thể thiết lập các bất biến và thu được các tài nguyên.

**17.4.1 Member Initialization**

Một lớp có thể được sử dụng để giữ thông tin cho một tổ chức nhỏ.

Danh sách trình khởi tạo thành viên bắt đầu bằng dấu hai chấm và các trình khởi tạo thành viên riêng lẻ được phân tách bằng dấu phẩy.

Các hàm tạo của thành viên được gọi trước phần thân của hàm tạo riêng của lớp chứa được thực thi.

Các hàm tạo được gọi theo thứ tự mà các thành viên được khai báo trong lớp chứ không phải thứ tự mà các thành viên xuất hiện trong danh sách trình khởi tạo.

Các hàm hủy thành viên được gọi theo thứ tự ngược lại của xây dựng sau khi phần thân của trình hủy của chính lớp đó đã được thực thi.

**17.4.1.1 Member Initialization and Assignment**

Bộ khởi tạo thành viên rất cần thiết cho các kiểu mà ý nghĩa của việc khởi tạo khác với của sự phân công.

**17.4.2 Base Initializers**

Các cơ sở của một lớp dẫn xuất được khởi tạo giống như cách các thành viên không phải là dữ liệu.

Thứ tự khởi tạo là thứ tự khai báo và nên chỉ định các trình khởi tạo cơ sở theo thứ tự đó. Các căn cứ được khởi tạo trước khi có thành viên và bị hủy sau khi có thành viên.

**17.4.3 Delegating Constructors**

Bạn không thể vừa ủy quyền vừa khởi tạo rõ ràng một thành viên.

Việc ủy quyền bằng cách gọi một phương thức khởi tạo khác trong danh sách thành viên và bộ khởi tạo cơ sở của một phương thức khởi tạo là rất khác với việc gọi một cách rõ ràng một phương thức khởi tạo trong phần thân của một phương thức khởi tạo.

Khi sử dụng một phương thức khởi tạo ủy quyền, đối tượng không được coi là đã xây dựng cho đến khi phương thức khởi tạo ủy quyền hoàn thành - chỉ hoàn thành hàm tạo được ủy quyền là không đủ.

Một trình hủy sẽ không được gọi cho một đối tượng trừ khi phương thức khởi tạo ban đầu của nó đã hoàn thành.

**17.4.4 In-Class Initializers**

Chúng ta có thể chỉ định một bộ khởi tạo cho một thành viên dữ liệu không tĩnh trong khai báo lớp.

Nếu một thành viên được khởi tạo bởi cả trình khởi tạo trong lớp và một phương thức khởi tạo, thì chỉ phương thức khởi tạo khởi tạo được thực hiện.

Trình khởi tạo thành viên trong lớp có thể sử dụng các tên có trong phạm vi tại thời điểm chúng được sử dụng trong khai báo thành viên.

**17.4.5 static Member Initialization**

Một thành viên lớp tĩnh được cấp phát tĩnh chứ không phải là một phần của mỗi đối tượng của lớp.

Có thể khởi tạo một thành viên tĩnh trong khai báo lớp. Phần tử tĩnh phải là một hằng số của kiểu tích phân hoặc kiểu liệt kê, hoặc một mã liên kết của một kiểu chữ và bộ khởi tạo phải là một biểu thức hằng.

Việc sử dụng chính của các hằng số thành viên là cung cấp các tên tượng trưng cho các hằng số cần thiết ở những nơi khác trong khai báo lớp.

**17.5 Copy and Move**

Khi chúng ta cần chuyển một giá trị từ a sang b, chúng ta thường có hai lựa chọn khác nhau về mặt logic:

• Sao chép là ý nghĩa quy ước của x = y; nghĩa là, hiệu quả là các giá trị của x và y là

cả hai đều bằng giá trị của y.

• Di chuyển các lá x với giá trị cũ của y và y với một số trạng thái đã chuyển. Vùng chứa, trạng thái được chuyển từ đó là '' trống ''.

Sự phân biệt logic đơn giản này bị nhầm lẫn bởi truyền thống và thực tế là chúng ta sử dụng cùng một ký hiệu cho cả di chuyển và sao chép.

**17.5.1 Copy**

Sao chép cho một lớp X được xác định bằng hai phép toán:

• Sao chép hàm tạo: X (const X &)

• Sao chép phép gán: X & operator = (const X &)

**17.5.1.1 Beware of Default Constructors**

Khi viết một thao tác sao chép, hãy đảm bảo sao chép mọi cơ sở và thành viên.

**17.5.1.2 Copy of Bases**

Đối với mục đích sao chép, một cơ sở chỉ là một thành viên: để sao chép một đối tượng của một lớp dẫn xuất mà bạn có để sao chép các cơ sở của nó.

**17.5.1.3 The Meaning of Copy**

Copy-on-write không phải là thuốc chữa bách bệnh, nhưng nó có thể là sự kết hợp hiệu quả của

tính đơn giản của sao chép đúng và hiệu quả của sao chép.

**17.5.1.4 Slicing**

**17.5.2 Move**

Đối với xử lý tài nguyên, các hoạt động di chuyển có xu hướng đơn giản và hiệu quả hơn đáng kể so với các thao tác sao chép.

Có các hoạt động di chuyển ảnh hưởng đến thành ngữ để trả về các đối tượng lớn từ các hàm.

Việc sử dụng new bên trong toán tử + () không rõ ràng và buộc người dùng + phải đối phó với bộ nhớ phức tạp vấn đề quản lý:

• Làm thế nào để đối tượng được tạo bởi new bị xóa?

• Chúng ta có cần người thu gom rác không?

• Chúng ta có nên sử dụng một nhóm Ma trận thay vì mới chung chung không?

• Chúng ta có cần các biểu diễn Ma trận được tính sử dụng không?

• Chúng ta có nên thiết kế lại giao diện của phần bổ sung Ma trận của chúng ta không?

• Người gọi toán tử + () có nhớ xóa kết quả không?

• Điều gì xảy ra với bộ nhớ mới được cấp phát nếu tính toán ném ra một ngoại lệ?

**17.6 Generating Default Operations**

Một lớp cung cấp:

• Một hàm tạo mặc định: X ()

• Hàm tạo bản sao: X (const X &)

• Phép gán bản sao: X & operator = (const X &)

• Một hàm tạo di chuyển: X (X &&)

• Chuyển nhượng nhiệm vụ: X & operator = (X &&)

• Một hàm hủy: ̃X ()

Nếu người lập trình khai báo bất kỳ hàm tạo nào cho một lớp, thì hàm tạo mặc định không được tạo cho lớp đó.

Nếu người lập trình khai báo một thao tác sao chép, một thao tác di chuyển hoặc một hàm hủy cho một lớp, không có thao tác sao chép, thao tác di chuyển hoặc trình hủy nào được tạo cho lớp đó.

**17.6.1 Explicit Defaults**

**17.6.2 Default Operations**

Ý nghĩa mặc định của mỗi hoạt động được tạo, như được triển khai khi trình biên dịch tạo ra nó, là áp dụng hoạt động cho mỗi thành viên dữ liệu cơ sở và không tĩnh của lớp.

Lưu ý rằng giá trị của một đối tượng được chuyển đến của một kiểu dựng sẵn là không thay đổi.

Trạng thái di chuyển từ mặc định là trạng thái mà trình hủy mặc định và gán bản sao mặc định hoạt động chính xác.

**17.6.3 Using Default Operations**

**17.6.3.1 Default Constructors**

Nếu chúng ta cũng muốn có hàm tạo mặc định, chúng ta có thể xác định một hoặc khai báo rằng chúng ta muốn hàm tạo mặc định do trình biên dịch tạo ra.

**17.6.3.2 Maintaining Invariants**

Các quy tắc để tạo ra các hoạt động mặc định là:

[1] Thiết lập một bất biến trong một phương thức khởi tạo

[2] Duy trì tính bất biến với các thao tác sao chép và di chuyển

[3] Thực hiện mọi thao tác dọn dẹp cần thiết trong trình hủy

**17.6.3.3 Resource Invariants**

Việc tạo ra các hoạt động sao chép chỉ không được chấp nhận, không bị cấm, vì vậy nếu bạn bỏ qua các cảnh báo, bạn có thể lấy ví dụ này qua trình biên dịch.

**17.6.3.4 Partially Specified Invariants**

**17.6.4 deleted Functions**

Chúng ta có thể ‘‘ xóa ’’ một hàm.

**17.7 Advice**

* Thiết kế hàm tạo, phép gán và hàm hủy dưới dạng một tập hợp các thao tác
* Sử dụng một hàm tạo để thiết lập một bất biến cho một lớp
* Nếu một hàm tạo có được một tài nguyên, thì lớp của nó cần một hàm hủy để giải phóng tài nguyên đó
* Nếu một lớp có một hàm ảo, nó cần một hàm hủy ảo
* Nếu một lớp không có hàm tạo, nó có thể được khởi tạo bằng cách khởi tạo thành viên
* Ưu tiên khởi tạo {} hơn khởi tạo = và ()
* Cung cấp cho một lớp một hàm tạo mặc định nếu và chỉ khi có giá trị mặc định là ‘‘natural’’
* Nếu một lớp là một vùng chứa, hãy cung cấp cho nó một phương thức khởi tạo danh sách khởi tạo
* Khởi tạo các thành viên và cơ sở theo thứ tự khai báo của chúng
* Nếu một lớp có một thành viên tham chiếu, nó có thể cần các hoạt động sao chép
* Thích khởi tạo thành viên hơn là gán trong một phương thức khởi tạo
* Sử dụng bộ khởi tạo trong lớp để cung cấp các giá trị mặc định
* Nếu một lớp là một xử lý tài nguyên, nó có thể cần các hoạt động sao chép và di chuyển
* Khi viết hàm tạo bản sao, hãy cẩn thận sao chép mọi phần tử cần sao chép
* Hoạt động sao chép phải cung cấp tính tương đương và độc lập
* Cẩn thận với các cấu trúc dữ liệu vướng víu
* Nếu một lớp cần một hoạt động sao chép hoặc một hàm hủy, nó có thể cần một hàm tạo, một hàm hủy, một phép gán bản sao và một phương thức tạo bản sao
* Nếu một lớp có một thành viên con trỏ, nó có thể cần một hàm hủy
* Nếu một lớp là một xử lý tài nguyên, thì nó cần một hàm tạo, một hàm hủy
* Nếu một hàm tạo, phép gán hoặc hàm hủy mặc định là phù hợp, hãy để trình biên dịch tạo ra nó
* Nói rõ ràng về những bất biến của bạn; sử dụng các hàm tạo để thiết lập chúng và gán cho duy trì chúng
* Đảm bảo rằng các bài tập sao chép là an toàn
* Khi thêm một thành viên mới vào một lớp, hãy kiểm tra xem liệu có các hàm tạo do người dùng xác định cần được cập nhật để khởi tạo thành viên

## Chương 18. Nạp chồng toán tử

18.1 Giới thiệu

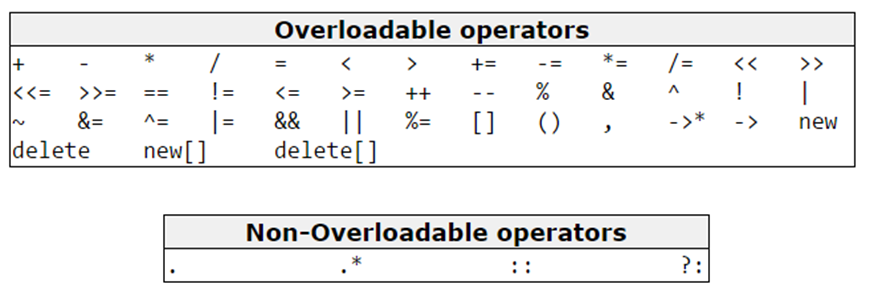
Nạp chồng toán tử (Operator Overloading) được dùng để định nghĩa toán tử cho có sẵn trong c++ phục vụ cho dữ liệu riêng do bạn tạo ra.  
Giả sử có lớp PhanSo và có các phương thức tính toán như Cong, Tru, Nhan, Chia.Nếu gặp một biểu thức phức tạp, số lượng phép tính nhiều thì việc sử dụng các phương thức trên khá khó khăn và có thể gây rối cho người lập trình. Vì thế ta sẽ nạp chồng lại các toán tử để có thể tạo một cái nhìn trực quan vào code, giảm thiểu các lỗi sai không đáng có.

### **Các loại toán tử**

* C++ chỉ cho phép người dùng overloading lại các toán tử có sẵn trong c++
* Một toán tử có thể được định nghĩa cho nhiều kiểu dữ liệu khác nhau.

#### 18.2

Các toán tử cho phép chồng hàm và không cho phép nạp chồng:



#### 18.2.1 Toán tử đơn, toán tử đôi

Toán tử đơn

Toán tử đơn là toán tử một ngôi (unary operator), có thể được dùng làm toán tử trước (prefix operator) và toán tử sau (postfix operator). Ví dụ phép tăng (++) hay phép giảm (–-)

**Ví dụ:**

* prefix operator: ++i;
* postfix operator: i++;

#### Toán tử đôi

Toán tử đôi là toán tử có 2 ngôi (binary operator).

**Ví dụ:** như A+B, A\*B, hay toán tử chỉ mục […] cũng là toán tử đôi.

18.2.2 Toán tử có sẵn- được định nghĩa sẵn(Predefined Meanings for Operators)

18.2.3 Toán tử do người dung tự định nghĩa(Operators and User-Defined Types)

18.2.4 Truyền đối tượng

Đối với các đối số có hai lựa chọn:

• Pass-by-value(truyền tham trị): Dùng cho các đối tượng nhỏ, sử dụng và thay đổi giá trị đối số

void Point::operator+=(*Point* *delta*); // pass-by-value

Để trả về tham trị:

*Matrix* operator+(const *Matrix*& *a*, const *Matrix*& *b*) // return-by-value

{

    Matrix res {*a*};

    return res+=*b*;

}

• Pass-by-reference(truyền tham chiếu): không thay đổi giá trị đối số

*Matrix* operator+(const *Matrix*&, const *Matrix*&); // pass-by-const-reference

Để trả về tham chiếu:

*Matrix*& Matrix::operator+=(const *Matrix*& *a*) // return-by-reference

{

    if (dim[0]!=a.dim[0] || dim[1]!=a.dim[1])

        throw std::exception("bad Matrix += argument");

    double∗ p = elem;

    double∗ q = a.elem;

    double∗ end = p+dim[0]∗dim[1];

    while(p!=end)

        ∗p++ += ∗q++

    return ∗*this*;

}

18.3 Kiểu số thực

18.3.1 Hàm thành viên và hàm không thành viên

Nếu bạn định nghĩa hàm nạp chồng toán tử của mình là hàm thành viên, thì trình biên dịch sẽ dịch các biểu thức như s1 + s2thành s1.operator+(s2). **Điều đó có nghĩa là, hàm thành viên được nạp chồng toán tử được gọi trên toán hạng đầu tiên**

Hàm thành viên hoạt động trên bất kỳ đối tượng nào của lớp mà nó là một thành viên, và có sự truy cập tới tất cả thành viên của một lớp cho đối tượng đó. Ví dụ:

class complex {

    double re, im;

public:

*complex*& operator+=(*complex* *a*); // needs access to representation

    // ...

};

*complex* operator+(*complex* *a*, *complex* *b*)

{

    return a += b; // access representation through +=

}

18.4 Ép kiểu

Để ngăn chặn các chuyển đổi ngầm định ngoài ý muốn được áp dụng, bạn có thể xác định toán tử chuyển đổi rõ ràng bằng cách định tính hàm chuyển đổi với công cụ gọi là explicit vào để chỉ định hàm:

template <typename *T*, typename *D* = default\_delete<*T*>>

class unique\_ptr {

public:

    // ...

    explicit operator bool() const noexcept; // does \*this hold a pointer (that is not nullptr)?

    // ...

};

## Chương 19 : Toán tử đặc biệt

19.1 Giới thiệu

19.2 Toán tử đặc biệt

- Các toán tử đặc biệt

[]        ()         −>       ++      −−      new     delete

19.2.1 Ghi đè toán tử

Để ghi đè 1 phép toán, cần đặt tên hàm gồm từ khóa operator rồi tiếp ngay sau đấy là toán tử của phép toán muốn ghi đè, ví dụ ghi đè [] : operator[].

struct *Assoc* {

    vector<pair<string,int>> vec; // vector of {name,value} pairs

    const int& operator[] (const *string*&) const;

    int& operator[](const *string*&);

};

int& Assoc::operator[](const *string*& *s*)

    // search for s; return a reference to its value if found;

    // otherwise, make a new pair {s,0} and return a reference to its value

{

    for (auto x : vec)

        if (s == x.first) return x.second;

    vec.push\_back({s,0}); // initial value: 0

    return vec.back().second; // return last element

}

19.2.2 Gọi hàm

Các hàm thường giao tiếp hay gọi đến nhau bằng lời gọi hàm( function call).

struct *Action* {

    int operator()(int);

*pair*<int,int> operator()(int,int);

    double operator()(double);

    // ...

};

void f(*Action* *act*)

{

    int x = act(2);//call funntion

    auto y = act(3,4); //call funntion

    double z = act(2.3); //call funntion

    // ...

};

Việc giao tiếp hay gọi đến nhau của các hàm thông qua truyền tham số. Các tham số được truyền vào theo 2 cách: truyền bằng tham trị hoặc bằng tham chiếu.

19.2.4: Toán tử tang và giảm

Toán tử tăng, giảm (kí hiệu: ++,**--**) có thể làm tiền tố (đứng trước) hoặc hậu tố (đứng sau) một biến (variable).

++x hoặc –x: tăng hoặc giảm x sau đó trả về x.

x++ hoặc x--: sao chép x, sau đó tăng hoặc giảm x, sau đó trả lại bản sao.

Ví dụ:

void f1(*X* *a*) // traditional use

{

    X v[200];

    X∗ p = &v[0];

    p−−;

    ∗p = a; // oops: p out of range, uncaught

    ++p;

    ∗p = a; // OK

}

19.2.5 Toán tử new và delete

Toán tử new để cấp phát bộ nhớ động cho bất kỳ kiểu dữ liệu nào trong C/C++.

Toán tử delete dung để giải phóng bộ nhớ đã cấp phát không còn cần thiết

void∗ operator new(*siz* *e\_t*); // use for individual object

void∗ operator new[](*siz* *e\_t*); // use for array

void operator delete(void∗, *siz* *e\_t*); // use for individual object

void operator delete[](void∗, *siz* *e\_t*); // use for array

19.3 Lớp string

19.3.1 Cách hoạt động

Trong C++, bạn có thể tạo ra một **đối tượng string** để lưu trữ chuỗi ký tự. Không giống mảng ký tự, đối tượng string không có kích thước cố định và có thể mở rộng nếu cần.

Đối tượng string được tạo bởi lớp string trong thư viện **#include <string>**.

#### **Cú pháp khai báo đối tượng string:**

**string** **<tên đối tượng string>**;

19.3.2 Quyền truy cập vào kí tự

Có thể gán một hằng **string** hoặc các **string**khác vào một biến string.

string str1 = "I love";

string str2 = str1;

Có thể gán hai chuỗi **string**, nối chuỗi bằng toán tử “+”, so sánh hai chuỗi bằng các **toán tử quan hệ (<, >, ==, !=,…)**. Một **string** với chiều dài **n**, những vị trí của các ký tự của **string** có phạm vi từ 0 tới n – 1. Có thể truy xuất ký tự trong **string** như mảng với **cú pháp**: **<tên chuỗi> [vị trí]**.

string str1 = "Apple"; string str2 = "apple";

string str3 = "apples"; string str4 = "orange";

bool t1 = (str1 == str2); // t1 = false

bool t2 = (str1 < str2); //t2 = true

bool t3 = (str2 < str3); //t3 = true

bool t4 = (str3 != str4); //t4 = true

bool t5 = (str4 > str3); // t5 = true

string myMsg = "How are you doing?";

cout<<myMsg[4]<<" "<<myMsg[12]<<myMsg[13]<<myMsg[16]<<endl; // a dog

};

19.3.3: Cách biểu diễn.

Quyền truy cập vào kí tự.19.3.4: Hàm thành viên:

Các hàm thành viên có thể được định nghĩa bên trong định nghĩa lớp hoặc sử dụng **toán tử phân giải phạm vi ::**.

String::String() // default constructor : x{""} = hàm tạo mặc định

        : sz{0}, ptr{ch} // ptr points to elements, ch is an initial location (§19.3.3)

    {

        ch[0] = 0; // terminating 0

    }

String::String(const char∗ p)

    :sz{strlen(p)},

    ptr{(sz<=short\_max) ? ch : **new** char[sz+1]},

    space{0}

{

    strcpy(ptr,p); // copy characters into ptr from p

}

19.3.5: Hàm trợ giúp.

Các hàm trợ giúp là (điều mà tôi tin rằng hầu hết mọi người có nghĩa là khi họ nói) thường là các hàm bao hàm một số chức năng hữu ích mà bạn sẽ sử dụng lại, rất có thể lặp đi lặp lại. Bạn có thể tạo các hàm trợ giúp có nghĩa là được sử dụng cho nhiều mục đích khác nhau ...

*ostream*& operator<<(*ostream*& *os*, const *String*& *s*)

{

    return os << s.c\_str(); // §36.3.3

}

*istream*& operator>>(*istream*& *is*, *String*& *s*)

{

    s = ""; // clear the target string

    is>>ws; // skip whitespace (§38.4.5.1)

    char ch = ' ';

    while(is.get(ch) && !isspace(ch))

        s += ch;

    return is;

}

bool operator==(const *String*& *a*, const *String*& *b*)

{

    if (a.size()!=b.siz e())

        return false;

    for (int i = 0; i!=a.size(); ++i)

        if (a[i]!=b[i])

            return false;

    return true;

}

bool operator!=(const *String*& *a*, const *String*& *b*)

{

    return !(a==b);

}

19.4 Hàm bạn

Hàm bạn của một lớp không phải là hàm thành viên nên nó không phụ thuộc vào lớp và có thể định nghĩa ở trong hoặc ngoài lớp.

Hàm bạn có thể truy cập trực tiếp các thành viên private và một lớp có thể có nhiều hàm bạn

Cú pháp khai báo hàm bạn đơn giản là thêm từ khóa friend trước hàm bình thường ta vẫn sử dụng:

Ví dụ :

constexpr rc\_max {4}; // row and column size

class Matrix;

class Vector {

    float v[rc\_max];

    // ...

    friend Vector operator∗(const Matrix&, const Vector&);

};

class Matrix {

    Vector v[rc\_max];

    // ...

    friend Vector operator∗(const Matrix&, const Vector&);

};

19.4.2 Hàm bạn và hàm thành viên

class X {

    // ...

    X(int);

    int m1(); // member = hàm thành viên

    int m2() const;

    friend int f1(*X*&); // friend, not member = hàm bạn, không phải hàm thành viên

    friend int f2(const *X*&);

    friend int f3(*X*);

};

* Hàm bạn của một lớp không phải là hàm thành viên nên nó không phụ thuộc vào lớp và có thể định nghĩa ở trong hoặc ngoài lớp. Hàm bạn có thể truy cập trực tiếp các thành viên private và một lớp có thể có nhiều hàm bạn

Nên dùng hàm bạn để nạp chồng toán tử vì:

* Toán tử nạp chồng là hàm bình thường, không phải là hàm thành viên (phương thức) nên chúng truy cập dữ liệu thông qua hàm truy cập và hàm biến đổi nên nó rất kém hiệu quả (do mất phụ phí lời gọi)
* Hàm bạn có thể truy cập trực tiếp dữ liệu từ biến private nên nó không mất phụ phí -> Cải thiện hiệu quả thực hiện, tránh gọi hàm thành viên truy cập || biến đổi.

## Chương 20 : Các lớp dẫn xuất

**20.2 Các lớp có nguồn gốc**

Cân nhắc xây dựng một chương trình giao dịch với những người được một công ty tuyển dụng. Một chương trình như vậy có thể có một cấu trúc dữ liệu như thế này:

Xây dựng strutc nhân viên :

struct Employee {

string first\_name , family\_name;

char middle\_initial;

Date hiring\_date;

short depar tment;

// ...

};

Xây dựng strutc quản lí:

struct Manager {

Employee emp; // hồ sơ nhân viên quản lí

list<Employee∗> group; // người quản lí

short level;

// ...

};

Nhược điểm của chương trình trên là không có gì cho trình biên dịch và các công cụ khác biết rằng Người quản lý cũng là Nhân viên. Người quản lý ∗ không phải là Nhân viên ∗ .Cách tiếp cận đúng là trình bày rõ ràng rằng Người quản lý là Nhân viên, với một số thông tin được thêm

Quản lý lớp có các thành viên của lớp Nhân viên (tên\_người đầu tiên, phòng ban, v.v.) ngoài các thành viên của lớp

void f(Manager m1, Employee e1)

{

list<Employee∗> elist {&m1,&e1);

// ...

}

Người quản lý (cũng) là một Nhân viên, vì vậy Người quản lý ∗ có thể được sử dụng như một Nhân viên ∗. Tuy nhiên, một Nhân viên không nhất thiết phải là một Người quản lý , vì vậy một Nhân viên ∗ không thể được sử dụng làm Người quản lý ∗. Nói chung, nếu một lớp Derived có một lớp cơ sở công khai

Cơ sở, sau đó một Xuất phát ∗ có thể được gán cho một biến kiểu Cơ sở ∗ mà không cần sử dụng chuyển đổi kiểu rõ ràng. Việc chuyển đổi ngược lại, từ Cơ sở ∗ sang Bắt nguồn ∗, phải rõ ràng. Ví dụ:

void g(Manager mm, Employee ee)

{

Employee∗ pe = &mm; // mọi người quản lí đều là nhân viên

Manager∗ pm = &ee; //  không phải mọi nhân viên dều là quản lí

pm−>level = 2; // lỗi : ee hông có cấp độ

pm = static\_cast<Manager∗>(pe); // điểm hoạt động pe

                                                         // cho người quản lí mm

pm−>level = 2; //  pm trỏ tới Trình quản lý mm có cấp độ

}

Nói cách khác, một đối tượng của một lớp dẫn xuất có thể được coi là một đối tượng của lớp cơ sở của nó khi được thao tác thông qua các con trỏ và tham chiếu.

class Employee;                                           // chỉ khai báo, không có định nghĩa

class Manager : public Employee {        //lỗi : Nhân viên không được xác định

// ...

};

**20.2.1 chức năng thành viên**

Cung cấp một kiểu thích hợp với một tập hợp các thao tác phù hợp và chúng ta cần làm như vậy mà không bị ràng buộc vào các chi tiết của một biểu diễn cụ thể. Ví dụ:

class Employee {

public:

void print() const;

string full\_name() const { return first\_name + ' ' + middle\_initial + ' ' + family\_name; }

// ...

private:

string first\_name , family\_name;

char middle\_initial;

// ...

};

class Manager : public Employee {

public:

void print() const;

// ...

};

Thành viên của một lớp dẫn xuất có thể sử dụng công khai - các thành viên của một cơ sở lớp như thể chúng được khai báo trong chính lớp dẫn xuất. Ví dụ:

void Manager::print() const{

cout << "name is " << full\_name() << '\n';

// ...

}

Tuy nhiên, một lớp dẫn xuất không thể truy cập các thành viên riêng của một lớp cơ sở

Thông thường, giải pháp sạch nhất là dành cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của

lớp cơ sở:

void Manager::print() const

{

Employee::print(); //in thông tin nhân viên

cout << level; //  in thông tin cụ thể về trình quản lí

// ...}

20.2.2 Hàm tạo và hàm hủy

Như thường lệ, hàm tạo và hàm hủy là thiết yếu:

• Các đối tượng được xây dựng từ dưới lên (cơ sở trước thành viên và thành viên trước

dẫn xuất) và hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở);

• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó (nhưng không trực tiếp là thành viên hoặc cơ sở của

căn cứ)

• Thông thường, các trình hủy trong một hệ thống phân cấp cần phải là ảo

• Các hàm tạo sao chép của các lớp trong một hệ thống phân cấp nên được sử dụng cẩn thận (nếu có) để tránh bị cắt

• Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast hoặc một typeid () trong một phương thức khởi tạo hoặc de-structor phản ánh giai đoạn xây dựng và phá hủy (chứ không phải là kiểu của đối tượng chưa được hoàn thành)

20.3 Cấu trúc phân cấp lớp

Bản thân một lớp dẫn xuất có thể là một lớp cơ sở.

class Employee { /\* ... \*/ };

class Manager : public Employee { /\* ... \*/ };

class Director : public Manager { /\* ... \*/ };

20.3.1 Nhập trường

Để sử dụng các lớp dẫn xuất không chỉ là một cách viết tắt thuận tiện trong khai báo. Có bốn giải pháp cơ bản:

1.Đảm bảo rằng chỉ các đối tượng của một kiểu duy nhất được trỏ tới

2.Đặt một trường kiểu trong lớp cơ sở để các chức năng kiểm tra

3.Sử dụng dynamic\_cast

4.Sử dụng các hàm ảo

**20.2.3 Chức năng ảo**

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình viên khai báo các hàm trong một lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. Trình biên dịch và trình liên kết sẽ đảm bảo sự tương ứng chính xác giữa các đối tượng và các chức năng được áp dụng cho chúng.

20.3.3 Ép kiểu tường minh

Gọi một hàm bằng toán tử phân giải phạm vi, ::, như được thực hiện trong Manager :: print () đảm bảo rằng cơ chế ảo không được sử dụng:

void Manager::print() const

{

Employee::print(); // not a virtual call

cout << "\tlevel " << level << '\n';

// ...

}

Nếu không, Manager :: print () sẽ phải chịu một đệ quy vô hạn. Việc sử dụng một tên đủ điều kiện có một hiệu quả mong muốn khác. Nghĩa là, nếu một hàm ảo cũng nội tuyến (không phải là hiếm), thì phép thay thế nội tuyến có thể được sử dụng cho các lệnh gọi được chỉ định bằng cách sử dụng ::. Điều này cung cấp cho người lập trình một cách hiệu quả để xử lý một số trường hợp đặc biệt quan trọng trong đó một hàm ảo gọi một hàm ảo khác cho cùng một đối tượng. Hàm Manager :: print () là một ví dụ về điều này. Bởi vì kiểu của đối tượng được xác định trong lệnh gọi Manager :: print (), nó không cần được xác định động lại cho lệnh gọi kết quả của Employee :: print ().

**20.3.4 Ghi đè hàm**

Nếu bạn khai báo một hàm trong lớp dẫn xuất có cùng tên và kiểu với hàm ảo trong lớp cơ sở, thì hàm trong lớp dẫn xuất sẽ ghi đè hàm trong lớp cơ sở.

Đó là một quy tắc đơn giản và hiệu quả. Tuy nhiên, đối với các cấu trúc phân cấp lớp lớn hơn, có thể khó để chắc chắn rằng bạn thực sự ghi đè chức năng mà bạn định ghi đè

20.3.4.1 từ khóa override

Override là việc viết lại một phương thức (method) trong lớp dẫn xuất (derived class) mà ở lớp cơ sở (base class).

Như chúng ta đã biết để lớp dẫn suất override 1 virtual function của base class thì 2 hàm phải có cùng tên, cùng tham số và cùng kiểu dữ liệu trả về. Tuy nhiên trong nhiều trường hợp ta có thể xảy ra nhầm lẫn khi override 1 virtual function.

**20.3.4.2 final**

từ khóa ***final*** khi dùng cho biến trong một phương thức, cũng có ý nghĩa tương tự như khi dùng từ khóa này với thuộc tính của lớp. Nó đều mang giá trị là một hằng số không thể thay đổi được.

**20.3.5 Sử dụng thành viên cơ sở**

Khi bạn khai báo một lớp dẫn xuất, một chỉ định truy cập có thể đứng trước mỗi lớp cơ sở trong danh sách cơ sở của lớp dẫn xuất. Điều này không làm thay đổi các thuộc tính truy cập của các thành viên riêng lẻ của lớp cơ sở như được thấy bởi lớp cơ sở, nhưng cho phép lớp dẫn xuất hạn chế quyền kiểm soát truy cập của các thành viên của lớp cơ sở.

Bạn có thể lấy các lớp bằng cách sử dụng bất kỳ ba mã truy cập nào:

Trong một publiclớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở vẫn là các thành viên công khai và được bảo vệ của lớp dẫn xuất.

Trong một protectedlớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở là các thành viên được bảo vệ của lớp dẫn xuất.

Trong một privatelớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở trở thành các thành viên riêng của lớp dẫn xuất.

Trong mọi trường hợp, các thành viên private của lớp cơ sở vẫn là private. Các thành viên riêng của lớp cơ sở không thể được sử dụng bởi lớp dẫn xuất trừ khi khai báo bạn bè trong lớp cơ sở cấp quyền truy cập rõ ràng cho chúng.

**20.3.5.1 Trình tạo kế thừa**

Khi kế thừa từ một lớp, nếu bạn cần kế thừa các hàm tạo của nó mà không cần khởi tạo thêm, bạn nên using-declarationkế thừa tất cả các hàm tạo của lớp cơ sở thay vì viết chúng bằng tay.

using-declaration đối với hàm tạo kế thừa là một tính năng C ++ 11 làm cho tất cả các hàm tạo của cơ sở hiển thị với độ phân giải quá tải khi khởi tạo lớp dẫn xuất.

Nếu bạn cần thay đổi khả năng truy cập của một trong các hàm tạo được kế thừa, bạn có thể làm điều đó bằng cách giữ using-declarationvà khai báo hàm tạo đó một cách rõ ràng là private

20.4 Các lớp trừu tượng

Các lớp trừu tượng đóng vai trò là biểu thức của các khái niệm chung mà từ đó các lớp cụ thể hơn có thể được dẫn xuất. Bạn không thể tạo một đối tượng thuộc loại lớp trừu tượng. Tuy nhiên, bạn có thể sử dụng con trỏ và tham chiếu đến các kiểu lớp trừu tượng.

Bạn tạo một lớp trừu tượng bằng cách khai báo ít nhất một hàm thành viên ảo thuần túy. Đó là một hàm ảo được khai báo bằng cách sử dụng cú pháp specifier ( ) thuần túy= 0 . Các lớp dẫn xuất từ lớp trừu tượng phải thực hiện hàm ảo thuần túy hoặc chúng cũng là các lớp trừu tượng.

**20.5 Kiểm soát truy cập**

Trong một lớp được sử dụng để đặt khả năng truy cập của các thành viên lớp. Đó là, nó đặt ra một số hạn chế đối với các thành viên lớp không được truy cập trực tiếp bởi các hàm bên ngoài.

Trong C++ có hổ trợ 3 loại access modifier đó là:

* Private
* Public
* Protected

Lưu ý: Nếu chúng ta không chỉ rõ bất kỳ access modifier cho thành viên của lớp, thì mặc định nó là Private.

**20.5.1 Protected**

Protected thường được dùng khi bạn biết chắc là có lớp khác sẽ kế thừa lớp này và những phương thức, thuộc tính đó chỉ được dùng trong lớp kế thừa nó.

Giả sử bạn khai báo lớp Động Vật, trong đó có hàm lưu dữ liệu động vật vào database, hàm này dùng chung cho tất cả các lớp con kế thừa lớp Động Vật. Và để bảo mật nên tôi không muốn ở bên ngoài lớp có thể sử dụng được, vì vậy tôi khai báo là protected.

Đó là những ví dụ cơ bản, chứ thực tế thì cũng tùy vào từng bài toán cụ thể mà bạn lựa chọn

**20.5.2 Quyền truy cập vào các lớp cơ sở**

Cách mà chúng ta có thể lấy được các lớp gọi là các chế độ truy cập. Các chế độ truy cập này có tác dụng:

* Public: nếu một lớp con được khai báo ở chế độ public thì các thành phần của lớp cơ sở được kế thừa bởi lớp con giống như chúng.
* Private: trong trường hợp này, tất cả các thành phần của lớp cơ sở đều trở thành private trong lớp con.
* Protected: các thành phần public của lớp cơ sở trở thành protected trong lớp con.

Các thành phần private của lớp cơ sở luôn là private trong lớp con.

**20.6 Trỏ tới thành viên**

Con trỏ đến các thành viên cho phép bạn tham chiếu đến các thành viên không cố định của các đối tượng lớp. Bạn không thể sử dụng con trỏ tới thành viên để trỏ đến thành viên lớp tĩnh vì địa chỉ của thành viên tĩnh không được liên kết với bất kỳ đối tượng cụ thể nào. Để trỏ đến một thành viên lớp tĩnh, bạn phải sử dụng một con trỏ bình thường.

## Chương 21: Phân cấp lớp

**21.1 Giới thiệu**

**21.2 Thiết kế cấu trúc phân cấp lớp**

Hãy xem xét một vấn đề thiết kế đơn giản: Cung cấp một cách để một chương trình (‘‘ một ứng dụng ’’) có được một tương tác giá trị mầm từ người dùng. Điều này có thể được thực hiện theo nhiều cách khác nhau. Để cách ly chương trình của chúng tôi từ sự đa dạng này và cũng để có cơ hội khám phá các lựa chọn thiết kế khả thi, chúng ta hãy bắt đầu bằng cách xác định mô hình chương trình của chúng tôi về thao tác nhập đơn giản này.

**21.2.1 Triển khai kế thừa**

Một lớp có thể được kế thừa từ hơn một lớp khác, nghĩa là, nó có thể kế thừa dữ liệu và hàm từ nhiều lớp cơ sở. Để định nghĩa một lớp kế thừa (Derived Class), chúng ta sử dụng một danh sách để xác định các lớp cơ sở. Danh sách này liệt kê một hoặc nhiều lớp cơ sở và có form sau:

class lop\_ke\_thua: access\_modifier lop\_co\_so

ví dụ :

#include <iostream>

using namespace std;

// lop co so: Hinh

class Hinh

{

    public:

    void setChieuRong(int rong)

    {

        chieurong = rong;

    }

    void setChieuCao(int cao)

    {

        chieucao = cao;

    }

    protected:

    int chieurong;

    int chieucao;

};

// day la lop ke thua: HinhChuNhat

class HinhChuNhat:

public Hinh

{

    public:

    int tinhDienTich()

    {

        return chieurong \* chieucao;

    }

};

int main(void)

{

    HinhChuNhat Hcn;

    Hcn.setChieuRong(14);

    Hcn.setChieuCao(30);

    // in dien tich cua doi tuong.

    cout << "Tong dien tich la: " << Hcn.tinhDienTich() << endl;

    return 0;

}

**21.2.1.1 Phê bình**

Thiết kế này hoạt động tốt theo nhiều cách, và đối với nhiều vấn đề, kiểu phân cấp này là một giải pháp tốt . Tuy nhiên, có một số chi tiết khó xử có thể khiến chúng tôi phải tìm kiếm các thiết kế thay thế.

**21.2.2 Kế thừa giao diện**

Một cách sử dụng đa kế thừa không gây tranh cãi liên quan đến *kế thừa giao diện* . Trong C ++, tất cả kế thừa là kế thừa *thực thi* , bởi vì mọi thứ trong lớp cơ sở, giao diện và sự thực thi, đều trở thành một phần của lớp dẫn xuất. Không thể chỉ kế thừa một phần của một lớp (chẳng hạn như giao diện một mình). Như Chương 14 của Tập 1 đã giải thích, kế thừa riêng và được bảo vệ có thể hạn chế quyền truy cập vào các thành viên được kế thừa từ các lớp cơ sở khi được sử dụng bởi các máy khách của một đối tượng lớp dẫn xuất, nhưng điều này không ảnh hưởng đến lớp dẫn xuất; nó vẫn chứa tất cả dữ liệu lớp cơ sở và có thể truy cập tất cả các thành viên lớp cơ sở không riêng tư.

**21.3 Đa kế thừa**

Multiple Inheritance là một tính năng của C ++ trong đó một lớp có thể kế thừa từ nhiều hơn một lớp.

Các hàm tạo của các lớp kế thừa được gọi theo thứ tự mà chúng được kế thừa. Ví dụ, trong chương trình sau, phương thức khởi tạo của B được gọi trước phương thức khởi tạo của A.

Một lớp trong C++ có thể kế thừa các thành viên từ nhiều lớp, và đây là cú pháp:

class lop\_ke\_thua: access\_modifier lop\_co\_so\_1, access\_modifier lop\_co\_so\_2 ...

#include<iostream>

using namespace std;

class A

{

public:

  A()  { cout << "A's constructor called" << endl; }

};

class B

{

public:

  B()  { cout << "B's constructor called" << endl; }

};

class C: public B, public A  // Note the order

{

public:

  C()  { cout << "C's constructor called" << endl; }

};

int main()

{

    C c;

    return 0;

}

Đầu ra:

Hàm tạo của B được gọi là

Hàm tạo của A được gọi là

Hàm tạo của C được gọi là

Các hàm hủy được gọi theo thứ tự ngược lại của các hàm tạo.

-Truy cập các thành phần lớp trong đa kế thừa :  
Việc truy nhập đến các thành phần của các lớp trong đa kế thừa được dựa trên các nguyên tắc sau:

* Việc truy nhập từ đối tượng lớp dẫn xuất đến các thành phần của mỗi lớp cơ sở được tuân theo quy tắc phạm vi tương tự như trong đơn kế thừa.
* Trong trường hợp các lớp cơ sở đều có các thành phần cùng tên, việc truy xuất đến thành phần của lớp nào phải được chỉ rõ bằng toán tử phạm vi: “<Tên lớp>::” đối với thành phần lớp cơ sở đó.

# Phần 2 : Chương trình

1. **Giới thiệu đề tài.**

### **1.Lí do chọn đề tài**

* Nhóm muốn lập trình một chương trình đơn giản , áp dụng những kiến thức của ngôn ngữ C++ đã được học trên lớp và tìm hiểu tại nhà tạo ra một chương trình quản lí sự kiện giúp người dùng có thể tra cứu tất cả các sự kiện , ngày lễ trong năm và quan trọng hơn là có thể tạo ra những sự kiện riêng của bản thân mình , từ những công việc nhỏ như thời khóa biểu hằng ngày như ( đi gặp đối tác , họp nhóm , hẹn đi chơi , xem phim ...) đến những ngày quan trọng ( sinh nhật , kỉ niệm 100 ngày yêu nhau , kỉ niệm 1 năm ngày cưới ...).
* Mục đích : Giúp người sử dụng tạo ra 1 chuỗi sự kiện hợp lí , không bị quên hay bỏ qua những sự kiện , ngày lễ quan trọng.

### **2.Mô tả bài toán**

* Sử dụng lập trình hướng đối tượng , tạo ra 1 chương trình gồm 2 chức năng chính là xem ngày lễ và tạo sự kiện các thao tác của người dùng sẽ dược thiết kế dưới dạng MENU.

**+**  Xem ngày lễ sẽ gồm các chức năng như : Xem thời gian hiện tại ,

xem ngày lễ trong tháng.

+ Tạo sự kiện sẽ gồm các chức năng như : Thêm mới , hiển thị , sắp

xếp , tìm kiếm , xóa , lưu, xem danh sách đã lưu.

### **3.Phân công công việc và thực hiện**

* Phân công công việc :

+ Nguyễn Hồ Hưng :lên ý tưởng đề tài, code khung chương trình , upload tài liệu tham khảo ,fix bug chương trình.

+ Dương Văn Hoàng : Code phần giao diện Menu chương trình , code chức năng, fix bug chương trình .

+ Đặng Đình Sang: Làm báo cáo bài tập lớn , code chức năng ,

fix bug chương trình .

* Thưc hiện : Do dich bệnh Covid 19 , nhóm em rất ít khi gặp nhau trực tiếp nên nhóm thảo luận và đưa ra phương án làm việc thông qua kênh trực tuyến là GitHud , Zalo và Discod .

1. **Nội dung cơ sở lý thuyết**

### 1 .Tính chất hướng đối tượng

* Tính đóng gói(Encapsulation) : là cách dùng để che dấu những tính chất xử lí bên trong của đối tượng , những đối tượng khác không thể tác động trực tiếp làm thay đổi trạng thái chỉ có thể tác động thông qua các method pubic của đối tượng đó.

Ví dụ trong bài :

* using namespace std;
* class DateTime{
* private:
* int day;
* int month;
* int year;
* int hour;
* int minute;
* int second;
* int ngayDaQua=0;
* int thangDaQua=0;
* int timThang;
* public:
* DateTime(){
* time\_t now = time(0);
* tm \*ltm = localtime(&now);
* day = ltm->tm\_mday;
* month = 1 + ltm->tm\_mon;
* year = 1900 + ltm->tm\_year;
* hour = ltm->tm\_hour;
* minute = ltm->tm\_min;
* second = ltm->tm\_sec;
* }
* void thoiGianHienTai();
* void ngayLeTrongThang1();
* void ngayLeTrongThang2();
* void ngayLeTrongThang3();
* void ngayLeTrongThang4();
* void ngayLeTrongThang5();
* void ngayLeTrongThang6();
* void ngayLeTrongThang7();
* void ngayLeTrongThang8();
* void ngayLeTrongThang9();
* void ngayLeTrongThang10();
* void ngayLeTrongThang11();
* void ngayLeTrongThang12();
* void xemNgayLeTrongThang();
* void xemNgayLeTrongNam();
* void xacDinhThuTrongTuan();
* void menuNgayLe();
* void ListMenuNgayLe();
* };
* **Tính kế thừa (Inheritance):** Là kỹ thuật cho phép kế thừa lại những tính năng mà một đối tượng khác đã có, giúp tránh việc code lặp dư thừa mà chỉ xử lý công việc tương tự.

Ví dụ trong bài :

* class HolidayInMonth : public DateTime{
* public:
* void xemNgayLeTrongThang(){
* Frames();
* FramesInput();
* **Tính trừu tượng(Abstraction):** Là phương pháp trừu tượng hóa định nghĩa lên những hành động, tính chất của loại đối tượng nào đó cần phải có

+ Sử dụng các biến có tên liên quan đến sự kiện như tên ngày tháng năm để người tạo chương trình hay người xem code dễ hình dung biến đó đang chứa giá trị gì và làm gì:

Ví dụ trong bài :

* struct Event
* {
* string TieuDe;
* int ngay, thang, nam;
* int gio, phut;
* string NoiDung;
* };
* **Tính đa hình (Polymorphism ):** Là một đối tượng thuộc các lớp khác nhau có thể hiểu cùng một thông điệp theo cách khác nhau.

**2.Cấu trúc dữ liệu và giải thuật dùng trong bài toán**

* **Trong bài sử dụng các cấu trúc trúc dữ liệu và giải thuật là :**

**2.1 Danh sách liên kết đơn:**

+Danh sách liên kết đơn (Single Linked List) là một cấu trúc dữ liệu động, nó là một danh sách mà mỗi phần tử đều liên kết với phần tử đúng sau nó trong danh sách. Mỗi phần tử (được gọi là một node hay nút) trong danh sách liên kết đơn là một cấu trúc có hai thành phần:

+Thành phần dữ liệu: lưu thông tin về bản thân phần tử đó.

+Thành phần liên kết: lưu địa chỉ phần tử đứng sau trong danh sách, nếu phần tử đó là phần tử cuối cùng thì thành phần này bằng NULL.

**Ví dụ trong chương trình:**

* // Tạo cấu trúc danh sách LK đơn
* struct SingleList
* {
* Node \*pHead;
* };
* // Khởi tạo danh sách LK đơn
* void Initialize(SingleList \*&list)
* {
* list=new SingleList;
* list->pHead=NULL;
* }
* // Tạo một Node mới
* Node \*CreateNode(Event \*ev)
* {
* Node \*pNode=new Node;
* if(pNode!=NULL)
* {
* pNode->data=ev;
* pNode->pNext=NULL;
* }
* else
* {
* cout<<"cap phat bo nho that bai!!!";
* }
* return pNode;
* }
* // Thêm một sự kiện vào danh sách LK đơn
* Event \*NhapEvent()
* {
* Event \*ev=new Event;
* Frames();
* FramesInput();
* lineTitle();
* gotoxy(54, 12); cout << "ADD EVENT";
* fflush(stdin);
* gotoxy(45, 14); cout << "Tieu de: ";
* getline(cin, ev->TieuDe);
* settingDay(\*ev);
* settingHour(\*ev);
* gotoxy(58, 24); cout << " ";
* fflush(stdin);
* gotoxy(44, 26); cout << "Noi dung: ";
* getline(cin, ev->NoiDung);
* gotoxy(70, 29); cout << "Nhan enter de hoan tat!";
* getch();
* return ev;
* }
* // Thêm một sự kiện vào cuối danh sách LK đơn
* void InsertLast(SingleList \*&list,Event \*ev)
* {
* Node \*pNode=CreateNode(ev);
* if(list->pHead==NULL)
* {
* list->pHead=pNode;
* }
* else
* {
* Node \*pTmp=list->pHead;
* while(pTmp->pNext!=NULL)
* {
* pTmp=pTmp->pNext;
* }
* pTmp->pNext=pNode;
* }
* }

**2.2 Thuật toán sắp xếp nổi bọt (Bubble Sort) : sẽ tiến hành việc so sánh các cặp phần tử liền kề nhau và tráo đổi vị trí của nó cho đúng thứ tự mà người dùng mong muốn.**

**Ví dụ trong chương trình :**

* // Hàm sắp xếp theo thời gian (sử dụng thuật toán sắp xếp nổi bọt)
* void swap( Event \*&a, Event \*&b ){
* Event \*tmp=a;
* a=b;
* b=tmp;
* }
* void SortList1(SingleList \*&list) {
* Node \*pTmp = list->pHead;
* if(pTmp == NULL) {
* cout << "Danh sach rong" << endl;
* return;
* }
* while(pTmp->pNext != NULL) {
* for(Node \*q = pTmp->pNext ; q != NULL ; q = q->pNext) {
* if(pTmp->data->nam > q->data->nam) {
* swap(pTmp->data, q->data);
* }
* else if(pTmp->data->nam == q->data->nam) {
* if(pTmp->data->thang > q->data->thang) {
* swap(pTmp->data, q->data);
* }
* else if(pTmp->data->thang == q->data->thang) {
* if(pTmp->data->ngay > q->data->ngay) {
* swap(pTmp->data, q->data);
* }
* else if(pTmp->data->ngay == q->data->ngay) {
* if(pTmp->data->gio > q->data->gio) {
* swap(pTmp->data, q->data);
* }
* else if(pTmp->data->gio == q->data->gio) {
* if(pTmp->data->phut > q->data->phut) {
* swap(pTmp->data, q->data);
* }
* else if(pTmp->data->phut == q->data->phut) {
* if(pTmp->data->gio > q->data->gio) {
* swap(pTmp->data, q->data);
* }
* }
* }
* }
* }
* }
* }
* pTmp = pTmp->pNext;
* }
* }

**2.3 Thụât toán tìm kiếm nhị phân(Binary Search):**thực hiện tìm kiếm một mảng đã sắp xếp bằng cách liên tục chia các khoảng tìm kiếm thành 1 nửa. Bắt đầu với một khoảng từ phần tử đầu mảng, tới cuối mảng. Nếu giá trị của phần tử cần tìm nhỏ hơn giá trị của phần từ nằm ở giữa khoảng thì thu hẹp phạm vi tìm kiếm từ đầu mảng tới giửa mảng và nguợc lại. Cứ thế tiếp tục chia phạm vi thành các nửa cho đến khi tìm thấy hoặc đã duyệt hết.

**Ví dụ trong chương trình :**

* // Tìm kiếm theo tiêu đề (sử dụng thuật toán tìm kiếm nhị phân)
* void SearchByTitleEvent(SingleList \*list)
* {
* string title;
* Frames();
* FramesInput();
* gotoxy(54, 12);
* fflush(stdin);
* cout << "Tieu de: ";
* getline(cin, title);
* Node \*pTmp=list->pHead;
* if(pTmp->data->TieuDe == title)
* {
* Event \*ev=pTmp->data;
* gotoxy(44, 14);
* cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";
* gotoxy(44, 16);
* cout << "Tieu de: " << ev->TieuDe;
* gotoxy(44, 17);
* cout << "Thoi gian: " << ev->gio << ":" << ev->phut << ", " << ev->ngay << "/" << ev->thang << "/" << ev->nam;
* gotoxy(44, 18);
* cout << "Noi dung: " << ev->NoiDung;
* gotoxy(44, 19);
* cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";
* }
* else{
* modalLook();
* }
* pTmp=pTmp->pNext;
* }

**3.Đồ họa – Ví dụ**

* **Trong chương trình phần đồ họa và giao diện chủ yếu nhóm dùng những thư viện có sẵn để định dạng màn hình console , thay đổi màu ,chữ kẻ khung , xác định vị trí của các chức năng , viết tiêu đề và tên của chương trình , ngoài ra nhóm có tự code thêm một số chức năng như điều chình mũi tên để chọn chức năng .**

**3.1 Thư viện windows.h:**

**Ví dụ trong chương trình:**

**+Định dạng kích cỡ của cmd :**

* // hàm thay đổi kích cỡ của khung cmd với tham số truyền vào là chiều cao, chiều rộng.
* void resizeConsole(int width, int height)
* {
* HWND console = GetConsoleWindow();
* RECT r;
* GetWindowRect(console, &r);
* MoveWindow(console, r.left, r.top, width, height, TRUE);
* }

**+Hàm định dạng màu chữ và fontsize chữ của chương trình:**

* void SetColor(int backgound\_color, int text\_color)
* {
* HANDLE hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);
* int color\_code = backgound\_color \* 16 + text\_color;
* SetConsoleTextAttribute(hStdout, color\_code);
* }
* void setFontSize(int FontSize)
* {
* CONSOLE\_FONT\_INFOEX info = {0};
* info.cbSize = sizeof(info);
* info.dwFontSize.Y = FontSize; // leave X as zero
* info.FontWeight = FW\_NORMAL;
* SetCurrentConsoleFontEx(GetStdHandle(STD\_OUTPUT\_HANDLE), NULL, &info);
* }

**+Hàm viết tiêu đề EVENT của chương trình:**

* void e1(){
* // hàm viết chữ E
* gotoxy(57, 3);
* cout << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << "  " ;
* gotoxy(57, 4);
* cout << char(186) << char(186) << "         " ;
* gotoxy(57, 5);
* cout << char(186) << char(186) << "         " ;
* gotoxy(57, 6);
* cout << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) ;
* gotoxy(57, 7);
* cout << char(186) << char(186) << "         " ;
* gotoxy(57, 8);
* cout << char(186) << char(186) << "         " ;
* gotoxy(57, 9);
* cout << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) ;
* }
* .
* .
* .
* void t(){
* gotoxy(80, 3);
* cout << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186) << char(186);
* gotoxy(80, 4);
* cout << "     " << char(186) << char(186);
* gotoxy(80, 5);
* cout << "     " << char(186) << char(186);
* gotoxy(80, 6);
* cout << "     " << char(186) << char(186);
* gotoxy(80, 7);
* cout << "     " << char(186) << char(186);
* gotoxy(80, 8);
* cout << "     " << char(186) << char(186);
* gotoxy(80, 9);
* cout << "     " << char(186) << char(186);
* }
* void Text(){
* e(); v(); e1(); n(); t();
* }

**+ Khung chức năng của chương trình:**

* void FramesInput(){
* // khung input
* gotoxy(45, 11); cout << char(218);
* gotoxy(76, 11); cout << char(191);
* gotoxy(45, 12); cout << char(179);
* gotoxy(76, 12); cout << char(179);
* gotoxy(45, 13); cout << char(192);
* for(int i=46 ; i < 76 ; i++){
* gotoxy(i, 13);
* cout << char(196);
* }
* gotoxy(76, 13); cout << char(217);
* }

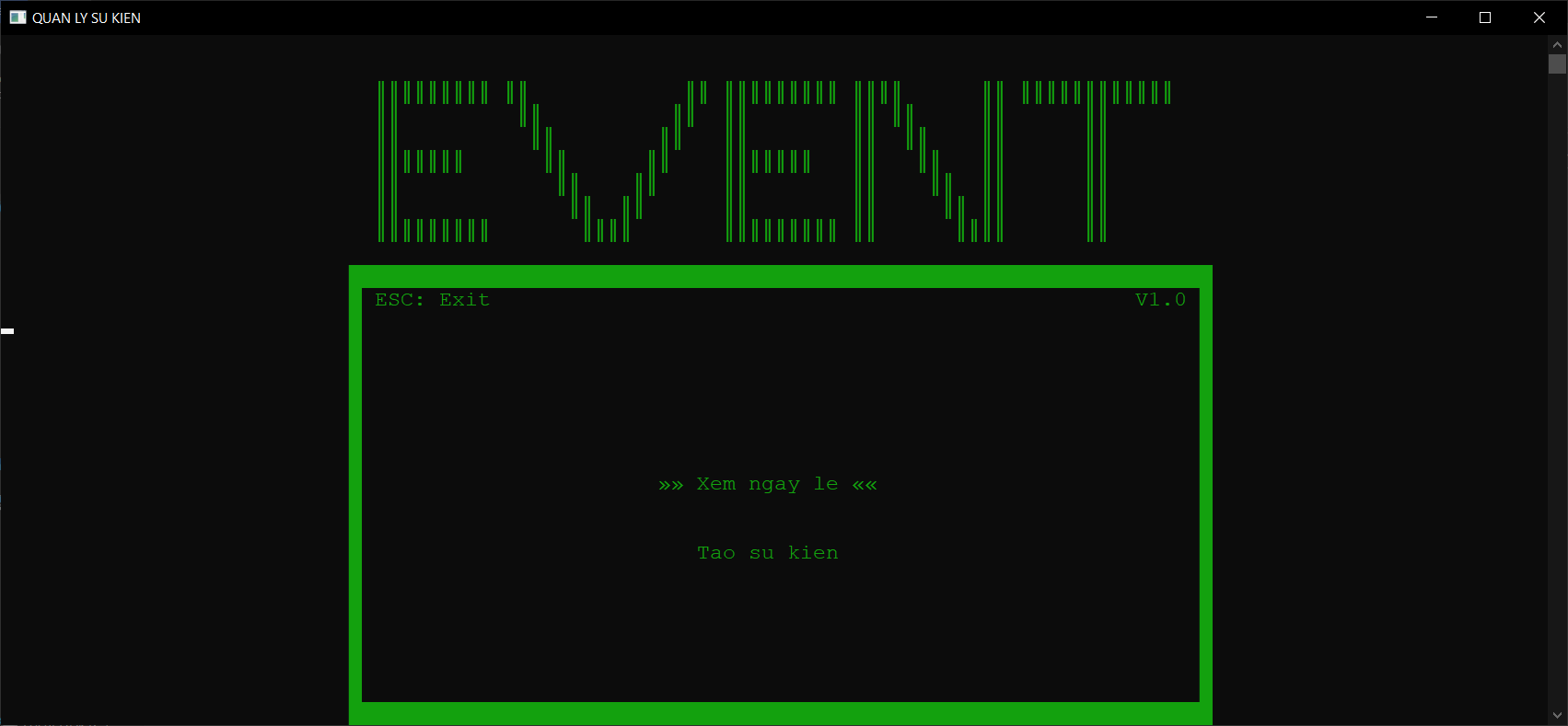
**3.2 Một số giao diện giao diện tự code :**

**+ Sử dụng mũi tên lên xuống để chọn và xem chức năng của chương trình**

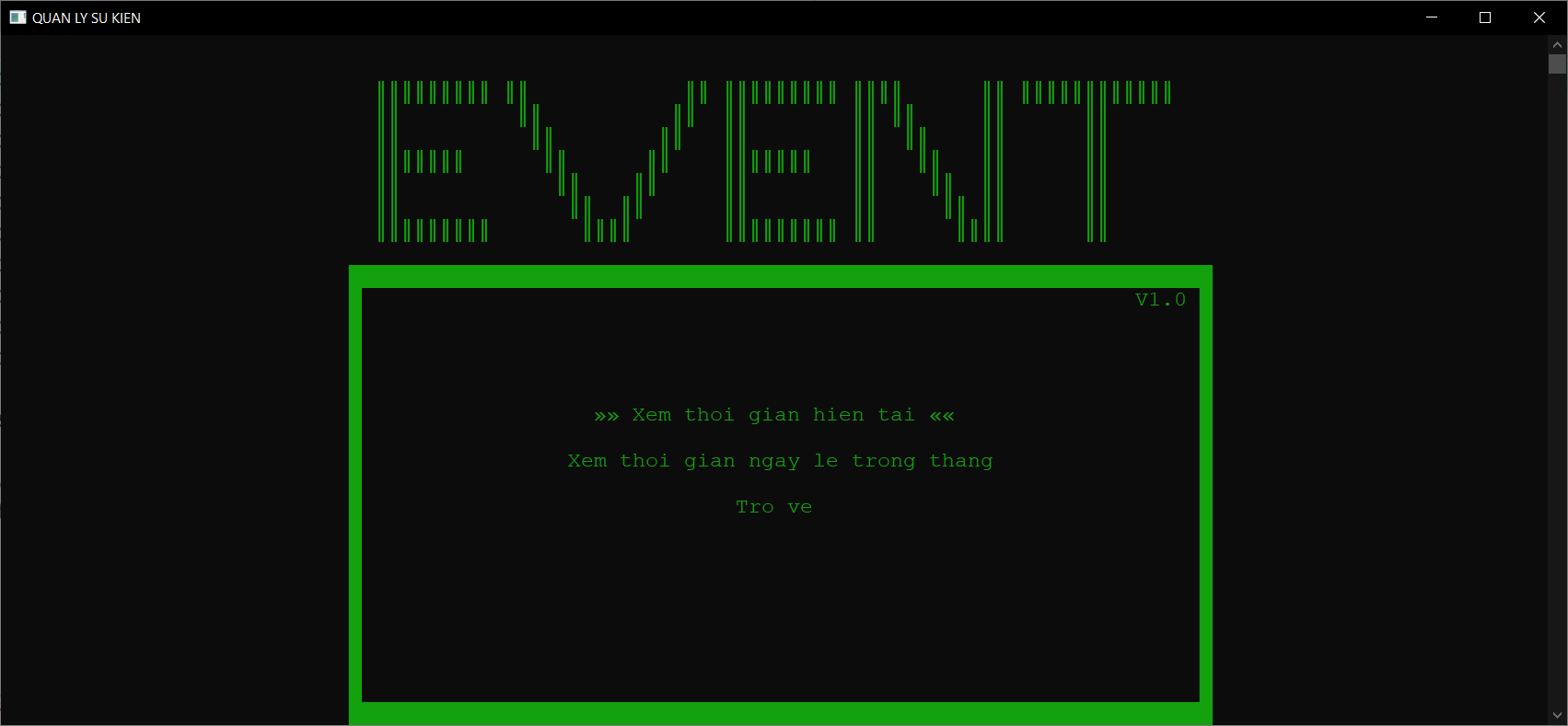
**Ví dụ trong chương trình :**

* int t = \_getch();
* if (t == 13)
* {
* switch (cn)
* {
* case 1:
* {
* time.ListMenuNgayLe();
* break;
* }
* case 2:
* {
* system("cls");
* Frames();
* listMenuEvent();
* break;
* }
* }
* }
* else if (t == 27)
* {
* system("cls");
* exit(0);
* }
* else if (t == 80)
* {
* cn++;
* }
* else if (t == 72)
* {
* cn--;
* }
* if (cn > 2)
* {
* cn = 1;
* }
* else if (cn < 1)
* {
* cn = 2;
* }
* }
* while (1);
* }

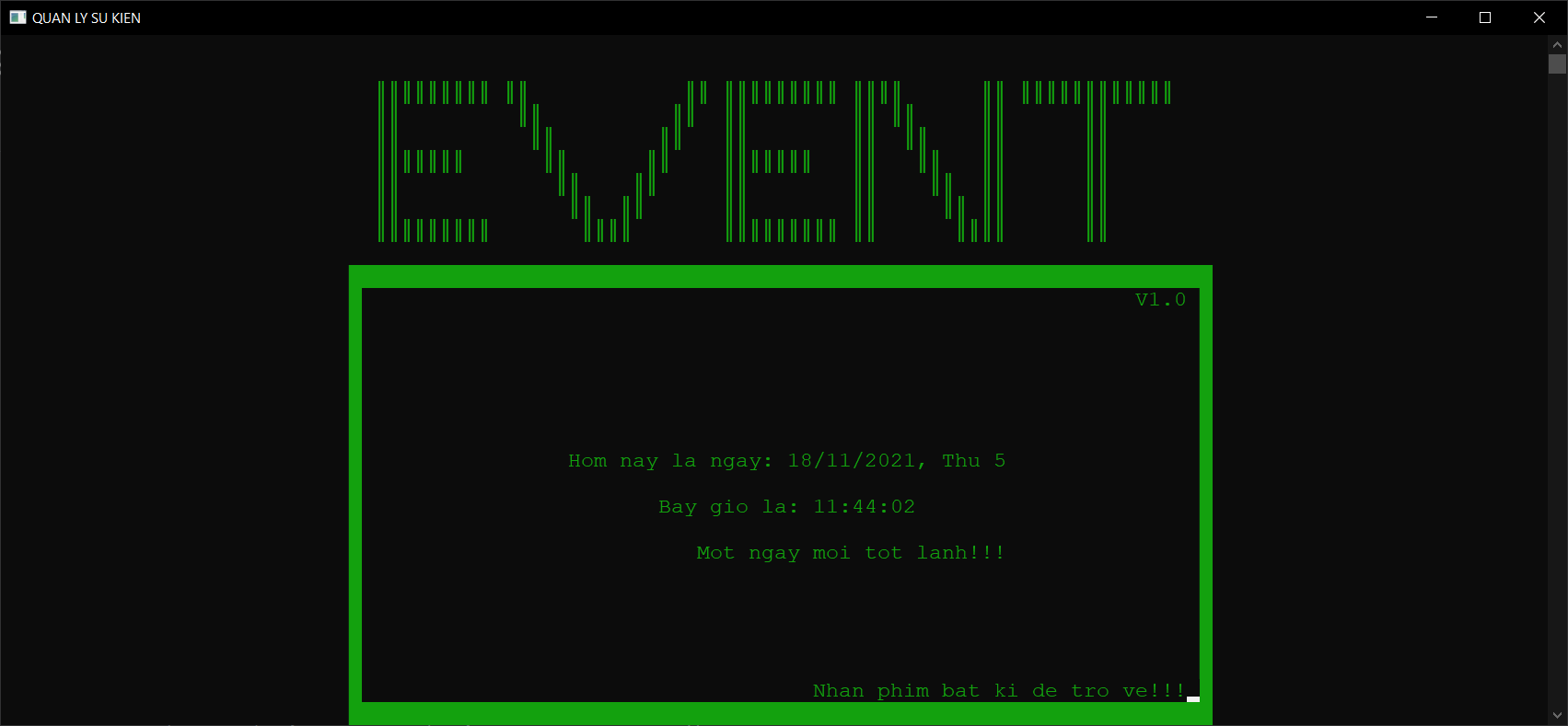
1. **Giao diện chương trình**
2. Giao diện MENU chương trình:



2.Chức năng xem ngày lễ :



* 1. Xem thời gian hiện tại :



* 1. Xem thời gian ngày lễ trong tháng :



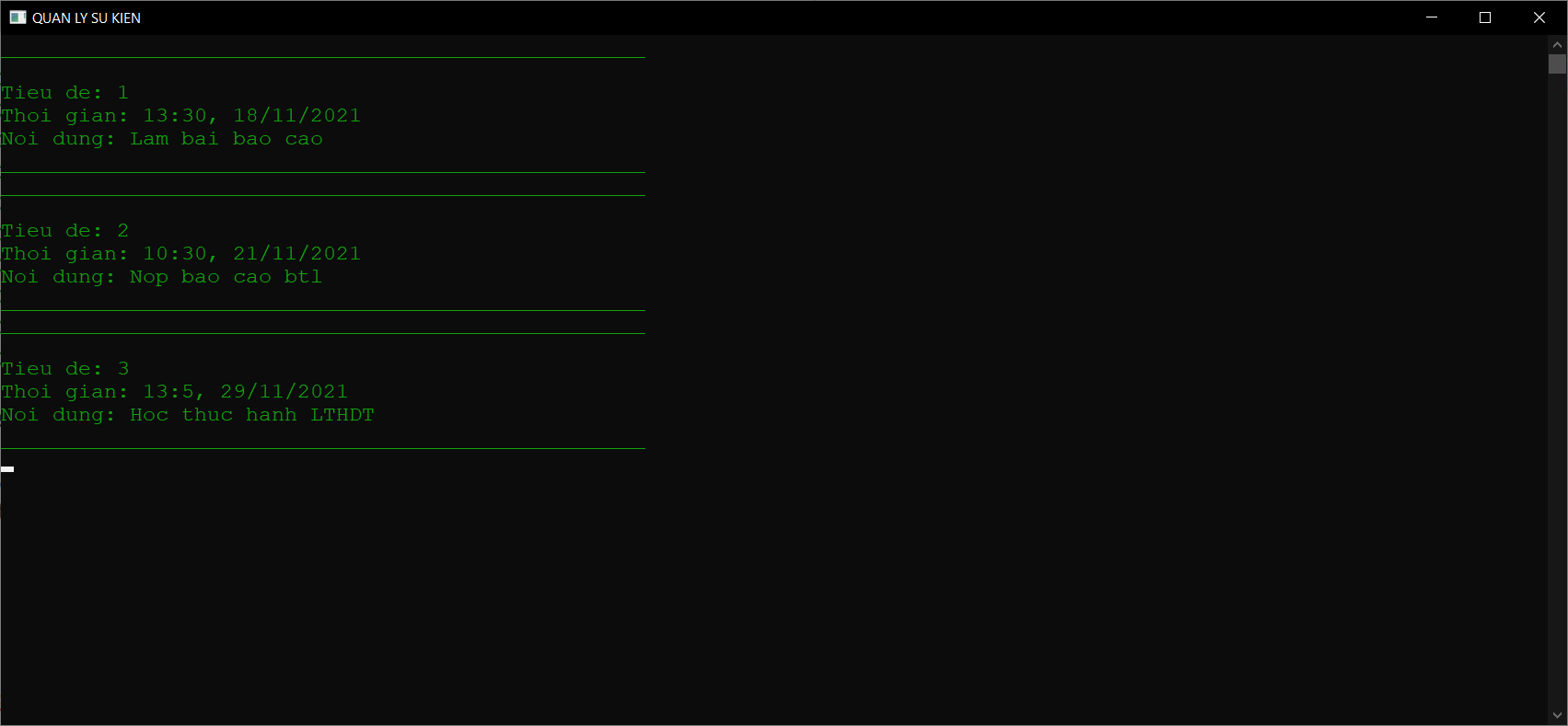
3. Chức năng tạo sự kiện :



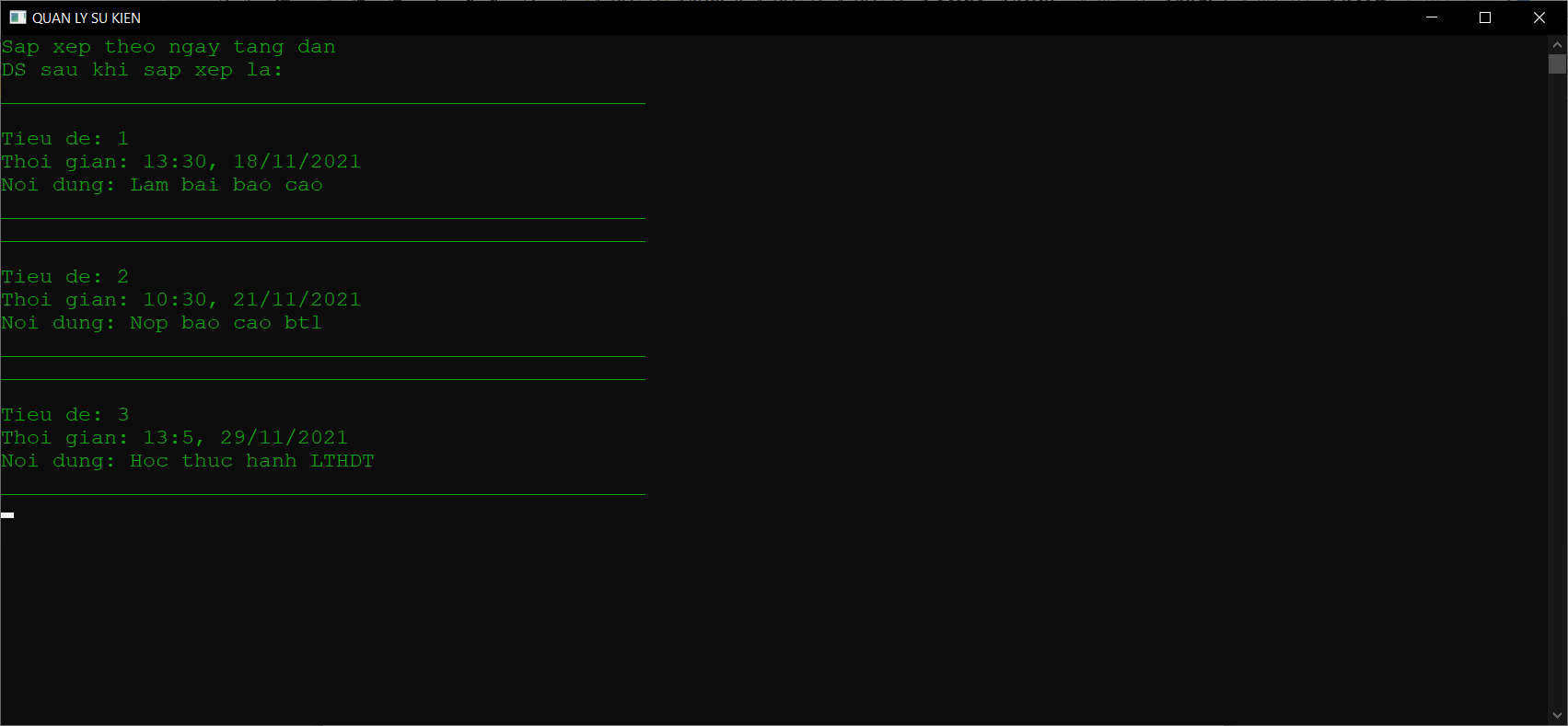
3.1 Thêm mới :



3.2 Hiển thị:



3.3 Sắp xếp theo ngày tăng dần :

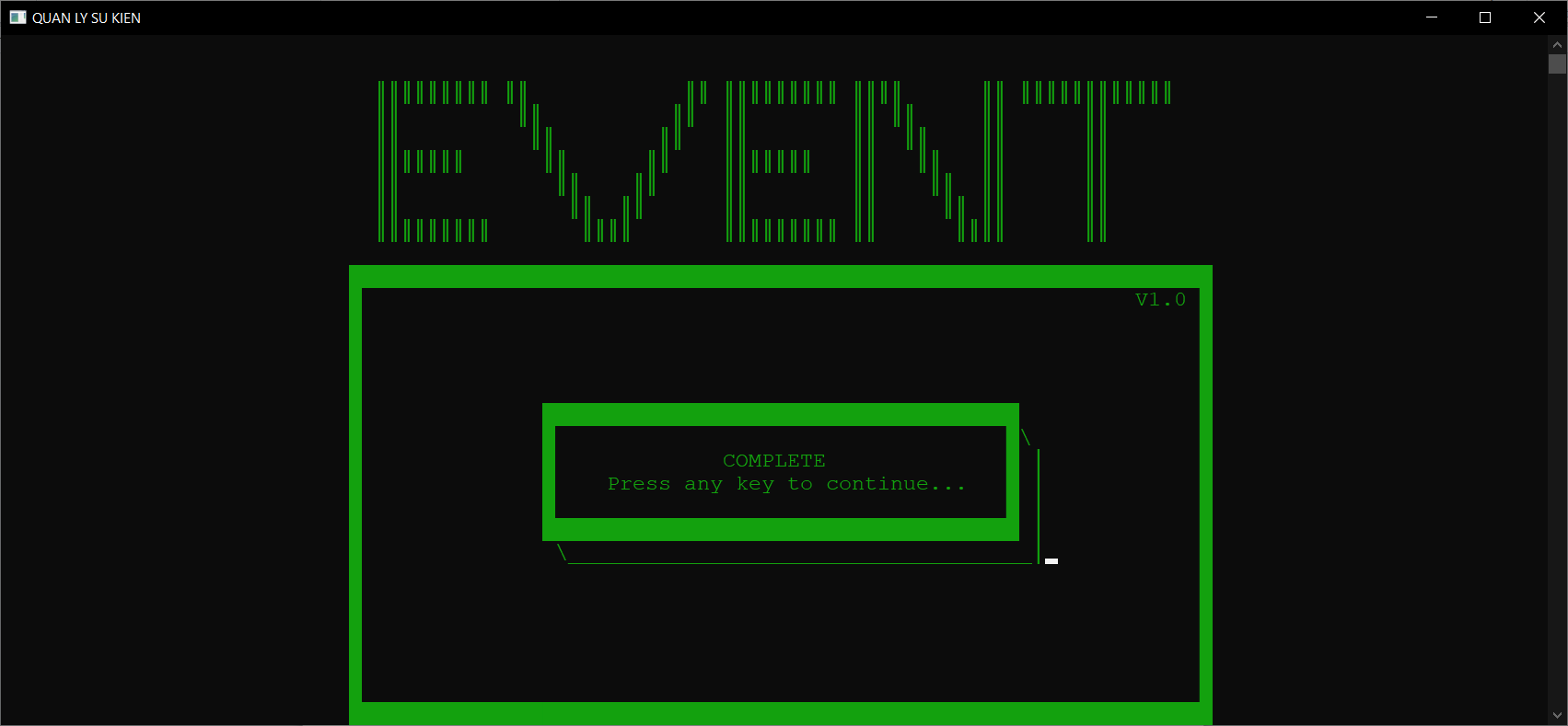


* 1. Tìm kiếm theo tiêu đề :

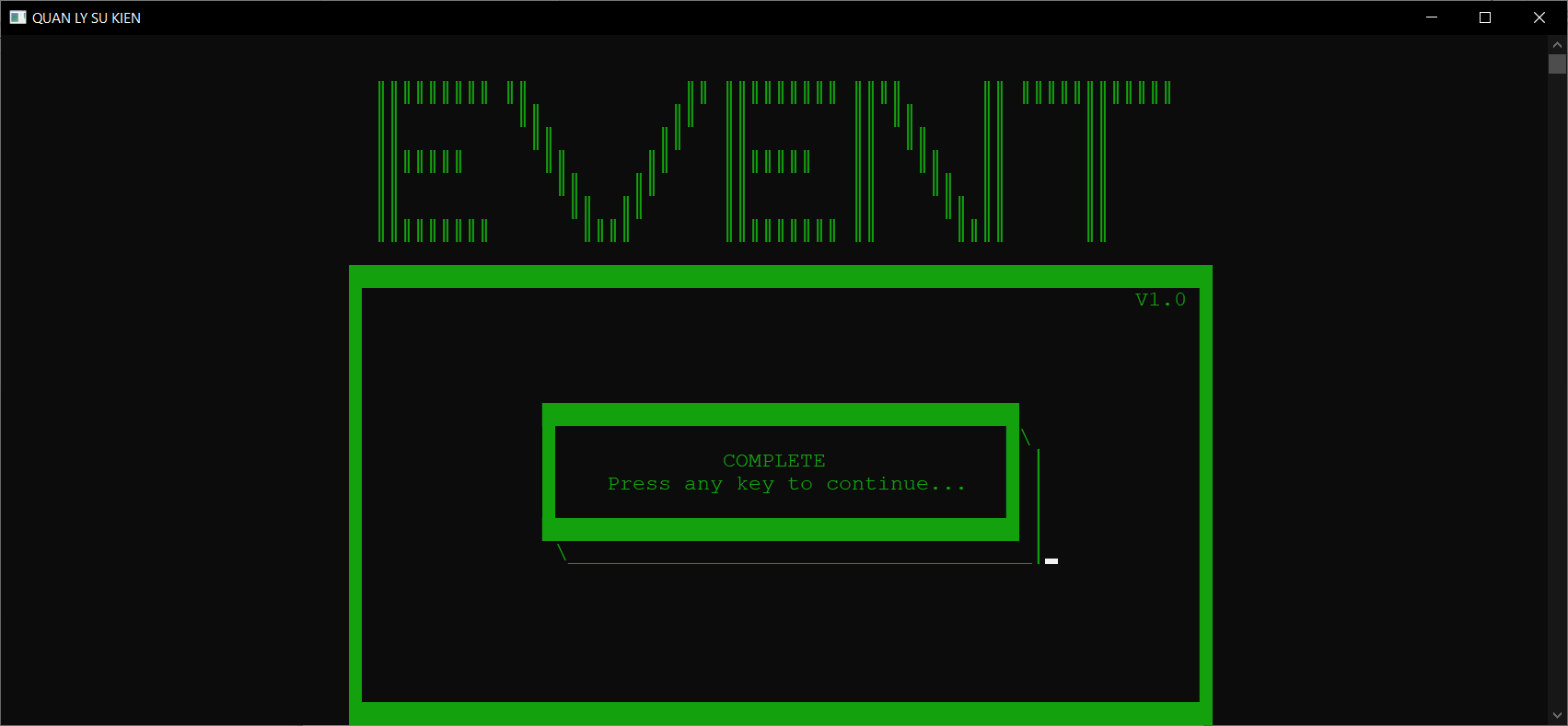


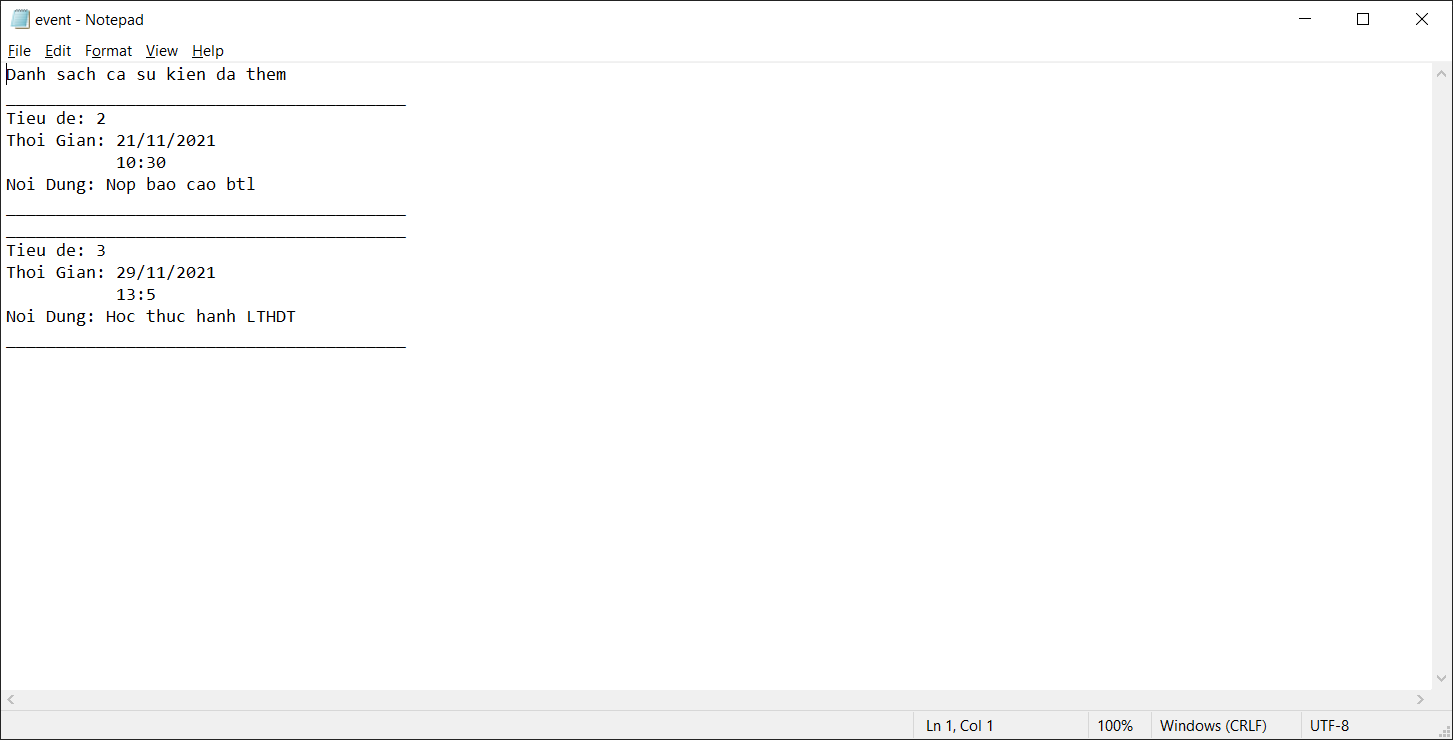
3.5 Xóa sự kiện :



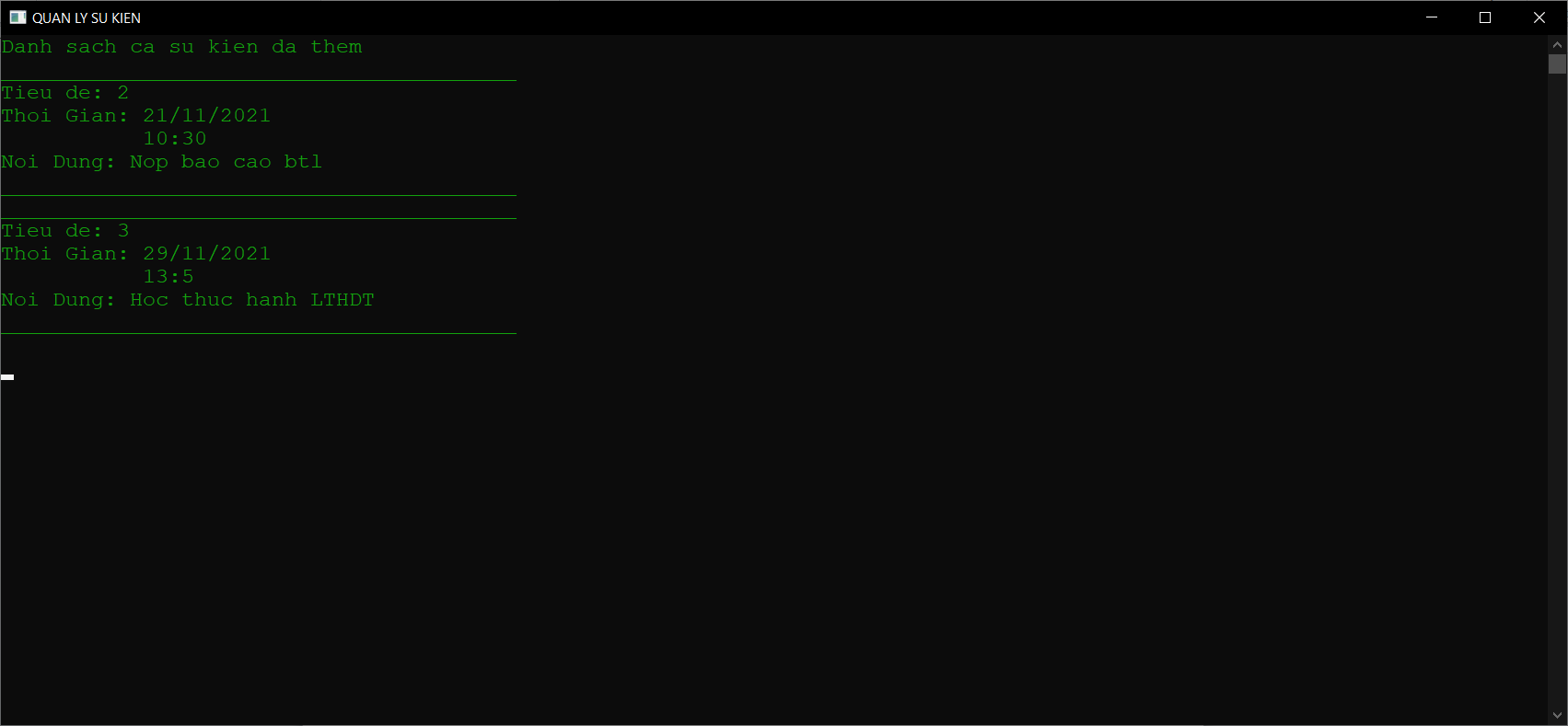


3.6 Lưu file sự kiện đã lưu:





3.7 Xem lại các sự kiện đã lưu:



**IV.Phụ lục**

1.Link github : <https://github.com/hung55410/BTL_LapTrinhHuongDoiTuong>

## 2. Hướng dẫn cài đặt chương trình:

* B1: Cài đặt 1 chương trình Dev C++

+ <https://sourceforge.net/projects/orwelldevcpp/>

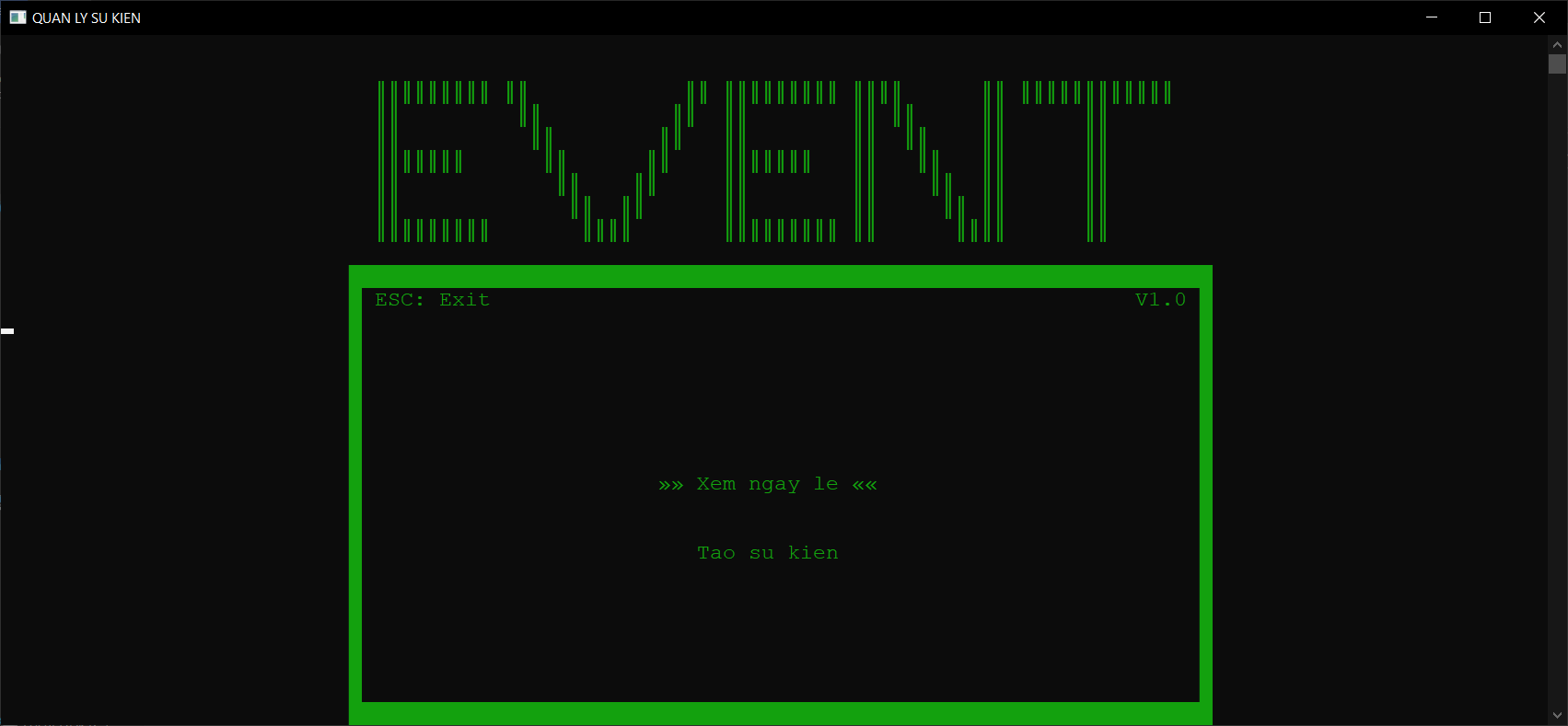
* B2 : Truy cập đường link google drive tải file bai tap lon nhom 16.rar và giải nén

+ <https://drive.google.com/file/d/1yb6rr6AhHRjmK0OV2aY6k9MvES7YBmc1/view?usp=sharing>

* B3: Sau khi giải nén file , vào Dev C++ chạy file main.cpp để chạy chương trình.

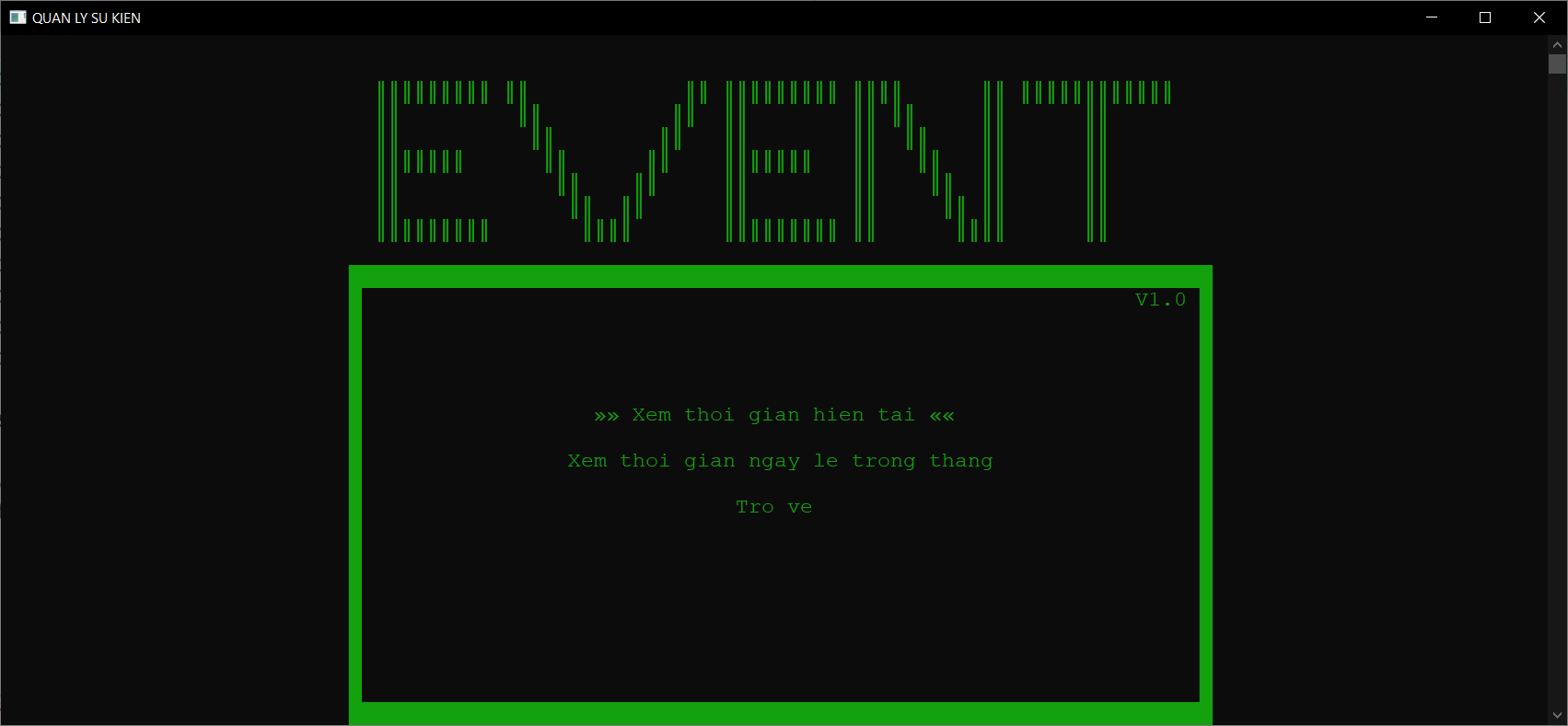
**3.Hướng dẫn sử dụng chương trình :**

3.1 Khi chạy chương trình sẽ hiện ra MENU và 2 tùy chọn là xem ngày lễ và tạo sự kiện sử dụng mũi tên lên xuống để chọn rồi nhấn enter để chọn :

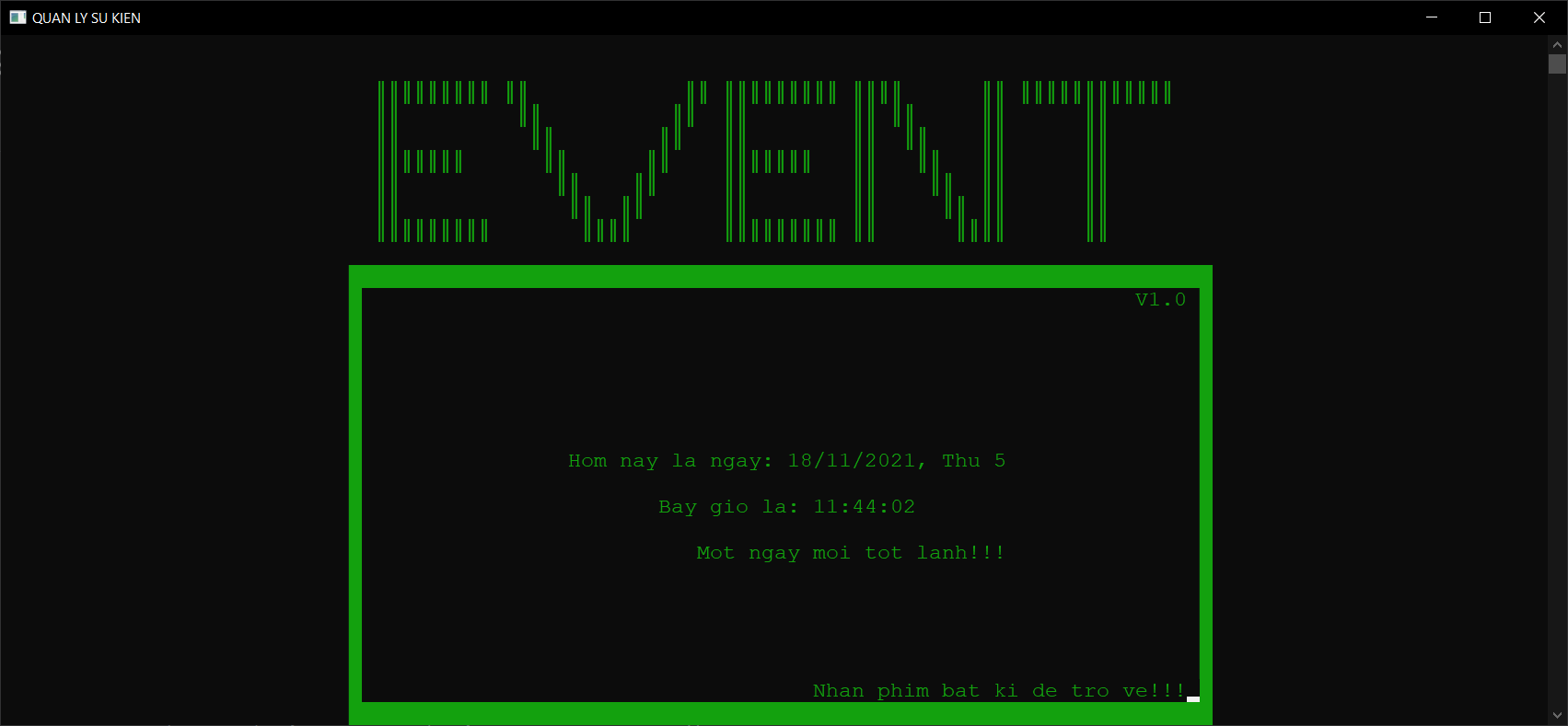


3.2 Nếu chọn chức năng xem ngày lễ màn hình sẽ hiện ra 2 chức năng là xem

thời gian hiện tại , xem thời gian ngày lễ trong tháng và trở về nhấn enter để chọn chức năng :



3.2.1 Chọn xem thời gian ở hiện tại sẽ hiển thị thứ ,ngày, tháng và giờ ở hiện tại :



3.2.2 Nếu chọn Xem thời gian ngày lễ trong tháng thì nhập tháng muốn xem ngày lễ rồi nhấn enter :



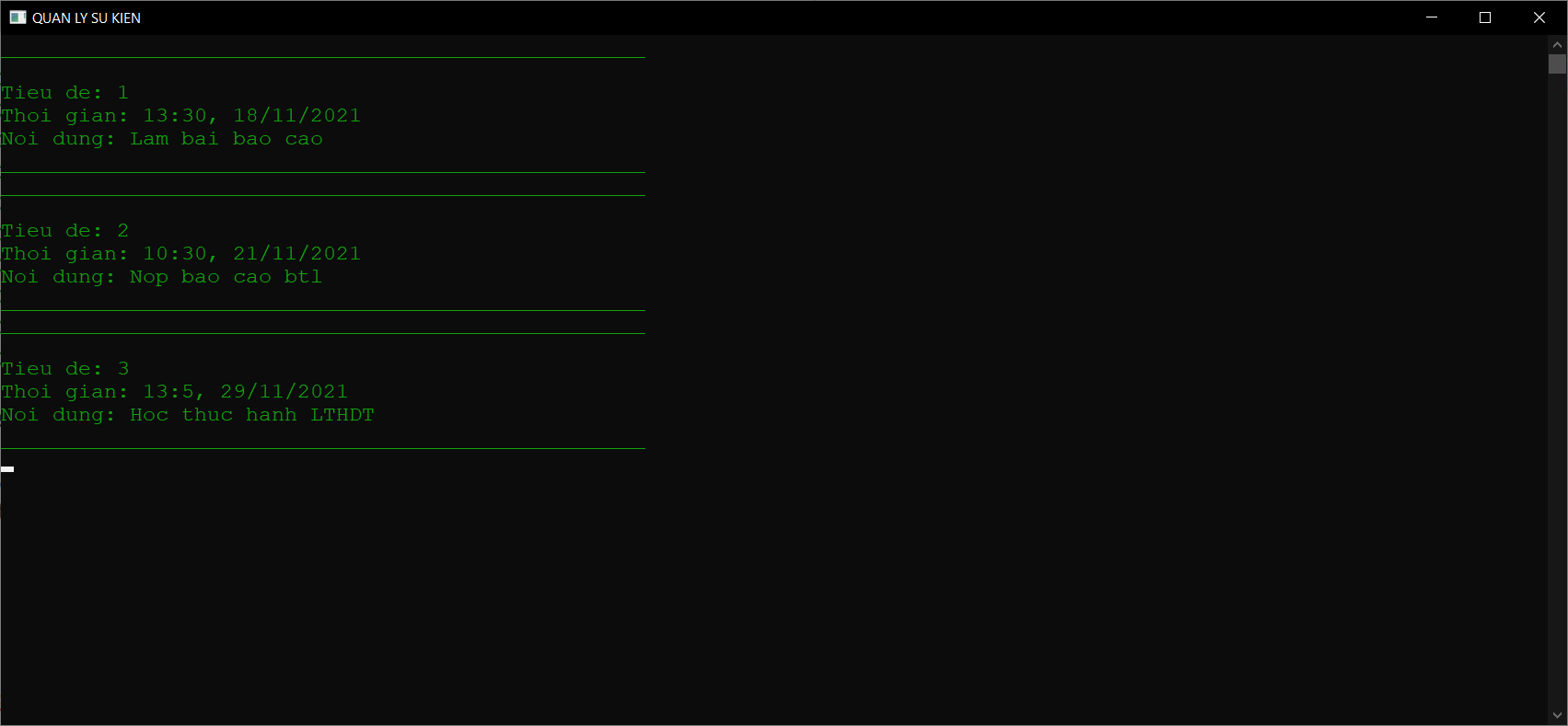
3.3. Khi chọn chức năng tạo sự kiện gồm có 8 phím chức năng sử dụng mũi tên lên xuống và chọn nhấn enter :



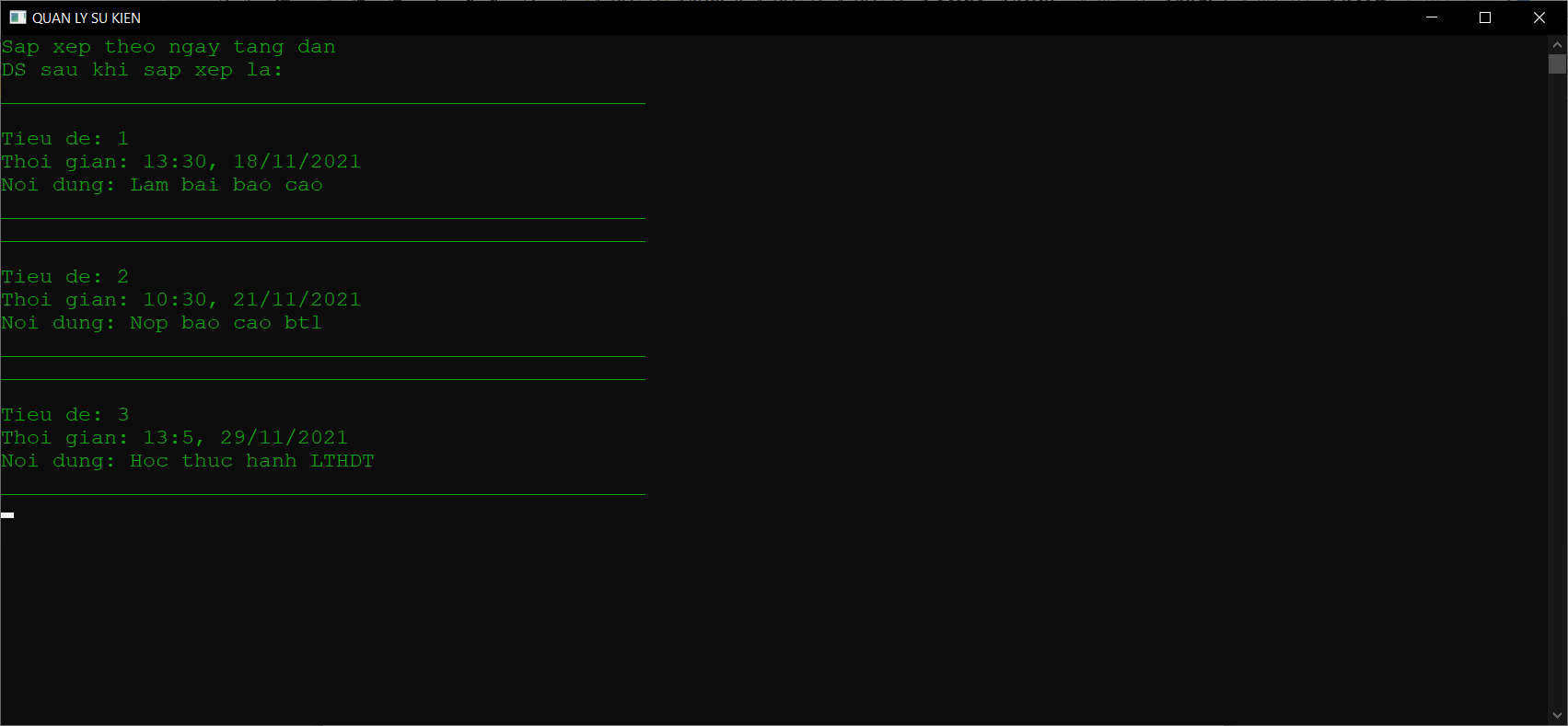
3.3.1 Chức năng thêm mới giúp nhập từ bàm phím 1 sự kiện vào danh sách lưu ý nhập đúng quy tắc của thời gian dể không phải nhập lại nhiều lần :



3.3.2 Khi chọn chức năng hiển thị chương trình sẽ xuất ra danh sách sự kiện đã nhập vào :



3.3.3 Muốn sắp xếp ngày theo tứ tự tăng dần thì chọn chức năng sắp xếp màn hình sẽ hiển thị thứ tự sự kiện tăng dần :

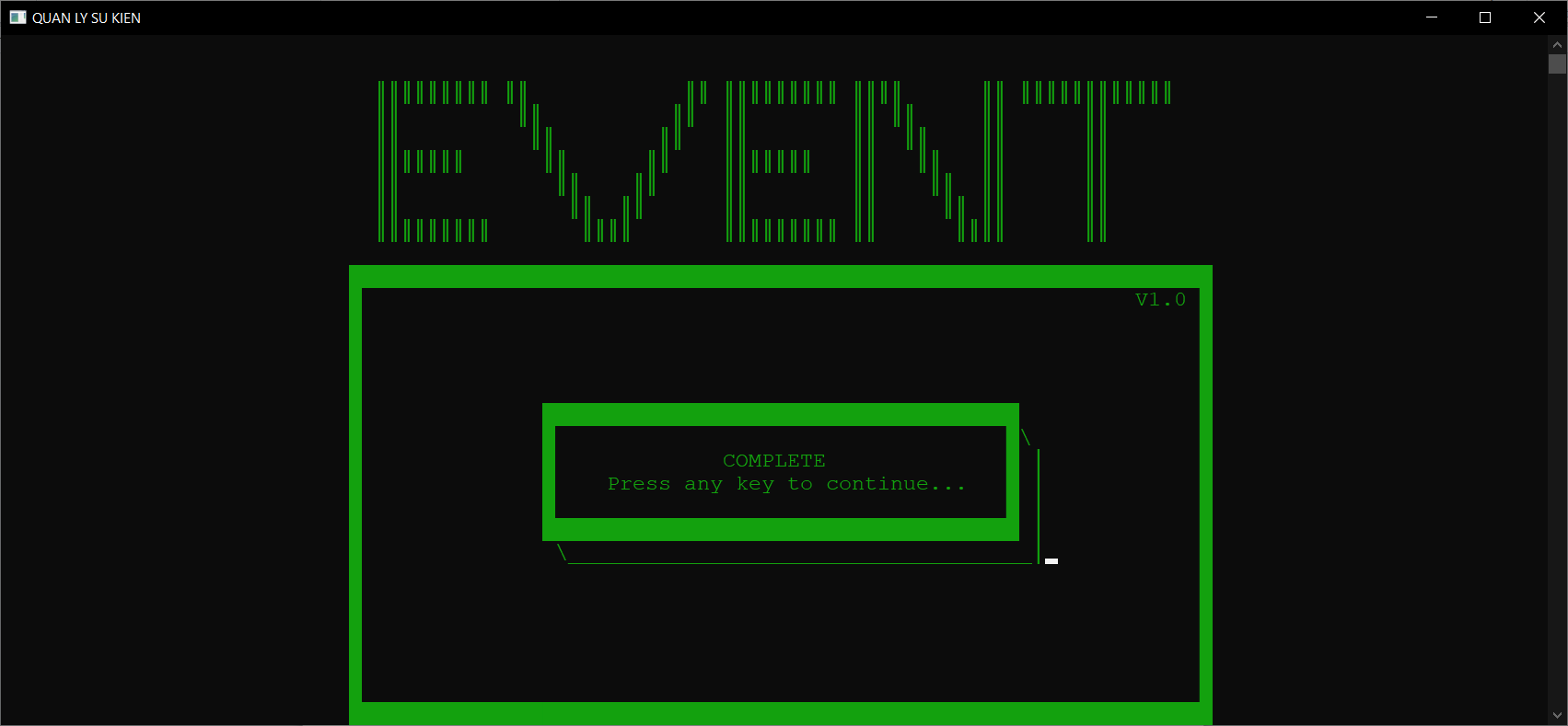


3.3.4 Muốn tìm sự kiện theo tiêu đề chọn chức năng tìm kiếm lưu ý nhập đúng tiêu đề để không phải nhập nhiều lần :

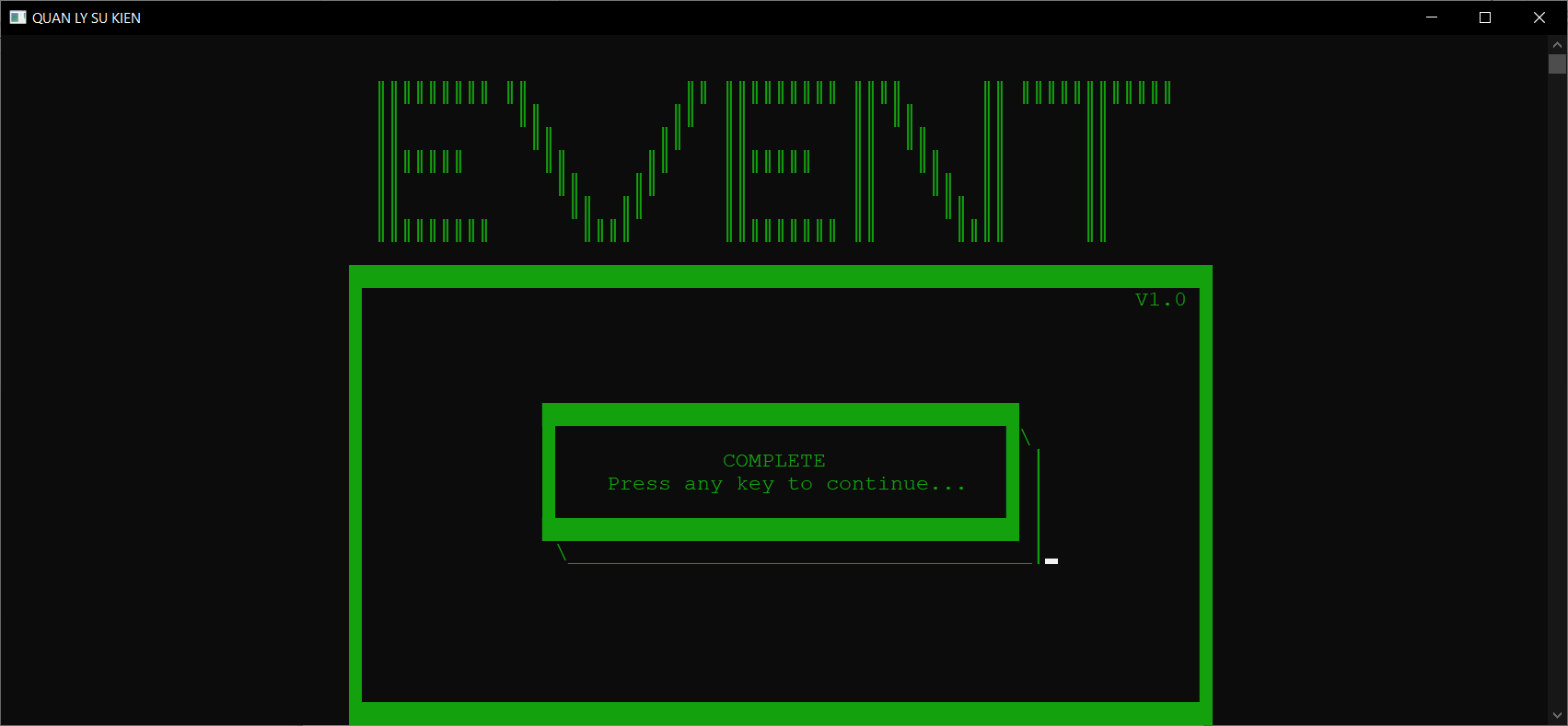


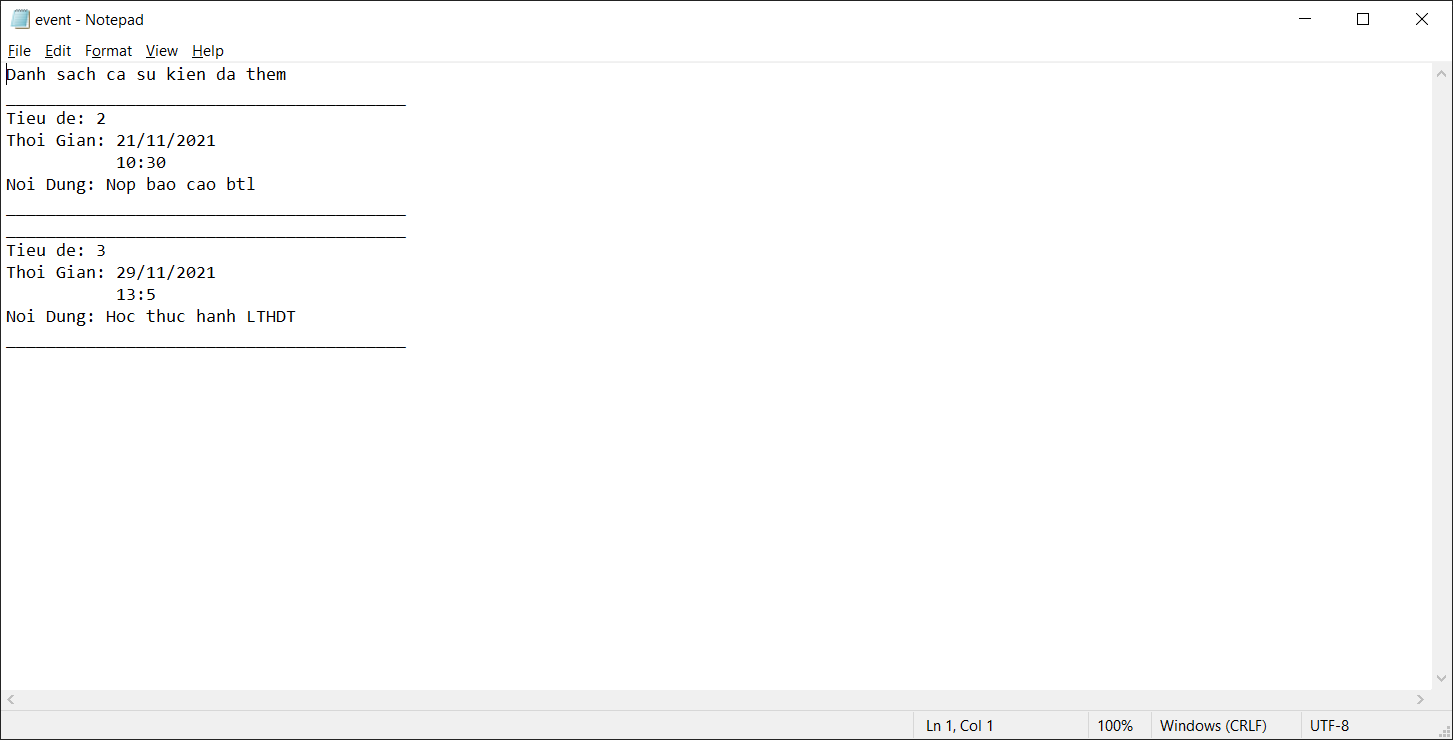
3.3.5 Nếu muốn xóa sự kiện , nhấn vào DELETE xong nhập tiêu đề sự kiện cần xóa rồi enter , lưu ý ghi đúng tiêu đề dể không phải nhập nhiều lần :



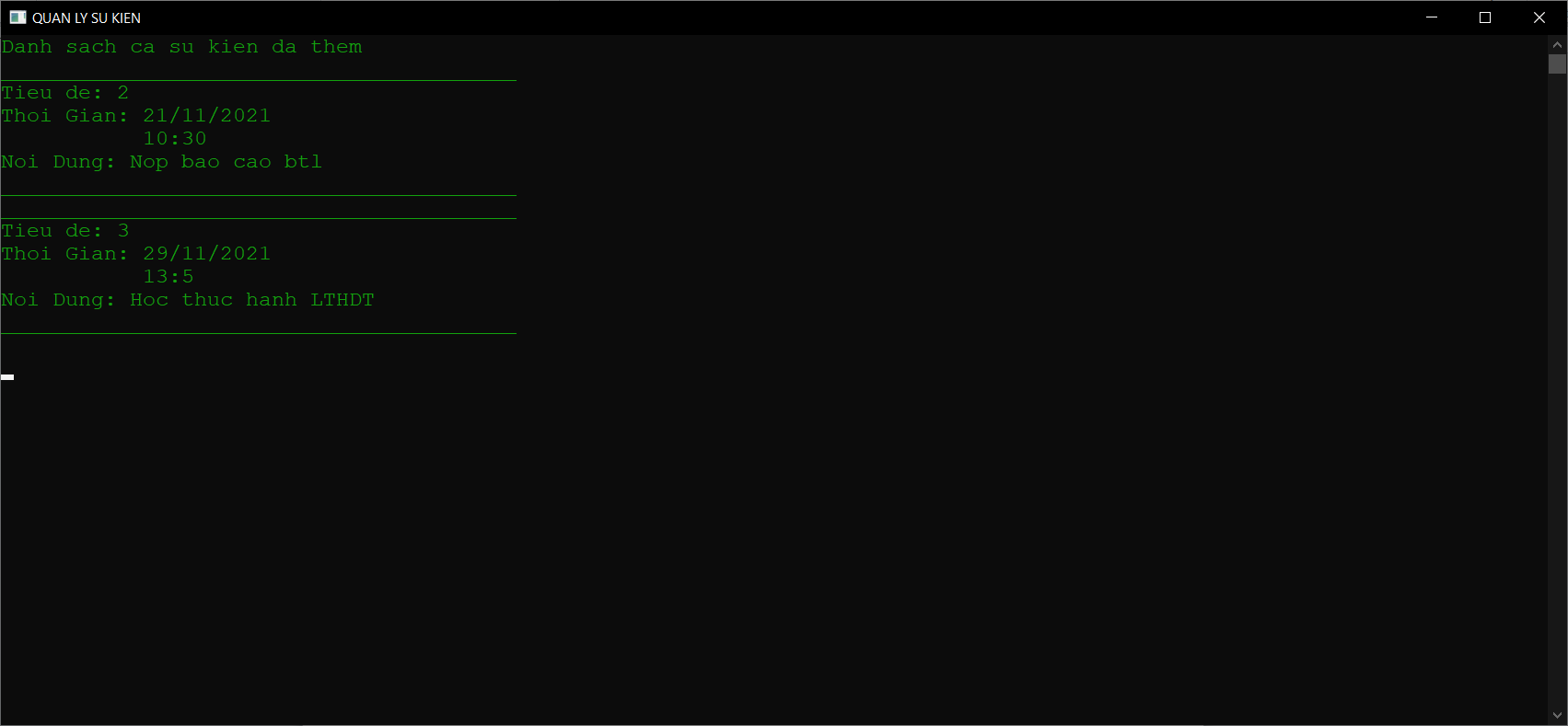


3.3.6 Sau khi nhập , xem , sắp xếp , xóa ,xong sử dụng chức năng SAVE FILE để lưu sự kiện và ghi ra 1 file Notepad ,lưu ý file Notepad sẽ được lưu :





3.3.7 Trước khi thoát khỏi chương trình để chắc chắn file Notepad đã đầy đủ các sự kiện muốn lưu , có thể sử dụng chức năng OPEN FILE để xem lại nhưng sự kiện đã lưu vào file Notepad:



**Hết.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*