

## **ТЕМА №3. РОЗШИРЕНЕ ПРЕДСТАВЛЕННЯ ДАНИХ**

### **Лекція №7. Зв'язні списки**

- 1 Дослідження представлення даних.
- 2 Від масиву до зв'язного списку.
- 3 Використання зв'язного списку.
- 4 Відображення списку.
- 5 Створення списку.
- 6 Звільнення пам'яті, яка була зайнята списком.
- 7 Додаткові міркування.

На даний момент ви вже маєте певні теоретичні та практичні навички у створенні змінних, структур, функцій і т. д. Однак з часом необхідно переходити на більш високий рівень, де реальним завданням стає проектування та реалізація проекту як єдиного цілого.

Тому почнемо сьогоднішню лекцію з ознайомлення з надзвичайно важливим аспектом проектування програми: способом представлення даних. Найчастіше найбільш важливим аспектом розробки програми є вибір відповідного представлення даних, якими вона буде маніпулювати. Правильний вибір представлення даних може перетворити написання програми на дуже просту задачу.

Ви вже знайомі з вбудованими типами даних: простими змінними, масивами, вказівниками, структурами та об'єднаннями. Проте, часто вибір правильного представлення даних не обмежується простим вибором типу. Ви повинні також подумати й про те, які операції вам доведеться виконувати. Тобто потрібно обирати спосіб зберігання даних і визначити, які операції є припустимими для обраного типу даних. Наприклад, в реалізаціях `c` тип `int` і тип вказівника зазвичай зберігаються як цілі числа, але для кожного з них визначено свій набір допустимих операцій. Скажімо, одне ціле число можна помножити на інше, але не можна помножити вказівники. Операцію `'*'` можна застосовувати для розіменування вказівника, але вона не має сенсу для цілочислового значення.

У мові `c` визначені допустимі операції для його фундаментальних типів. Проте, при проектуванні схеми представлення даних може знадобитися самостійно визначити допустимі операції. Мовою `c` це можна зробити шляхом розробки функцій, які представляють бажані операції. Тобто, проектування типу даних складається з визначення способу зберігання даних і розробки функцій для керування даними.

Ви також ознайомитеся з деякими алгоритмами, які є готовими рецептами для маніпулювання даними. Як програміст, ви з часом будете мати набір таких рецептів, які доведеться знову і знову застосовувати для вирішення схожих завдань.

На цій та наступних лекціях ми будемо розглядати з вами процес проектування типів даних, тобто процес зіставлення алгоритмів з представленнями даних. Ви розглянете низку поширених форм даних, таких як черга, список і двійкове дерево пошуку.

Також буде представлена концепція абстрактного типу даних (**abstract data type – ADT**). Тип **ADT** упаковує методи та представлення даних проблемно-орієнтованим, а не мовно-орієнтованим способом. Після того як ви спроектували абстрактний тип даних, його можна легко багаторазово використовувати при різних обставинах. Розуміння типів **ADT** концептуально підготує вас до світу об'єктно-орієнтованого програмування та мов **C++**, **C#**, **Java**.

## 1 Дослідження представлення даних

Давайте почнемо з обмірковування даних. Припустимо, що потрібно створити програму для адресної книги. Яку форму даних необхідно використовувати для зберігання інформації? Оскільки з кожним записом пов'язана різноманітна інформація, кожен запис має сенс представити у вигляді структури. А як представити кілька записів? За допомогою стандартного масиву структур? За допомогою динамічного масиву? За допомогою якоїсь іншої форми? Чи повинні записи бути впорядковані в алфавітному порядку? Чи потрібна можливість пошуку в записах за поштовим індексом? Чи потрібен пошук за міжміським телефонним кодом? Дії, які потрібно виконувати, можуть впливати на вибір способу зберігання інформації. Тобто, перш ніж приступати до створення коду, доведеться прийняти масу проектних рішень.

А як ви представите растрові графічні зображення, які повинні зберігатися в пам'яті? У растровому зображенні кожен піксель на екрані встановлюється індивідуально. За часів чорно-білих екранів для представлення одного пікселя можна було використовувати один біт (1 або 0) – звідси і англійська назва растрових графічних зображень **bitmapped** (бітове зображення). На кольорових моніторах опис одного пікселя займає більше одного біта. Наприклад, виділення по 8 бітів для кожного пікселя дозволяє отримати 256 кольорів. На даний час

відбувся перехід до 65 536 кольорів (16 бітів на піксель), 16 777 216 кольорів (24 біти на піксель); 2 147 483 648 (32 біти на піксель) і навіть більше. При наявності 32-бітових кольорів і роздільної здатності монітора 2560×1440 пікселів для представлення одного екрану растрової графіки вам знадобиться близько 118 мільйонів бітів (14 мегабайтів). Чи слід змиритися з цим або ж розробити якийсь метод стиснення інформації? Чи повинно це стиснення виконуватися без втрат або с втратами (порівняно неважливих даних)? І знову, перш ніж займатися кодуванням, доведеться прийняти безліч проектних рішень.

Розглянемо конкретний випадок представлення даних. Припустимо, що потрібно написати програму, яка дозволяє вводити список усіх фільмів (на відеокасетах, дисках DVD і дисках Blu-ray), що були переглянуті протягом року. Для кожного фільму бажано реєструвати різноманітну інформацію, таку як назва, рік випуску, імена та прізвища режисера та провідних акторів, тривалість і жанр (комедія, наукова фантастика, романтика, мелодрама та ін.), рейтинг і т. д. Це передбачає застосування структури для кожного фільму та масиву структур для списку фільмів. Для спрощення обмежимо структуру двома членами: назвою фільму та власною оцінкою його рейтингу за 10-бальною шкалою. Текст програми, що використовує цей підхід має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define TSIZE 45      // розмір масиву для зберігання назви
#define FMAX 5        // максимальна кількість назв фільмів

struct Film
{
    char title[TSIZE];
    int rating;
};

char *s_gets(char *st, int n);
int main(void)
{
    struct Film movies[FMAX];
    int i = 0;
    int j;

    SetConsoleCP(1251);
```

```

SetConsoleOutputCP(1251);

puts("Введіть назву першого фільму:");

while(i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
    movies[i].title[0] != '\0')
{
    puts("Введіть своє значення рейтингу <0-10>:");
    scanf("%d", &movies[i++].rating);
    while(getchar() != '\n')
        continue;
    puts("Введіть назву наступного фільму\n"
        "(або порожній рядок для припинення вводу):");
}
if(i == 0)
    printf("\nДані не введені.");
else
    printf("\nСписок фільмів:\n");
for(j = 0; j < i; j++)
    printf("Фільм: %s.    Рейтинг: %d\n", movies[j].title,
        movies[j].rating);
printf("Програма завершена.\n");
return 0;
}

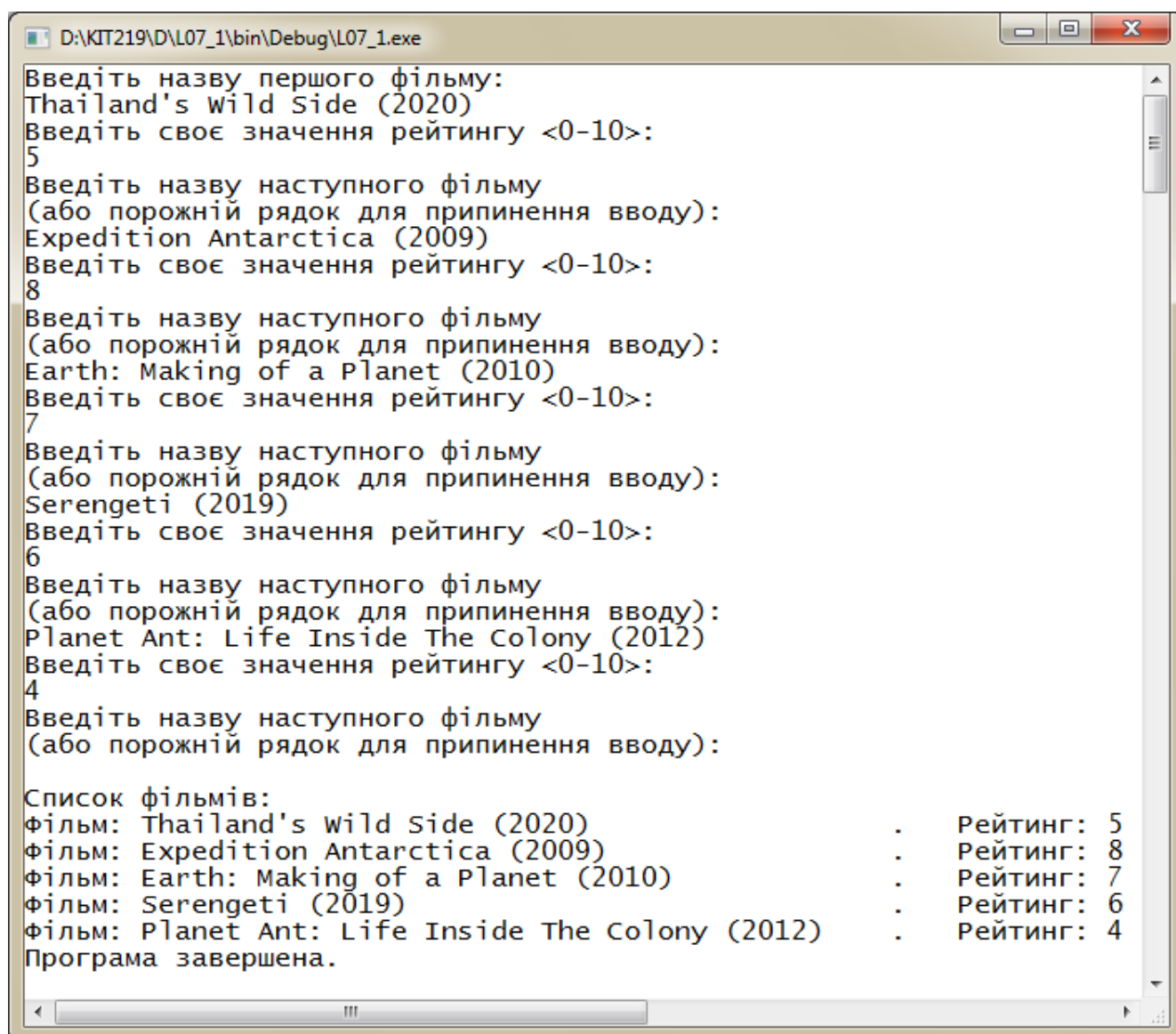
char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;

    ret_val = fgets(st, n, stdin);
    if(ret_val)
    {
        find = strchr(st, '\n');    // пошук нового рядка
        if(find)                    // якщо адреса не дорівнює NULL,
            *find = '\0';           // помістити туди нульовий символ
        else
            while(getchar() != '\n')
                continue;           // відкинути залишок рядка
    }
    return ret_val;
}

```

Результат виконання програми наведено на рис. 1.1.

Програма створює масив структур і заповнює його даними, які вводить користувач. Введення інформації триває аж до заповнення масиву (перевірка **FMAX**), до досягнення кінця файлу (перевірка **NULL**) або до натискання користувачем клавіші **<Enter>** на початку рядка (перевірка **'\0'**).



```

D:\KIT219\D\L07_1\bin\Debug\L07_1.exe
Введіть назву першого фільму:
Thailand's Wild Side (2020)
Введіть своє значення рейтингу <0-10>:
5
Введіть назву наступного фільму
(або порожній рядок для припинення вводу):
Expedition Antarctica (2009)
Введіть своє значення рейтингу <0-10>:
8
Введіть назву наступного фільму
(або порожній рядок для припинення вводу):
Earth: Making of a Planet (2010)
Введіть своє значення рейтингу <0-10>:
7
Введіть назву наступного фільму
(або порожній рядок для припинення вводу):
Serengeti (2019)
Введіть своє значення рейтингу <0-10>:
6
Введіть назву наступного фільму
(або порожній рядок для припинення вводу):
Planet Ant: Life Inside The Colony (2012)
Введіть своє значення рейтингу <0-10>:
4
Введіть назву наступного фільму
(або порожній рядок для припинення вводу):

Список фільмів:
Фільм: Thailand's Wild Side (2020) . Рейтинг: 5
Фільм: Expedition Antarctica (2009) . Рейтинг: 8
Фільм: Earth: Making of a Planet (2010) . Рейтинг: 7
Фільм: Serengeti (2019) . Рейтинг: 6
Фільм: Planet Ant: Life Inside The Colony (2012) . Рейтинг: 4
Програма завершена.

```

Рисунок 1.1 – Результат виконання програми, яка представляє дані у вигляді структури

Така організація програми пов'язана з низкою проблем. По-перше, швидше за все, програма буде марно витрачати великий об'єм пам'яті, оскільки назви більшості фільмів містять менш ніж 45 символів, але, в той самий час, назви деяких фільмів можуть бути надто довгими. По-друге, обмеження в п'ять фільмів на рік багатьом буде здаватися надто суворим. Звичайно, цю межу можна збільшити, але якою вона повинна бути? Хтось переглядає до 500 фільмів на рік, тому значення **FMAX** можна було б збільшити до 500, але для деяких і цього може виявитися замало, в той час як для інших воно приводило б до марної витрати величезного об'єму пам'яті.

Крім того, деякі компілятори за замовчуванням обмежують об'єм пам'яті, доступної для змінних з автоматичним класом зберігання на зразок **movies**, і такий великий масив міг би перевищити вказане обмеження. Ситуацію можна виправити, зробивши масив статичним або зовнішнім або надавши інструкцію компілятору про необхідність застосування стеку більшого розміру. Однак це не вирішує проблему.

Проблема полягає в тому, що представлення даних визначено абсолютно негнучким чином. На етапі компіляції вам доводиться приймати рішення, які доцільніше приймати під час виконання. Це передбачає перехід до представлення даних, яке використовує динамічний розподіл пам'яті.

Можна спробувати застосувати наступний код:

```
#define TSIZE    45          // розмір масиву для зберігання назви
struct Film
{
    char title[TSIZE];
    int rating;
};

...

int n, i;
struct Film *movies;      // вказівник на структуру
...

printf("Вкажіть максимальну кількість фільмів: \n");
scanf("%d", &n);
movies = (struct Film *) malloc(n * sizeof(struct Film));
```

Вказівник **movies** можна застосовувати таким чином, ніби він є ім'ям масиву:

```
while(i < FMAX && gets(movies[i].title) != NULL &&
      movies[i].title[0] != '\0')
```

За рахунок використання функції **malloc()** ви можете відкласти визначення кількості елементів до моменту виконання програми, тому не слід буде виділяти пам'ять для 500 елементів, якщо їх необхідно тільки 20. Але при такому підході обов'язок ввести коректне значення для кількості записів покладається на користувача.

## 2 Від масиву до зв'язного списку

В ідеалі бажано мати можливість додавати дані необмежено (або до тих пір, поки програма не вичерпана доступну пам'ять), не вказуючи заздалегідь кількість записів, які будуть створені, і, не змушуючи програму виділяти величезний об'єм пам'яті без реальної на те необхідності. Цієї мети можна досягти, викликаючи `malloc()` після введення кожного запису і виділяючи лише такий об'єм пам'яті, якого достатньо для нового запису. Якщо користувач вводить інформацію про три фільми, програма викликає функцію `malloc()` три рази. Якщо користувач вводить інформацію про 300 фільмів, програма викликає `malloc()` триста разів.

Така прекрасна на перший погляд ідея породжує нову проблему. Щоб побачити, в чому вона полягає, порівняйте одноразовий виклик `malloc()` для виділення пам'яті під 300 структур `Film` з 300-кратним викликом цієї функції для виділення пам'яті кожного разу тільки для однієї структури `Film`. У першому випадку пам'ять розподіляється у вигляді одного безперервного блоку, і для відстеження вмісту потрібно єдина змінна вказівника на структуру `Film`, яка вказує на першу структуру в блоці. Як було показано в наведеному раніше фрагменті коду, проста форма запису з масивом забезпечує за допомогою вказівника доступ до кожної структури всередині блоку. Проблема з другим підходом – відсутність будь-якої гарантії того, що послідовні виклики `malloc()` призведуть до виділення суміжних блоків пам'яті. Це означає, що структури не обов'язково будуть збережені безперервно (рис. 1.2).

Таким чином, замість зберігання одного вказівника на блок з 300 структур доведеться зберігати 300 вказівників – по одному для кожної незалежно виділеної структури.

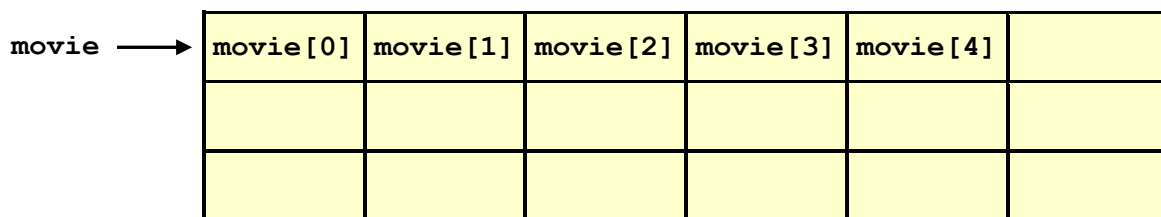
Одне з можливих рішень, яке, однак, ми застосовувати не будемо, передбачає створення великого масиву вказівників і присвоювання значень вказівниками в міру виділення пам'яті під нові структури:



```

struct Film *movie;
movie = (struct Film *) malloc(5 * sizeof(struct Film));

```



```

int i;
struct Films *movies[5];
for(i = 0; i < 5; i++)
    movies[i] = (struct Films *) malloc(sizeof(struct Films));

```

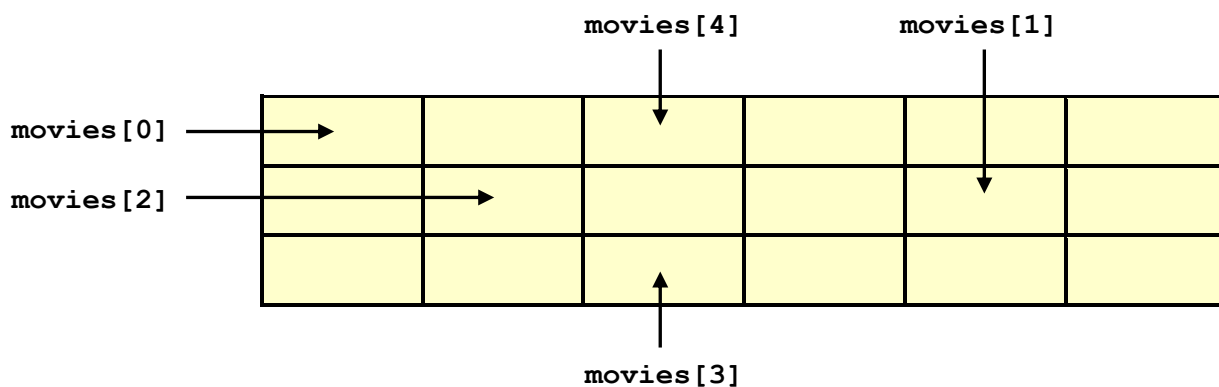


Рисунок 1.2 – Виділення пам'яті під структури одним блоком та індивідуально

```

#define TSIZE 45      // розмір масиву для збереження назв
#define FMAX 500     // максимальна кількість назв фільмів

struct Film
{
    char title[TSIZE];
    int rating;
};

...
struct Film *movies[FMAX]; // масив вказівників на структури
int i;
...
movies[i] = (struct Film *) malloc(sizeof(struct Film));

```

Цей підхід дозволяє зекономити великий об'єм пам'яті, якщо ви не використовуєте повний комплект вказівників, оскільки масив з 500 вказівників займає значно менше пам'яті, ніж масив з 500 структур. Проте, як і раніше простір витрачається марно на зберігання невикористаних вказівників, до того ж продовжує діяти обмеження в 500 структур.

Існує більш ефективний спосіб. При кожному виклику функції `malloc()` для виділення пам'яті під нову структуру одночасно можна виділяти пам'ять і для нового вказівника. Але ви можете заперечити, що тоді треба буде мати ще один вказівник для відстеження вказівника, який щойно був виділений, а для його відстеження необхідний ще один вказівник, і так до нескінченості. Запобігти цій потенційній проблемі можна шляхом перевизначення структури таким чином, щоб вона містила вказівник на наступну структуру. Тоді при кожному створенні нової структури її адресу можна буде зберігати в попередній структурі. Таким чином, структуру `Film` потрібно перевизначити так, як показано нижче:

```
#define TSIZE 45    // розмір масиву для зберігання назв
struct Film
{
    char title[TSIZE];
    int rating;
    struct Film *next;
};
```

Дійсно, структура не може містити структуру того ж самого типу, але вона може мати вказівник на структуру того ж самого типу. Визначення подібного роду є основою зв'язного списку – списку, в якому кожен елемент містить інформацію про місцезнаходження наступного елемента.

Перш ніж поглянути на код с-програми для зв'язного списку, давайте докладніше розглянемо концепції, що лежать в основі такого списку. Припустимо, що в якості назви фільму користувач вводить `Modern Times` і `10` для значення рейтингу. Програма виділила б пам'ять для структури `Film`, скопіювала б рядок `Modern Times` до члену `title` і встановила б значення члену `rating` у значення, яке дорівнює `10`. Щоб вказати на те, що за цією структурою немає жодних інших структур, програма повинна була б встановити значення члена-вказівника `next` в `NULL`. Зрозуміло, що при цьому необхідно відстежувати місце зберігання першої структури. Це можна зробити, присвоївши адресу окремому вказівнику, який ми будемо називати вказівником на заголовок списку. Вказівник на заголовок вказує на перший елемент у зв'язному списку елементів. На рис. 1.3 показано, як виглядає ця структура.

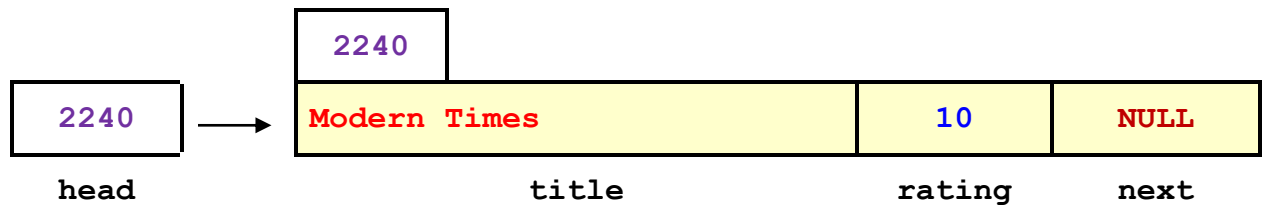


Рисунок 1.3 – Перший елемент у зв’язному списку

Тепер припустимо, що користувач вводить назву і рейтинг другого фільму – скажімо, **Midnight in Paris** і **8**. Програма виділяє пам’ять для другої структури **Film** і зберігає адресу нової структури в члені **next** першої структури (шляхом перезапису значення **NULL**, яке було встановлено раніше), щоб вказівник **next** посилався на наступну структуру у зв’язному списку.

```
#define TSIZE 45
struct Film
{
    char title[TSIZE];
    int rating;
    struct File *next;
};
struct File *head;
```

Потім програма копіює значення **Midnight in Paris** і **8** до нової структури і встановлює значення її члену **next** в **NULL**, вказуючи, що тепер ця структура є останньою у списку. Такий список з двох елементів наведено на рис. 1.4.

Обробка інформації про кожен новий фільм буде виконуватися аналогічно.

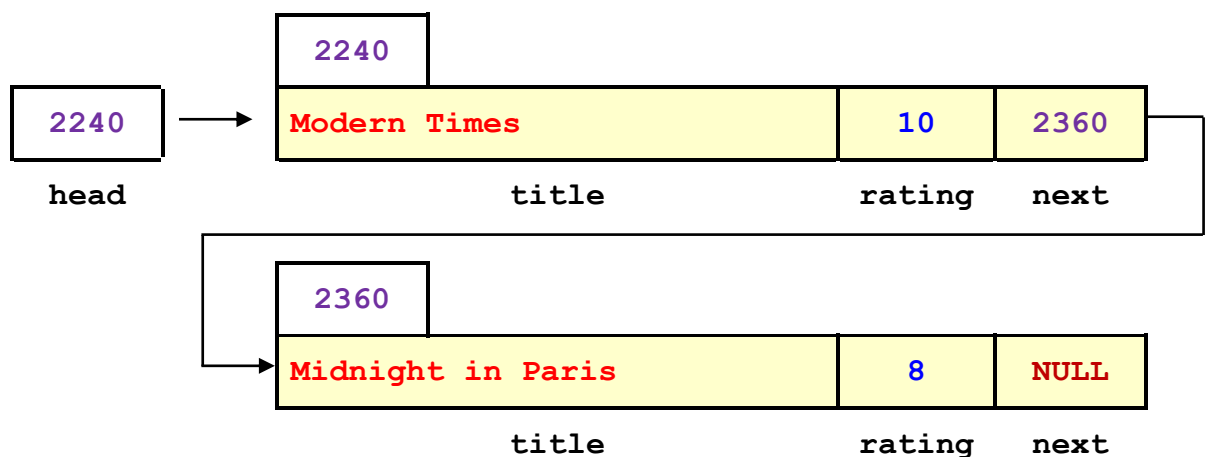


Рисунок 1.4 – Зв’язний список з двома елементами

Адреса нової структури буде зберігатися в попередній структурі, у новій структурі буде міститися введена інформація, а значення члену **next** нової структури буде встановлюватися в **NULL**, що призведе до створення зв'язного списку, подібного до представленого на рис. 1.5.

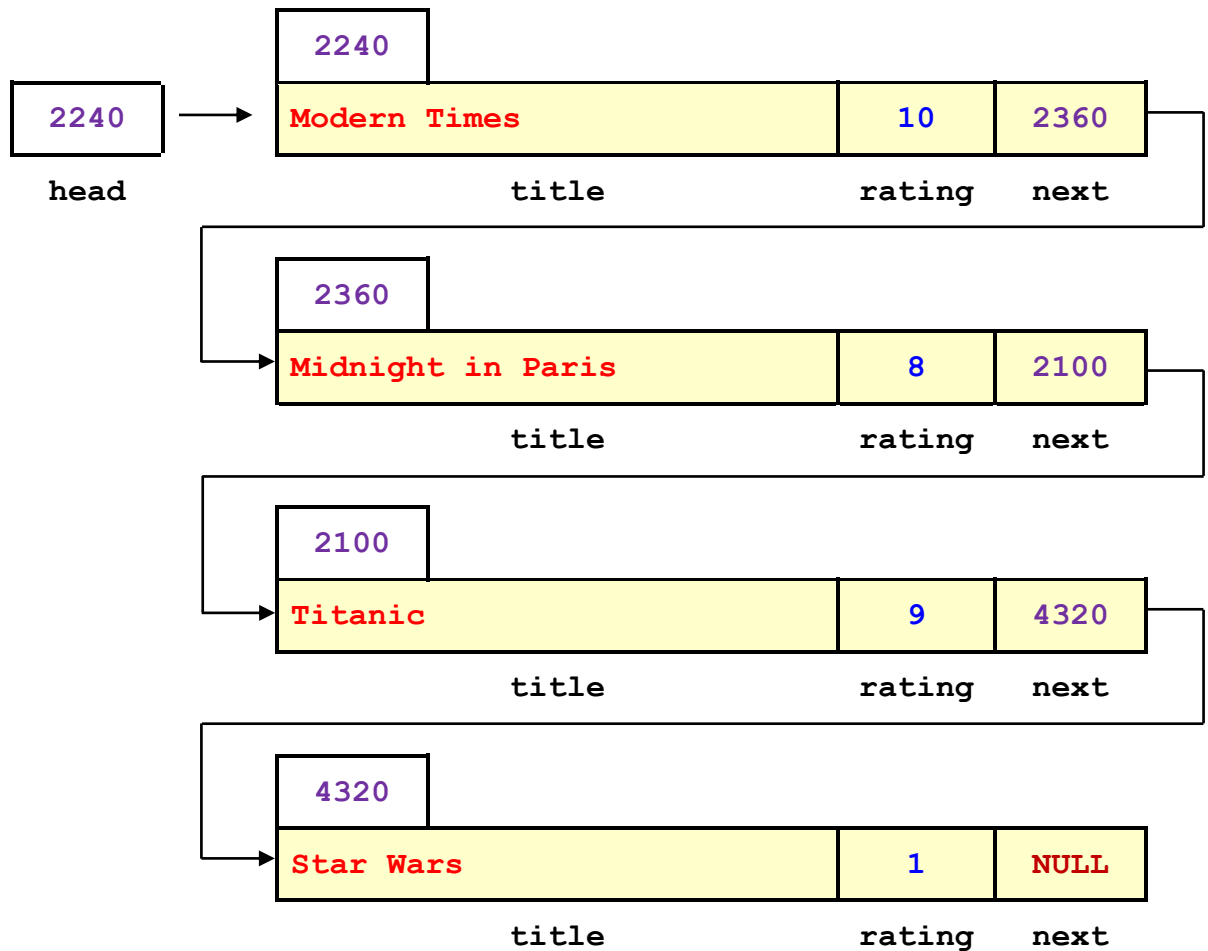


Рисунок 1.5 – Зв'язний список з чотирма елементами

Припустимо, що список необхідно вивести на екран. При кожному виводі елемента для знаходження наступного елемента, можна застосовувати адресу, яка була збережена у відповідній структурі. Однак щоб ця схема працювала, треба мати вказівник, який буде відстежувати самий перший елемент у списку, оскільки жодна структура у списку не зберігає адресу першого елемента. На щастя, це вже зроблено за допомогою вказівника на заголовок списку.

### 3 Використання зв'язного списку

Тепер, коли ви отримали уявлення про роботу зв'язного списку, давайте спробуємо його реалізувати. Текст програми, в якому для зберігання інформації про фільми замість масиву застосовується зв'язний список, має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>          // містить прототип функції malloc()
#include <string.h>          // містить прототип функції strcpy()
#define TSIZE 45            // розмір масиву для зберігання назв

struct Film
{
    char title[TSIZE];
    int rating;
    struct Film *next;      // вказує на наступну структуру в списку
};

char *s_gets(char *st, int n);

int main(void)
{
    struct Film *head = NULL;
    struct Film *prev, *current;
    char input[TSIZE];

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    // Збирання та збереження інформації
    puts("Введіть назву першого фільму: ");
    while(s_gets(input, TSIZE) != NULL && input[0] != '\0')
    {
        current = (struct Film *) malloc(sizeof(struct Film));
        if(head == NULL)      // перша структура
            head = current;
        else                  // наступні структури
            prev->next = current;
        current->next = NULL;

        strcpy(current->title, input);
        puts("Введіть своє значення рейтингу <0-10>:");
        scanf("%d", &current->rating);
        while(getchar() != '\n')
            continue;
        puts("Введіть назву наступного фільму\n"
            "(або порожній рядок для припинення вводу):");
        prev = current;
    }
}
```

```

// Відображення списку фільмів
if(head == NULL)
    printf("Дані не введені.");
else
    printf("Список фільмів:\n");
current = head;
while(current != NULL)
{
    printf("Рейтинг: %d. Фільм: %s. \n",
           current->rating, current->title);
    current = current->next;
}

// Програма виконана, тому можна звільнити пам'ять
current = head;
while(current != NULL)
{
    free(current);
    current = current->next;
}
printf("Програма завершена.\n");
return 0;
}

char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;

    ret_val = fgets(st, n, stdin);
    if(ret_val)
    {
        find = strchr(st, '\n');    // пошук нового рядка
        if(find)                    // якщо адреса не дорівнює NULL,
            *find = '\0';           // помістити туди нульовий символ
        else
            while(getchar() != '\n')
                continue;           // відкинути залишок рядка
    }
    return ret_val;
}

```

Програма вирішує два завдання з використанням зв'язного списку.

По-перше, вона конструює список і заповнює його вхідними даними.

По-друге, вона відображає список.

Відображення списку – це більш просте завдання, тому спочатку розглянемо саме його.

Результат роботи програми наведено на рис. 1.6.

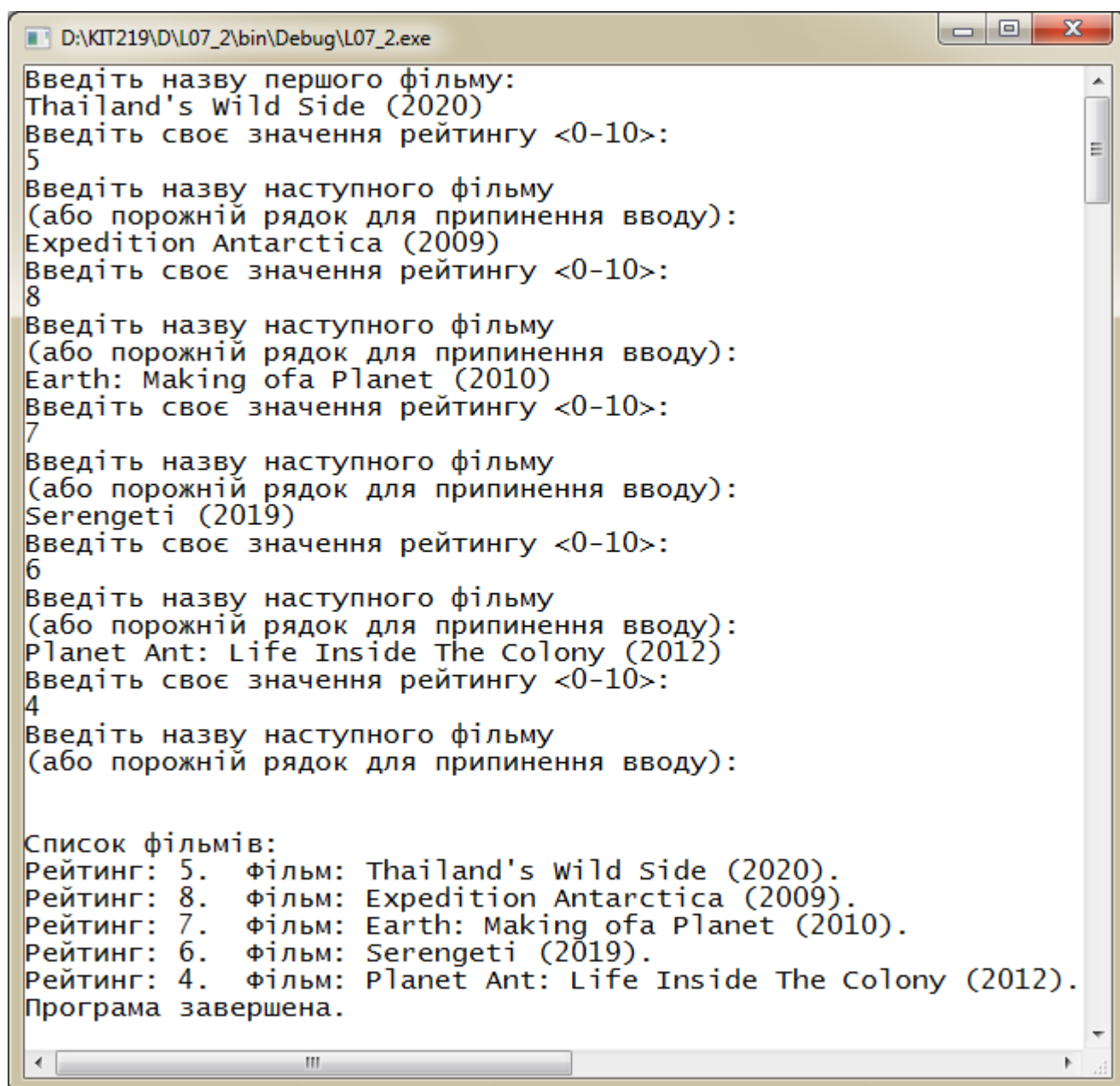


Рисунок 1.6 – Результат виконання програми, яка представляє дані у вигляді зв’язного списку

#### 4 Відображення списку

Ідея полягає в тому, щоб розпочати зі встановлення вказівника **current** для посилання на першу структуру. Оскільки вказівник на заголовок **head** вже вказує куди треба, наступного коду цілком достатньо:

```
current = head;
```

Потім за допомогою форми запису з вказівником можна звернутися до членів цієї структури:

```
printf("Рейтинг: %d. Фільм: %s. \n",
       current->rating, current->title);
```

Далі вказівник **current** перевстановлюється для посилення на наступну структуру в списку. Ця інформація зберігається в члені **next** структури, тому задача вирішується за допомогою такого коду:

```
current = current->next;
```

Після цього весь процес необхідно повторити. Після відображення останнього елемента в списку вказівник **current** буде встановлений в **NULL**, оскільки це значення члену **next** останньої структури. Даною обставиною можна скористатися для припинення виводу. Фрагмент коду, що застосовується для відображення списку, виглядає наступним чином:

```
while(current != NULL)
{
    printf("Рейтинг: %d. Фільм: %s. \n",
           current->rating, current->title);
    current = current->next;
}
```

Чому б для переміщення по списку не скористатися **head** замість того, щоб створювати новий вказівник (**current**)? Причина полягає в тому, що це призвело б до зміни значення **head**, і програма втратила б можливість знаходити початок списку.

## 5 Створення списку

Створення списку передбачає виконання трьох основних дій:

- використання функції **malloc()** для виділення достатнього простору під зберігання структури;
- збереження адреси структури;
- копіювання до структури коректної інформації.

Не має сенсу створювати структуру, якщо вона поки непотрібна, тому для прийому від користувача інформації про назву фільму в програмі застосовується тимчасове сховище (масив **input**). Якщо користувач відтворює за допомогою клавіатури символ **EOF** або вводить порожній рядок, цикл вводу завершується:

```
while(s_gets(input, TSIZE) != NULL && input[0] != '\0')
```



За наявності введених даних програма запитує простір для структури і присвоює її адресу змінній типу вказівника **current**:

```
current = (struct Film *) malloc(sizeof(struct Film));
```

Адреса самої першої структури повинна бути збережена у змінній типу вказівника **head**. Адреса кожної наступної структури повинна зберігатися в члені **next** попередньої структури. Таким чином, програмі потрібен спосіб для з'ясування того, чи є поточна структура першою. Простіше за все вирішити цю задачу, ініціалізуючи вказівник **head** значенням **NULL** на початку програми. Потім у програмі можна використовувати значення вказівника **head** для прийняття рішення про подальші дії:

```
if(head == NULL)                // перша структура
    head = current;
else                            // наступні структури
    prev->next = current;
```

В цьому коді **prev** – вказівник на структуру, яка була виділена минулого разу. Далі знадобиться встановити члени структури у відповідні значення. Зокрема, член **next** повинен бути встановлений в **NULL** для вказівки на те, що поточна структура є останньою в списку. Ви повинні скопіювати назву фільму з масиву **input** до члену **title** і отримати значення для члену **rating**. Ці дії виконує наступний код:

```
current->next = NULL;
strcpy(current->title, input);
puts("Введіть своє значення рейтингу <0-10>:");
scanf("%d", &current->rating);
```

Оскільки виклик **s\_gets()** обмежує дані, що вводяться, межею в **TSIZE-1** символів, рядок у масиві **input** поміститься до члену **title**, тому цілком безпечно застосовувати функцію **strcpy()**.

Нарешті, ви повинні підготувати програму до наступного циклу вводу. Зокрема, вказівник **prev** необхідно встановити так, щоб він посилався на поточну структуру, оскільки після вводу назви наступного фільму та розподілу наступної структури поточна структура стане попередньою. Програма встановлює цей вказівник наприкінці циклу:

```
prev = current;
```

## 6 Звільнення пам'яті, яка була зайнята списком

В багатьох середовищах програма при своєму завершенні звільнює пам'ять, що була виділена за допомогою функції `malloc()`, але краще, щоб ви звикли кожен виклик функції `malloc()` супроводжувати викликом `free()`. Таким чином, програма очищує використану пам'ять з застосуванням функції `free()` до кожної виділеної структури:

```
current = head;
while(current != NULL)
{
    free(current);
    current = current->next;
}
```

## 7 Додаткові міркування

Можливості розглянутої програми є дещо обмеженими. Наприклад, в ній відсутня перевірка, чи вдалося функції `malloc()` знайти потрібну пам'ять, і вона позбавлена будь-яких засобів для видалення елементів зі списку. Проте, такі недоліки можуть бути усунені. Скажімо, можна додати код, який перевіряє, чи дорівнює нулю (`NULL` – ознака невдачі в отриманні бажаної пам'яті) значення, що повертається `malloc()`. Якщо програма потребує видалення записів, в ній можна передбачити додатковий код.

Такий спеціалізований підхід до вирішення проблем і додаванню функціональних можливостей в міру необхідності не завжди є найкращим стилем програмування. З іншого боку, як правило, не вдається вгадати абсолютно все, що буде потрібно програмі. У міру зростання масштабу проектів модель завчасного планування всіх необхідних функціональних засобів стає все менш реалістичною. Було виявлено, що найуспішнішими виявлялися ті великі програми, які поетапно розвивалися від невеликих успішних програм.

З огляду на те, що плани можуть переглядатися, має сенс розробляти початкові ідеї в такий спосіб, що спрощує подальшу модифікацію. Розглянутий приклад не дотримується цього принципу. Так, в ньому проявляється тенденція до змішування деталей кодування та концептуальної моделі. Наприклад, в цьому

кодi концептуальна модель полягає в тому, що елементи додаються до списку. Програма затiнює цей iнтерфейс, виносячи на переднiй план такi деталi, як `malloc()` i вказiвник `current->next`. Дуже бажано, якби ви змогли писати програму в стилi, який робить очевидним те, що ви додаєте елемент до списку, i одночасно приховує такi допомiжнi дiї, як виклик функцiй керування пам'яттю i встановлення вказiвникiв. Вiдокремлення призначеного для користувача iнтерфейсу вiд деталей реалiзацiї спростить розумiння та оновлення програми. Згаданих цiлей можна досягти, створивши програму заново. Розпочавши розробку з нуля, ви можете досягти таких цiлей.