**1. Cấu trúc Node**

typedef struct Node {

int data;

struct Node\* next;

} Node;

* **Giải thích**:
  + data: Lưu giá trị số nguyên của nút (ví dụ: 10, 20, 30).
  + next: Con trỏ trỏ tới nút tiếp theo trong danh sách, hoặc NULL nếu là nút cuối.
  + typedef: Đặt tên alias là Node để viết code ngắn gọn, không cần viết struct Node mỗi lần sử dụng.
* **Tại sao**:
  + Danh sách liên kết đơn cần mỗi nút lưu trữ một giá trị và một liên kết tới nút tiếp theo để tạo thành chuỗi.
  + next cho phép liên kết các nút, tạo cấu trúc động có thể đảo ngược bằng cách thay đổi con trỏ.
* **Ví dụ**:
  + Với danh sách [10 -> 20 -> 30]:
    - Nút đầu: data = 10, next trỏ tới nút có data = 20.
    - Nút cuối: data = 30, next = NULL.

**2. Hàm create\_node**

Node\* create\_node(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: Node\* new\_node = (Node\*)malloc(sizeof(Node));
     + Cấp phát bộ nhớ động cho một nút mới bằng malloc.
     + sizeof(Node): Lấy kích thước của cấu trúc Node (thường là kích thước của int + con trỏ next).
     + (Node\*): Ép kiểu con trỏ trả về từ malloc thành Node\* để phù hợp với kiểu dữ liệu.
  2. **Dòng 2**: new\_node->data = data;
     + Gán giá trị data (tham số đầu vào) vào trường data của nút mới.
  3. **Dòng 3**: new\_node->next = NULL;
     + Đặt con trỏ next của nút mới về NULL, vì nút mới chưa liên kết với nút nào khác.
  4. **Dòng 4**: return new\_node;
     + Trả về con trỏ tới nút mới vừa tạo.
* **Tại sao**:
  1. Cần một hàm riêng để tạo nút mới, đảm bảo mỗi nút được khởi tạo đúng (gán data, đặt next = NULL).
  2. Sử dụng malloc để cấp phát bộ nhớ động trên heap, vì danh sách liên kết cần lưu trữ nút ngoài phạm vi hàm.
  3. Đặt next = NULL để tránh lỗi truy cập bộ nhớ ngẫu nhiên (undefined behavior).
* **Ví dụ**:
  1. Gọi create\_node(10):
     + Tạo nút với data = 10, next = NULL.
     + Trả về con trỏ tới nút này.

**3. Hàm insert\_at\_tail**

void insert\_at\_tail(Node\*\* head\_ref, int data) {

Node\* new\_node = create\_node(data);

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

Node\* current = \*head\_ref;

while (current->next != NULL) {

current = current->next;

}

current->next = new\_node;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void insert\_at\_tail(Node\*\* head\_ref, int data)
     + Nhận head\_ref (con trỏ đến con trỏ Node\*) để thay đổi con trỏ head trong hàm gọi nếu danh sách rỗng.
     + data: Giá trị cần chèn vào nút mới.
  2. **Dòng 2**: Node\* new\_node = create\_node(data);
     + Gọi create\_node để tạo một nút mới với giá trị data.
  3. **Dòng 3-5**: if (\*head\_ref == NULL) { \*head\_ref = new\_node; return; }
     + Nếu danh sách rỗng (head == NULL), đặt head trỏ tới nút mới và thoát hàm.
  4. **Dòng 6**: Node\* current = \*head\_ref;
     + Tạo con trỏ current trỏ tới nút đầu danh sách để duyệt tới nút cuối.
  5. **Dòng 7-8**: while (current->next != NULL) { current = current->next; }
     + Duyệt current tới nút cuối (nút có next == NULL).
  6. **Dòng 9**: current->next = new\_node;
     + Liên kết nút cuối với nút mới, thêm nút mới vào cuối danh sách.
* **Tại sao**:
  1. Chèn vào cuối danh sách giữ nguyên thứ tự nhập của người dùng (ví dụ: nhập 10, 20, 30 tạo [10 -> 20 -> 30]).
  2. Dùng Node\*\* head\_ref để thay đổi head trong trường hợp danh sách rỗng.
  3. Phương pháp này đảm bảo danh sách được tạo đúng theo input, phù hợp với yêu cầu nhập danh sách bất kỳ.
* **Ví dụ**:
  1. Ban đầu: head = NULL.
  2. Gọi insert\_at\_tail(&head, 10): [10].
  3. Gọi insert\_at\_tail(&head, 20): [10 -> 20].
  4. Gọi insert\_at\_tail(&head, 30): [10 -> 20 -> 30].

**4. Hàm print\_list**

void print\_list(Node\* head) {

if (head == NULL) {

printf("Danh sach rong\n");

return;

}

Node\* current = head;

while (current != NULL) {

printf("%d", current->data);

if (current->next != NULL) {

printf(" -> ");

}

current = current->next;

}

printf("\n");

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void print\_list(Node\* head)
     + Nhận con trỏ head trỏ tới nút đầu của danh sách.
  2. **Dòng 2-4**: if (head == NULL) { printf("Danh sach rong\n"); return; }
     + Kiểm tra nếu danh sách rỗng (head == NULL), in thông báo và thoát hàm.
  3. **Dòng 5**: Node\* current = head;
     + Tạo con trỏ current để duyệt danh sách, bắt đầu từ nút đầu.
  4. **Dòng 6-10**: while (current != NULL)
     + In giá trị data của nút hiện tại (current->data).
     + Nếu có nút tiếp theo (current->next != NULL), in " -> " để biểu thị liên kết.
     + Di chuyển current tới nút tiếp theo (current->next).
  5. **Dòng 11**: printf("\n");
     + In ký tự xuống dòng để định dạng đầu ra.
* **Tại sao**:
  1. Hàm này in danh sách để người dùng thấy cấu trúc và nội dung trước/sau khi đảo ngược.
  2. Kiểm tra head == NULL để xử lý trường hợp danh sách rỗng, tránh lỗi truy cập bộ nhớ.
  3. Dấu " -> " giúp hình dung rõ ràng các liên kết giữa các nút.
  4. Không cần giới hạn số lần in (như trong bài kiểm tra vòng) vì danh sách không có vòng.
* **Ví dụ**:
  1. Với danh sách [10 -> 20 -> 30 -> 40]:
     + In: 10 -> 20 -> 30 -> 40.
  2. Với danh sách rỗng:
     + In: Danh sach rong.

**5. Hàm reverse\_list**

void reverse\_list(Node\*\* head\_ref) {

Node\* prev = NULL;

Node\* current = \*head\_ref;

Node\* next = NULL;

while (current != NULL) {

next = current->next; *// Lưu nút tiếp theo*

current->next = prev; *// Đảo ngược liên kết*

prev = current; *// Di chuyển prev lên*

current = next; *// Di chuyển current lên*

}

\*head\_ref = prev; *// Cập nhật head*

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void reverse\_list(Node\*\* head\_ref)
     + Nhận head\_ref (con trỏ đến con trỏ Node\*) để thay đổi con trỏ head trong hàm gọi.
  2. **Dòng 2**: Node\* prev = NULL;
     + Khởi tạo con trỏ prev là NULL, đại diện cho nút trước nút hiện tại (ban đầu không có nút trước).
  3. **Dòng 3**: Node\* current = \*head\_ref;
     + Khởi tạo current trỏ tới nút đầu danh sách (\*head\_ref là head).
  4. **Dòng 4**: Node\* next = NULL;
     + Khởi tạo con trỏ next để lưu nút tiếp theo trong danh sách.
  5. **Dòng 5-9**: while (current != NULL)
     + next = current->next: Lưu con trỏ tới nút tiếp theo để không mất liên kết khi thay đổi current->next.
     + current->next = prev: Đảo ngược liên kết bằng cách cho current trỏ về prev thay vì nút tiếp theo.
     + prev = current: Di chuyển prev lên để trỏ tới nút hiện tại.
     + current = next: Di chuyển current tới nút tiếp theo (đã lưu trong next).
  6. **Dòng 10**: \*head\_ref = prev;
     + Cập nhật head để trỏ tới nút cuối cùng của danh sách ban đầu (nay là nút đầu sau khi đảo ngược).
* **Tại sao**:
  1. Dùng Node\*\* head\_ref để thay đổi con trỏ head trong hàm gọi, vì sau khi đảo ngược, nút đầu sẽ là nút cuối ban đầu.
  2. Sử dụng ba con trỏ (prev, current, next) để đảo ngược liên kết mà không mất kết nối giữa các nút.
  3. Thuật toán này hiệu quả với độ phức tạp O(n) về thời gian và O(1) về không gian (không cần cấp phát bộ nhớ mới).
* **Ví dụ** (với danh sách [10 -> 20 -> 30 -> 40]):
  1. **Ban đầu**: head -> [10 -> 20 -> 30 -> 40], prev = NULL, current -> 10, next = NULL.
  2. **Lần 1**:
     + next = 20 (lưu nút 20).
     + current->next = NULL (10 trỏ về NULL).
     + prev = 10, current = 20.
     + Danh sách: [10], [20 -> 30 -> 40].
  3. **Lần 2**:
     + next = 30.
     + current->next = 10 (20 trỏ về 10).
     + prev = 20, current = 30.
     + Danh sách: [20 -> 10], [30 -> 40].
  4. **Lần 3**:
     + next = 40.
     + current->next = 20 (30 trỏ về 20).
     + prev = 30, current = 40.
     + Danh sách: [30 -> 20 -> 10], [40].
  5. **Lần 4**:
     + next = NULL.
     + current->next = 30 (40 trỏ về 30).
     + prev = 40, current = NULL.
     + Danh sách: [40 -> 30 -> 20 -> 10].
  6. **Cuối cùng**: \*head\_ref = prev (head trỏ tới nút 40).

**6. Hàm free\_list**

void free\_list(Node\*\* head\_ref) {

Node\* current = \*head\_ref;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

\*head\_ref = NULL;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void free\_list(Node\*\* head\_ref)
     + Nhận head\_ref (con trỏ đến con trỏ Node\*) để thay đổi con trỏ head trong hàm gọi.
  2. **Dòng 2**: Node\* current = \*head\_ref;
     + Khởi tạo current trỏ tới nút đầu danh sách (\*head\_ref là head).
  3. **Dòng 3-6**: while (current != NULL)
     + Node\* temp = current: Lưu con trỏ tới nút hiện tại để giải phóng.
     + current = current->next: Di chuyển current tới nút tiếp theo.
     + free(temp): Giải phóng bộ nhớ của nút được lưu trong temp.
  4. **Dòng 7**: \*head\_ref = NULL;
     + Đặt head về NULL để biểu thị danh sách rỗng.
* **Tại sao**:
  1. Mỗi nút được tạo bằng malloc cần được giải phóng bằng free để tránh rò rỉ bộ nhớ (memory leak).
  2. Dùng temp để lưu current trước khi di chuyển, vì sau khi gọi free(temp), truy cập current->next sẽ gây lỗi (undefined behavior).
  3. Dùng Node\*\* head\_ref để đặt head = NULL trong hàm gọi, đảm bảo danh sách được đánh dấu rỗng.
* **Ví dụ** (với danh sách [40 -> 30 -> 20 -> 10]):
  1. **Ban đầu**: current -> 40.
  2. **Lần 1**: temp = 40, current = 30, giải phóng nút 40.
  3. **Lần 2**: temp = 30, current = 20, giải phóng nút 30.
  4. **Lần 3**: temp = 20, current = 10, giải phóng nút 20.
  5. **Lần 4**: temp = 10, current = NULL, giải phóng nút 10.
  6. **Cuối cùng**: \*head\_ref = NULL.

**7. Hàm main**

int main() {

Node\* head = NULL;

int n, value;

*// Nhập số lượng phần tử*

printf("Nhap so luong phan tu cua danh sach: ");

scanf("%d", &n);

if (n < 0) {

printf("So luong phan tu khong hop le!\n");

return 1;

}

*// Nhập các phần tử*

printf("Nhap %d gia tri cho danh sach:\n", n);

for (int i = 0; i < n; i++) {

scanf("%d", &value);

insert\_at\_tail(&head, value);

}

*// In danh sách ban đầu*

printf("Danh sach vua nhap: ");

print\_list(head);

*// Đảo ngược danh sách*

reverse\_list(&head);

printf("Danh sach sau khi dao nguoc: ");

print\_list(head);

*// Giải phóng danh sách*

free\_list(&head);

printf("Danh sach sau khi giai phong: ");

print\_list(head);

return 0;

}

* **Giải thích từng phần**:
  1. **Dòng 2-3**: Khởi tạo biến
     + Node\* head = NULL: Khởi tạo danh sách rỗng.
     + int n, value: Biến để lưu số lượng phần tử và giá trị nhập từ người dùng.
     + **Tại sao**: Cần khởi tạo head = NULL để bắt đầu với danh sách rỗng. Biến n và value hỗ trợ nhập liệu từ người dùng.
  2. **Dòng 5-10**: Nhập số lượng phần tử
     + In thông báo yêu cầu nhập n (số lượng phần tử).
     + Dùng scanf để đọc n.
     + Kiểm tra n < 0 để đảm bảo số lượng hợp lệ, nếu không hợp lệ thì in lỗi và thoát chương trình.
     + **Tại sao**: Cho phép người dùng xác định kích thước danh sách. Kiểm tra lỗi để tránh input không hợp lệ (ví dụ: số lượng âm).
  3. **Dòng 12-16**: Nhập các phần tử
     + In thông báo yêu cầu nhập n giá trị.
     + Dùng vòng lặp for để đọc n giá trị số nguyên từ người dùng.
     + Gọi insert\_at\_tail để thêm từng giá trị vào cuối danh sách.
     + **Tại sao**: Giữ thứ tự nhập của người dùng, tạo danh sách linh hoạt theo input (ví dụ: nhập 10, 20, 30 tạo [10 -> 20 -> 30]).
  4. **Dòng 18-19**: In danh sách ban đầu
     + In thông báo và gọi print\_list để in danh sách vừa nhập.
     + **Tại sao**: Xác nhận danh sách được tạo đúng theo input của người dùng trước khi đảo ngược.
  5. **Dòng 21-22**: Đảo ngược danh sách
     + Gọi reverse\_list(&head) để đảo ngược danh sách.
     + In thông báo và gọi print\_list để in danh sách sau khi đảo ngược.
     + **Tại sao**: Đáp ứng yêu cầu đảo ngược danh sách và hiển thị kết quả để kiểm tra (ví dụ: [10 -> 20 -> 30] thành [30 -> 20 -> 10]).
  6. **Dòng 24-25**: Giải phóng danh sách
     + Gọi free\_list(&head) để giải phóng bộ nhớ của danh sách.
     + In thông báo và gọi print\_list để xác nhận danh sách rỗng.
     + **Tại sao**: Giải phóng bộ nhớ là thực hành tốt trong C để tránh rò rỉ bộ nhớ. In danh sách rỗng để xác nhận giải phóng thành công.
  7. **Dòng 27**: return 0;
     + Kết thúc chương trình với mã trả về 0, báo hiệu chương trình chạy thành công.
* **Tại sao cấu trúc này**:
  1. Đáp ứng yêu cầu nhập danh sách bất kỳ từ người dùng, thay vì cố định danh sách [10, 20, 30, 40].
  2. In danh sách trước và sau khi đảo ngược để kiểm tra kết quả rõ ràng.
  3. Giải phóng bộ nhớ để đảm bảo chương trình an toàn và đúng chuẩn.
  4. Giao diện nhập liệu thân thiện, dễ sử dụng, với xử lý lỗi cơ bản (kiểm tra n < 0).

**Kết quả chạy**

**Trường hợp 1: Nhập danh sách [10, 20, 30, 40]**

Nhap so luong phan tu cua danh sach: 4

Nhap 4 gia tri cho danh sach:

10 20 30 40

Danh sach vua nhap: 10 -> 20 -> 30 -> 40

Danh sach sau khi dao nguoc: 40 -> 30 -> 20 -> 10

Danh sach sau khi giai phong: Danh sach rong

**Trường hợp 2: Nhập danh sách rỗng**

Nhap so luong phan tu cua danh sach: 0

Nhap 0 gia tri cho danh sach:

Danh sach vua nhap: Danh sach rong

Danh sach sau khi dao nguoc: Danh sach rong

Danh sach sau khi giai phong: Danh sach rong

**Trường hợp 3: Nhập danh sách một phần tử [5]**

Nhap so luong phan tu cua danh sach: 1

Nhap 1 gia tri cho danh sach:

5

Danh sach vua nhap: 5

Danh sach sau khi dao nguoc: 5

Danh sach sau khi giai phong: Danh sach rong

**Hình ảnh minh họa**

**Danh sách ban đầu (nhập 10, 20, 30, 40):**

head -> [10] -> [20] -> [30] -> [40] -> NULL

**Quá trình đảo ngược (reverse\_list):**

1. **Lần 1**:
   * prev = NULL, current -> 10, next -> 20.
   * next = 20, current->next = NULL, prev = 10, current = 20.
   * Danh sách: [10], [20 -> 30 -> 40].
2. **Lần 2**:
   * prev -> 10, current -> 20, next -> 30.
   * next = 30, current->next = 10, prev = 20, current = 30.
   * Danh sách: [20 -> 10], [30 -> 40].
3. **Lần 3**:
   * prev -> 20, current -> 30, next -> 40.
   * next = 40, current->next = 20, prev = 30, current = 40.
   * Danh sách: [30 -> 20 -> 10], [40].
4. **Lần 4**:
   * prev -> 30, current -> 40, next = NULL.
   * next = NULL, current->next = 30, prev = 40, current = NULL.
   * Danh sách: [40 -> 30 -> 20 -> 10].
5. **Cuối cùng**: \*head\_ref = prev (head trỏ tới nút 40).

**Danh sách sau khi đảo ngược:**

head -> [40] -> [30] -> [20] -> [10] -> NULL

**Quá trình giải phóng (free\_list):**

1. **Lần 1**: Giải phóng nút 40, head -> [30 -> 20 -> 10].
2. **Lần 2**: Giải phóng nút 30, head -> [20 -> 10].
3. **Lần 3**: Giải phóng nút 20, head -> [10].
4. **Lần 4**: Giải phóng nút 10, head = NULL.

**Danh sách sau khi giải phóng:**

head -> NULL

**Notes**

1. **Hiểu con trỏ trong C**:
   * Node\*: Con trỏ tới một nút, chứa địa chỉ của một Node.
   * Node\*\*: Con trỏ tới con trỏ Node\*, dùng để thay đổi head trong hàm gọi (ví dụ: trong reverse\_list, insert\_at\_tail, free\_list).
   * **Ví dụ**: head\_ref chứa địa chỉ của head, \*head\_ref là head.
2. **Hiểu thuật toán đảo ngược**:
   * Vẽ danh sách trên giấy (ví dụ: [10 -> 20 -> 30]).
   * Theo dõi ba con trỏ prev, current, next qua mỗi bước để thấy cách liên kết thay đổi.
   * **Ví dụ**: current->next = prev làm cho nút hiện tại trỏ về nút trước, tạo liên kết ngược.
3. **Xử lý trường hợp đặc biệt**:
   * Kiểm tra danh sách rỗng (head == NULL) trong print\_list và reverse\_list để tránh lỗi truy cập bộ nhớ.
   * Kiểm tra n < 0 trong main để xử lý input không hợp lệ.
   * Danh sách một nút hoặc rỗng vẫn hoạt động đúng trong reverse\_list (không cần thay đổi liên kết).
4. **Debug**:
   * Nếu gặp lỗi, thêm printf("%p\n", node) để in địa chỉ của các nút, kiểm tra liên kết có đúng không.
   * Trong reverse\_list, thêm printf để theo dõi giá trị của prev, current, next qua mỗi bước.
   * **Ví dụ**:

printf("prev: %p, current: %p, next: %p\n", prev, current, next);

1. **Tại sao giải phóng bộ nhớ?**:
   * Mỗi nút được tạo bằng malloc chiếm bộ nhớ heap. Nếu không giải phóng, chương trình sẽ gây rò rỉ bộ nhớ, đặc biệt với danh sách lớn.
   * Đặt head = NULL sau khi giải phóng giúp tránh lỗi truy cập bộ nhớ đã giải phóng (dangling pointer).
2. **Tại sao dùng insert\_at\_tail thay vì insert\_at\_head?**:
   * insert\_at\_tail giữ thứ tự nhập của người dùng (nhập 10, 20, 30 tạo [10 -> 20 -> 30]).
   * insert\_at\_head sẽ tạo danh sách ngược (nhập 10, 20, 30 tạo [30 -> 20 -> 10]), không phù hợp với yêu cầu nhập danh sách bất kỳ.