

Họ và tên : Lý Gia Huy
BTVN : session 6
Lớp : CNTT 5

BÀI TẬP 1 :

[BÀI TẬP] REFACTOR CODE ĐỂ DỄ ĐỌC

1. Mục tiêu

Rèn kỹ năng yêu cầu AI refactor code cũ thành phiên bản dễ đọc, chuẩn hóa, dễ bảo trì.

2. Code gốc (chưa refactor, khó đọc)

```
#include <stdio.h>

int f(int n){
    int s=0;
    for(int i=2;i<=n;i++){
        int c=0;
        for(int j=1;j<=i;j++){
            if(i%j==0) c++;
        }
        if(c==2) s+=i;
    }
    return s;
}

int main(){
    int x;
    scanf("%d",&x);
    printf("%d",f(x));
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Nhận xét code gốc

- Tên hàm và biến không rõ nghĩa (f, s, c, x)
- Không có chú thích
- Kiểm tra số nguyên tố chưa tối ưu
- Một hàm chứa quá nhiều logic

3. Prompt dùng để refactor

“Hãy refactor đoạn code C dưới đây thành phiên bản dễ đọc, chuẩn hóa, đặt tên biến rõ nghĩa, tách hàm hợp lý và thêm chú thích.”

4. Code sau khi refactor (dễ đọc, chuẩn hóa)

```
#include <stdio.h>
```

```
// Hàm kiểm tra một số có phải là số nguyên tố hay không
```

```
int isPrime(int number) {
```

```
    if (number < 2)
```

```
        return 0;
```

```
    for (int i = 2; i * i <= number; i++) {
```

```
        if (number % i == 0)
```

```
            return 0;
```

```
    }
```

```
    return 1;
```

```
}
```

```
// Hàm tính tổng các số nguyên tố từ 1 đến n
```

```
int sumOfPrimes(int n) {
```

```
    int sum = 0;
```

```
    for (int i = 2; i <= n; i++) {
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
        if (isPrime(i)) {
            sum += i;
        }
    }
    return sum;
}

int main() {
    int n;
    scanf("%d", &n);

    printf("%d", sumOfPrimes(n));
    return 0;
}
```

5. So sánh code trước và sau khi refactor

Tiêu chí	Code gốc	Code refactor
Tên biến, tên hàm	Không rõ nghĩa	Rõ nghĩa
Cấu trúc	Dồn vào một hàm	Tách hàm rõ ràng
Dễ đọc	Khó hiểu	Dễ hiểu
Hiệu năng	Thấp	Cao hơn
Chú thích	Không có	Có
Khả năng bảo trì	Kém	Tốt

6. Điểm cải thiện chính

- Đặt tên biến và hàm có ý nghĩa
- Tách chức năng kiểm tra số nguyên tố thành hàm riêng
- Thuật toán kiểm tra số nguyên tố được tối ưu
- Code rõ ràng, dễ kiểm tra và mở rộng

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

7. Kết luận

Sau khi refactor, chương trình trở nên dễ đọc hơn, logic rõ ràng hơn và đạt chuẩn viết code C.

Việc refactor giúp chương trình dễ bảo trì và cải thiện hiệu suất.

BÀI TẬP 2 :

[BÀI TẬP] VIẾT HÀM THEO YÊU CẦU RÕ RÀNG

1. Mục tiêu

Rèn kỹ năng viết prompt mô tả yêu cầu hàm một cách rõ ràng, đủ chi tiết để AI sinh ra đoạn code đúng ý, dễ đọc và đúng logic.

2. Hàm được chọn

Hàm **kiểm tra số nguyên tố**.

3. Phiên bản Prompt 1 (ngắn, ít ràng buộc)

Prompt 1:

“Viết hàm C kiểm tra một số có phải số nguyên tố hay không.”

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Code AI sinh ra (theo Prompt 1)

```
int isPrime(int n) {  
    if (n < 2) return 0;  
    for (int i = 2; i < n; i++) {  
        if (n % i == 0)  
            return 0;  
    }  
    return 1;  
}
```

Nhận xét

- Code đúng chức năng
- Không có chú thích
- Chưa tối ưu (vòng lặp chạy đến $n - 1$)
- Không nói rõ kiểu dữ liệu đầu vào

4. Phiên bản Prompt 2 (rõ ràng, đầy đủ yêu cầu)

Prompt 2:

“Viết hàm C kiểm tra số nguyên tố.

Yêu cầu:

- Đầu vào là số nguyên dương n
- Trả về 1 nếu là số nguyên tố, 0 nếu không
- Chỉ kiểm tra ước số đến căn bậc hai của n
- Có chú thích giải thích từng bước trong hàm”

Code AI sinh ra (theo Prompt 2)

```
// Hàm kiểm tra số nguyên tố  
// Trả về 1 nếu n là số nguyên tố, ngược lại trả về 0
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
int isPrime(int n) {  
    // Số nhỏ hơn 2 không phải số nguyên tố  
    if (n < 2)  
        return 0;  
  
    // Kiểm tra các ước từ 2 đến căn bậc hai của n  
    for (int i = 2; i * i <= n; i++) {  
        if (n % i == 0)  
            return 0;  
    }  
  
    return 1;  
}
```

Nhận xét

- Code rõ ràng, dễ đọc
- Có chú thích giải thích từng bước
- Thuật toán được tối ưu
- Đúng yêu cầu đề bài

5. So sánh kết quả AI sinh code

Tiêu chí	Prompt 1 (ngắn)	Prompt 2 (rõ ràng)
Mô tả yêu cầu	Chung chung	Cụ thể, chi tiết
Chú thích	Không có	Có
Hiệu năng	Chưa tối ưu	Tối ưu
Độ dễ đọc	Trung bình	Cao
Khả năng dừng lại	Hạn chế	Tốt

6. Điểm khác biệt rút ra

- Prompt càng rõ ràng thì code AI sinh ra càng đúng ý

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

- Việc nêu rõ ràng đầu vào, đầu ra và ràng buộc giúp tránh code kém chất lượng
- Prompt ngắn thường cho kết quả đúng nhưng chưa tối ưu
- Prompt chi tiết giúp AI sinh code chuẩn và dễ bảo trì hơn

7. Kết luận

Viết prompt rõ ràng là yếu tố quan trọng để AI sinh ra code chất lượng.

Prompt càng đầy đủ yêu cầu thì kết quả càng chính xác, dễ đọc và phù hợp với mục tiêu học tập.

BÀI TẬP 3 :

[BÀI TẬP] THÊM YÊU CẦU ĐOẠN MÃ DỄ ĐỌC

1. Mô tả bài toán chung

Yêu cầu hàm:

Viết hàm C tính **tổng các số lẻ nhỏ hơn n** (n là số nguyên dương).

2. Prompt 1 – Prompt đơn giản (không yêu cầu dễ đọc)

Prompt 1

Viết hàm C tính tổng các số lẻ nhỏ hơn n.

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Code sinh ra từ Prompt 1

```
int sumOdd(int n) {  
    int s = 0;  
    for (int i = 1; i < n; i++) {  
        if (i % 2 != 0) {  
            s += i;  
        }  
    }  
    return s;  
}
```

3. Prompt 2 – Prompt có yêu cầu code dễ đọc, có chú thích

Prompt 2

Viết hàm C tính tổng các số lẻ nhỏ hơn n.

Yêu cầu:

- Đặt tên biến rõ ràng, dễ hiểu
- Có chú thích giải thích từng bước trong hàm
- Code viết rõ ràng, dễ đọc cho sinh viên năm nhất

Code sinh ra từ Prompt 2

```
// Hàm tính tổng các số lẻ nhỏ hơn n  
int calculateSumOfOddNumbers(int n) {  
    int sum = 0; // Biến lưu tổng các số lẻ  
  
    // Duyệt các số từ 1 đến n - 1  
    for (int currentNumber = 1; currentNumber < n; currentNumber++) {
```


Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
// Kiểm tra nếu currentNumber là số lẻ
if (currentNumber % 2 != 0) {
    sum += currentNumber; // Cộng số lẻ vào tổng
}

return sum; // Trả về kết quả
}
```

4. So sánh hai phiên bản code

Tiêu chí	Prompt 1	Prompt 2
Tên hàm	Ngắn, khó hiểu	Rõ nghĩa, mô tả đúng chức năng
Tên biến	s, i	sum, currentNumber
Chú thích	Không có	Có chú thích từng bước
Dễ đọc	Trung bình	Rất dễ hiểu
Phù hợp sinh viên	Chưa tốt	Rất phù hợp

5. Nhận xét – Điểm khác biệt & cải thiện

- Prompt 1 cho code **đúng chức năng** nhưng:
 - Khó đọc
 - Không thân thiện với người mới học
- Prompt 2:
 - Code **dài hơn nhưng dễ hiểu hơn**
 - Người khác đọc vào là hiểu ngay
 - Phù hợp để học, sửa lỗi, bảo trì

Kết luận:

Thêm yêu cầu “dễ đọc, có chú thích” trong prompt giúp AI sinh ra **code chất lượng cao hơn**, không chỉ đúng mà còn **dễ học – dễ dùng**.

BÀI TẬP 4 :

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

[BÀI TẬP] SỬA LỖI TỪ STACK TRACE

1. Mục tiêu

- Biết cách **copy lỗi/stack trace** khi chạy chương trình
- Viết **prompt đúng cách** để AI phân tích nguyên nhân
- Hiểu **vì sao lỗi xảy ra và cách sửa**

2. Đoạn code C bị lỗi (lỗi truy cập mảng ngoài phạm vi)

Code gốc (có lỗi)

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int sum = 0;

    for (int i = 0; i <= 5; i++) {
        sum += arr[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

3. Lỗi khi chạy chương trình

Khi chạy code, chương trình báo lỗi:

Segmentation fault (core dumped)

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

4. Prompt dùng để hỏi AI (copy đúng lỗi)

Prompt

Đây là lỗi khi chạy code C:

Segmentation fault (core dumped)

Dưới đây là đoạn code C của tôi (đã chạy và bị lỗi).

Hãy:

1. Phân tích nguyên nhân gây lỗi
2. Chỉ ra dòng code sai
3. Gợi ý cách sửa và viết lại code đúng

5. Phân tích lỗi (giải thích như AI)

Nguyên nhân lỗi

- Mảng arr có **5 phần tử**, chỉ số hợp lệ là: 0 → 4
- Vòng lặp:

```
for (int i = 0; i <= 5; i++)
```

cho phép i = 5

- Khi truy cập arr[5] → **truy cập vùng nhớ không hợp lệ**
- C gây lỗi **Segmentation fault**

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Dòng code gây lỗi

```
sum += arr[i]; // i = 5 -> arr[5] (sai)
```

6. Cách sửa lỗi

Cách 1 – Sửa điều kiện vòng lặp (đúng nhất)

```
for (int i = 0; i < 5; i++) {  
    sum += arr[i];  
}
```

7. Code sau khi sửa (đã đúng)

```
#include <stdio.h>
```

```
int main() {  
    int arr[5] = {1, 2, 3, 4, 5};  
    int sum = 0;  
  
    for (int i = 0; i < 5; i++) {  
        sum += arr[i];  
    }  
  
    printf("Sum = %d\n", sum);  
    return 0;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

8. So sánh trước và sau khi sửa

Tiêu chí	Code gốc	Code sau sửa
Điều kiện vòng lặp	<code>i <= 5</code>	<code>i < 5</code>
Truy cập mảng	Ngoài phạm vi	Đúng phạm vi
Kết quả	Lỗi runtime	Chạy bình thường

9. Kết luận – Bài học rút ra

- **Segmentation fault** thường do:
 - Truy cập mảng ngoài phạm vi
 - Dùng con trỏ sai
- Khi gặp lỗi:
 - Copy **nguyên văn lỗi**
 - Dán **toàn bộ code**
 - Hỏi AI phân tích **nguyên nhân + dòng lỗi**

BÀI TẬP 5 :

[BÀI TẬP] YÊU CẦU GIẢI THÍCH NGUYÊN NHÂN LỖI

1. Mục tiêu

- Hiểu vì sao **Segmentation fault** xảy ra, không chỉ sửa cho chạy
- Thấy rõ **prompt chi tiết** → câu trả lời chất lượng hơn

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

2. Đoạn code C bị lỗi (cùng 1 lỗi cho cả 2 prompt)

```
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};

    for (int i = 0; i <= 3; i++) {
        printf("%d\n", arr[i]);
    }

    return 0;
}
```

3. Prompt 1 – Chỉ yêu cầu “Sửa lỗi”

Prompt 1

Code C của tôi bị lỗi Segmentation fault.

Hãy sửa lỗi để chương trình chạy đúng.

Kết quả AI (tóm tắt)

- AI sửa vòng lặp:

```
for (int i = 0; i < 3; i++)
```

- Đưa code đã sửa
- **Không** giải thích rõ vì sao lỗi xảy ra

Người học **biết sửa**, nhưng **chưa hiểu bản chất**.

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

4. Prompt 2 – Yêu cầu giải thích nguyên nhân + minh họa bộ nhớ

Prompt 2

Code C của tôi bị lỗi Segmentation fault.

Hãy:

1. Giải thích chi tiết **tại sao lỗi xảy ra**
2. Minh họa bằng **sơ đồ bộ nhớ (stack / mảng)**
3. Sau đó mới **đưa ra cách sửa và code đúng**

5. Phân tích lỗi (theo Prompt 2)

Nguyên nhân lỗi

- Mảng arr có **3 phần tử**
- Chỉ số hợp lệ: 0, 1, 2
- Vòng lặp:

```
for (int i = 0; i <= 3; i++)
```

cho phép i = 3

- Khi truy cập arr[3]:
 - Truy cập vùng nhớ **không thuộc mảng**
 - Hệ điều hành chặn → **Segmentation fault**

Minh họa sơ đồ bộ nhớ (stack)

Bộ nhớ stack:

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
+-----+
| arr[0] | 10
+-----+
| arr[1] | 20
+-----+
| arr[2] | 30
+-----+
| ???   | ← arr[3] (ngoài phạm vi)
+-----+
```

arr[3] không tồn tại → lỗi.

6. Cách sửa lỗi (sau khi hiểu nguyên nhân)

```
#include <stdio.h>
```

```
int main() {
    int arr[3] = {10, 20, 30};

    for (int i = 0; i < 3; i++) {
        printf("%d\n", arr[i]);
    }

    return 0;
}
```

7. So sánh 2 prompt cho cùng 1 lỗi

Tiêu chí	Prompt 1: “Sửa lỗi”	Prompt 2: “Giải thích + sửa”
Có sửa được lỗi	Có	Có
Giải thích nguyên nhân	Không rõ	Rất chi tiết

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Hiểu bộ nhớ	Không	Có minh họa
Phù hợp học tập	Thấp	Rất cao
Nhớ lâu	Không	Có

8. Nhận xét – Prompt chi tiết giúp ích gì?

- Prompt chi tiết:
 - Giúp **hiểu bản chất lỗi**
 - Biết **vì sao Segmentation fault** xuất hiện
 - Tránh lặp lại lỗi trong tương lai
- Prompt ngắn:
 - Chỉ phù hợp khi cần sửa nhanh
 - Không tốt cho người mới học

Kết luận:

Khi học lập trình, nên hỏi AI theo kiểu

“Giải thích nguyên nhân + minh họa + sửa lỗi”

thay vì chỉ hỏi **“sửa lỗi”**

BÀI TẬP 6 :

[BÀI TẬP] TỐI ƯU CODE VỀ HIỆU NĂNG

(Phân tích lỗi runtime từ stack trace và sửa code)

1. Đoạn code ban đầu (có lỗi)

```
#include <stdio.h>
```

```
int main() {  
    int arr[3] = {1, 2, 3};  
    printf("%d\n", arr[3]); // Lỗi truy cập ngoài mảng  
    return 0;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

}

2. Lỗi khi chạy chương trình

Khi chạy, chương trình có thể gặp:

Segmentation fault (core dumped)

hoặc in ra giá trị rác (undefined behavior).

3. Prompt dùng để hỏi AI (đúng yêu cầu đề bài)

Prompt

Dưới đây là đoạn code C của tôi và thông báo lỗi khi chạy:

Segmentation fault (core dumped)

Hãy:

1. Phân tích nguyên nhân gây lỗi dựa trên code và stack trace
2. Giải thích lỗi bằng ngôn ngữ đơn giản, dễ hiểu cho sinh viên năm nhất
3. Đưa ra cách sửa hợp lý
4. Viết lại đoạn code đã sửa để chương trình chạy an toàn

4. Phân tích nguyên nhân lỗi

- Mảng arr được khai báo với kích thước **3 phần tử**
- Chỉ số hợp lệ của mảng: 0, 1, 2
- Dòng code:

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
printf("%d\n", arr[3]);
```

cố truy cập **phần tử thứ 4**, không tồn tại

- Điều này gây:
 - Truy cập vùng nhớ không hợp lệ
 - Hành vi không xác định (undefined behavior)
 - Có thể dẫn đến **Segmentation fault**

5. Giải thích lỗi bằng ngôn ngữ đơn giản

Có thể hình dung mảng như 3 ô nhớ liên tiếp:

arr[0]	arr[1]	arr[2]
1	2	3

Khi truy cập arr[3], chương trình cố đọc một ô nhớ **không thuộc mảng**, hệ điều hành không cho phép → chương trình bị lỗi.

6. Cách sửa lỗi (an toàn và đúng)

Cách 1 – Truy cập đúng chỉ số mảng

```
printf("%d\n", arr[2]);
```

7. Code sau khi sửa (chạy an toàn)

```
#include <stdio.h>
```

```
int main() {
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
int arr[3] = {1, 2, 3};

// Truy cập phần tử cuối cùng của mảng (chỉ số 2)
printf("%d\n", arr[2]);

return 0;
}
```

8. Nhận xét về “tối ưu hiệu năng”

- Sửa lỗi truy cập ngoài mảng:
 - Tránh crash chương trình
 - Tránh hành vi không xác định
- Code sau khi sửa:
 - Ổn định hơn
 - An toàn hơn
 - Không phát sinh lỗi runtime

Tối ưu hiệu năng bắt đầu từ việc code phải đúng và an toàn, sau đó mới tính đến tốc độ.

9. Kết luận – Bài học rút ra

- Khi gặp lỗi runtime:
 - Copy **nguyên văn lỗi**
 - Đưa **toàn bộ code**
 - Hỏi AI phân tích **nguyên nhân + sửa**
- Prompt càng chi tiết → lời giải càng dễ hiểu

BÀI TẬP 7 :

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

[BÀI TẬP] TỐI ƯU CODE VỀ HIỆU NĂNG

Chủ đề: Hàm kiểm tra số nguyên tố

1. Mô tả bài toán chung

Viết hàm C kiểm tra một số nguyên dương n có phải số nguyên tố hay không.

2. Prompt 1 – Prompt ngắn, không nhấn mạnh hiệu năng

Prompt 1

Viết hàm C kiểm tra số nguyên tố, đầu vào là số nguyên dương n .

Code AI sinh ra từ Prompt 1

```
int isPrime(int n) {  
    if (n < 2) {  
        return 0;  
    }  
  
    for (int i = 2; i < n; i++) {  
        if (n % i == 0) {  
            return 0;  
        }  
    }  
  
    return 1;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Nhận xét nhanh về code Prompt 1

- Đúng chức năng
- Dễ hiểu
- Chưa tối ưu hiệu năng:
 - Vòng lặp chạy từ 2 \rightarrow $n-1$
 - Khi n lớn \rightarrow chậm

3. Prompt 2 – Prompt chi tiết, yêu cầu chú thích & tối ưu

Prompt 2

Viết hàm C kiểm tra số nguyên tố với các yêu cầu sau:

- Đầu vào là số nguyên dương n
- Có chú thích giải thích từng bước
- Tên biến rõ ràng
- Tối ưu hiệu năng, không duyệt thừa
- Phù hợp cho sinh viên năm nhất đọc hiểu

Code AI sinh ra từ Prompt 2 (đã tối ưu)

```
#include <math.h>

// Hàm kiểm tra n có phải là số nguyên tố hay không
int isPrime(int n) {
    // Số nhỏ hơn 2 không phải số nguyên tố
    if (n < 2) {
        return 0;
    }
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
// Chỉ cần kiểm tra từ 2 đến căn bậc hai của n
for (int i = 2; i * i <= n; i++) {
    // Nếu chia hết thì không phải số nguyên tố
    if (n % i == 0) {
        return 0;
    }
}

// Nếu không chia hết cho số nào
return 1;
}
```

4. So sánh hai phiên bản code

Tiêu chí	Prompt 1 (ngắn)	Prompt 2 (chi tiết)
Đúng chức năng	Có	Có
Độ dễ đọc	Trung bình	Cao
Chú thích	Không	Có
Phạm vi vòng lặp	$2 \rightarrow n-1$	$2 \rightarrow \sqrt{n}$
Hiệu năng	Thấp hơn	Tốt hơn
Phù hợp học tập	Vừa	Rất tốt

5. Phân tích tối ưu hiệu năng

- Prompt 1:
 - Số vòng lặp $\approx n$
 - Khi $n = 1.000.000 \rightarrow \sim 1$ triệu lần lặp
- Prompt 2:
 - Số vòng lặp $\approx \sqrt{n}$
 - Khi $n = 1.000.000 \rightarrow \sim 1.000$ lần lặp

Giảm rất nhiều phép toán, chương trình chạy nhanh hơn rõ rệt.

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

6. Nhận xét – Prompt chi tiết giúp ích gì?

- Prompt chi tiết giúp AI:
 - Chủ động **tối ưu thuật toán**
 - Viết code **có chú thích, dễ đọc**
 - Tránh cách làm “ngây thơ” nhưng chậm
- Prompt ngắn:
 - Phù hợp khi cần code nhanh
 - Không phù hợp khi học hoặc cần hiệu năng

Kết luận:

Muốn code **nhANH + dễ hiểu**, prompt phải **nói rõ yêu cầu tối ưu**, không chỉ yêu cầu “viết hàm”.

7. Tổng kết bài tập

- Cùng một bài toán
- Chỉ khác cách viết prompt
- Kết quả code **khác nhau rõ rệt về hiệu năng và chất lượng**

BÀI TẬP 8 :

[BÀI TẬP] THÊM TEST CASE ĐỂ KIỂM THỬ CHỨC NĂNG

1. Mục tiêu

Hiểu rõ prompt .

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

2. Hàm cần kiểm thử

```
int isEven(int n) {  
    return n % 2 == 0;  
}
```

Chức năng:

- Trả về 1 nếu n là số chẵn
- Trả về 0 nếu n là số lẻ

3. Prompt dùng để hỏi AI

Prompt

Viết thêm các test case để kiểm thử hàm `isEven(int n)`

Yêu cầu:

- Có test case bình thường
- Có test case biên
- Có test case bất thường

Sau đó viết code test bằng `printf` để in kết quả.

4. Test case do AI đề xuất

Phân loại test của AI

Test bình thường

- $n = 2 \rightarrow$ chẵn
- $n = 3 \rightarrow$ lẻ
- $n = 10 \rightarrow$ chẵn

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Test biên

- $n = 0$
- $n = 1$

Test bất thường

- $n = -2$
- $n = -3$

Code test theo AI

```
#include <stdio.h>
```

```
int isEven(int n) {  
    return n % 2 == 0;  
}
```

```
int main() {  
    printf("isEven(2)  = %d\n", isEven(2));  
    printf("isEven(3)  = %d\n", isEven(3));  
    printf("isEven(10) = %d\n", isEven(10));  
  
    printf("isEven(0)  = %d\n", isEven(0));  
    printf("isEven(1)  = %d\n", isEven(1));  
  
    printf("isEven(-2) = %d\n", isEven(-2));  
    printf("isEven(-3) = %d\n", isEven(-3));  
  
    return 0;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

5. Test case tự nghĩ thêm (ngoài AI)

Lý do cần thêm test

- AI thường nghĩ test **cơ bản**
- Người học cần nghĩ đến:
 - Giá trị rất lớn
 - Giá trị sát biên kiểu dữ liệu
 - Hành vi toán học đặc biệt

Test case bổ sung

Test giá trị lớn

- $n = 1000000 \rightarrow$ chẵn
- $n = 999999 \rightarrow$ lẻ

Test biên kiểu dữ liệu

- $n = \text{INT_MAX}$
- $n = \text{INT_MIN}$

Code test mở rộng (AI + tự thêm)

```
#include <stdio.h>
#include <limits.h>

int isEven(int n) {
    return n % 2 == 0;
}

int main() {
    // Test bình thường
    printf("isEven(2)          = %d\n", isEven(2));
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
printf("isEven(3)          = %d\n", isEven(3));

// Test biên
printf("isEven(0)          = %d\n", isEven(0));
printf("isEven(1)          = %d\n", isEven(1));

// Test bất thường (số âm)
printf("isEven(-2)         = %d\n", isEven(-2));
printf("isEven(-3)         = %d\n", isEven(-3));

// Test giá trị lớn
printf("isEven(1000000)    = %d\n", isEven(1000000));
printf("isEven(999999)     = %d\n", isEven(999999));

// Test biên kiểu dữ liệu
printf("isEven(INT_MAX)    = %d\n", isEven(INT_MAX));
printf("isEven(INT_MIN)    = %d\n", isEven(INT_MIN));

return 0;
}
```

6. So sánh test case của AI và test tự nghĩ

Tiêu chí	AI đề xuất	Tự nghĩ thêm
Test cơ bản	Có	Có
Test biên	Có	Có
Test số âm	Có	Có
Test số rất lớn	Không	Có
Test biên kiểu dữ liệu	Không	Có
Tư duy kiểm thử	Cơ bản	Sâu hơn

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

7. Nhận xét – Bài học rút ra

- AI giúp:
 - Không quên các test case phổ biến
 - Tiết kiệm thời gian
- Con người cần bổ sung:
 - Test hiếm gặp
 - Test biên hệ thống
- **Kiểm thử tốt = nghĩ ra tình huống mà code có thể sai**

Kết luận:

Không nên chỉ dùng test case của AI, mà phải **tự suy nghĩ thêm** để đảm bảo chương trình an toàn.

BÀI TẬP 9 :

[BÀI TẬP] PHÂN TÍCH LOGIC ĐOẠN MÃ

1. Đoạn code ban đầu (chạy được nhưng sai kết quả)

```
#include <stdio.h>
```

```
int main() {  
    int n = 5;  
    int giai_thua = 0;  
  
    for (int i = 1; i <= n; i++) {  
        giai_thua = giai_thua * i;  
    }  
  
    printf("Giai thua cua %d la %d\n", n, giai_thua);  
    return 0;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

2. Hiện tượng khi chạy chương trình

- Chương trình **không báo lỗi**
- Kết quả in ra:

Giai thua của 5 là 0

✗ Sai, vì $5! = 120$

3. Prompt dùng để hỏi AI

Prompt

Code C sau chạy không báo lỗi nhưng cho kết quả sai.

Hãy:

1. Giải thích chi tiết lỗi logic trong chương trình
2. Chỉ ra dòng gây lỗi
3. Sửa lại code để cho kết quả đúng

4. Phân tích lỗi logic (giải thích như AI)

Lỗi logic chính

- Biến `giai_thua` được **khởi tạo sai giá trị ban đầu**

```
int giai_thua = 0;
```

- Trong phép tính giai thừa:
 - Công thức: $n! = 1 \times 2 \times 3 \times \dots \times n$
 - Giá trị ban đầu phải là **1**, không phải 0
- Khi nhân với 0:

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

- $0 \times i = 0$ với mọi i
- Vì vậy kết quả luôn bằng 0

Dòng gây lỗi logic

```
int giai_thua = 0; // sai
```

5. Code sau khi sửa (đúng logic)

```
#include <stdio.h>

int main() {
    int n = 5;
    int giai_thua = 1; // khởi tạo đúng

    for (int i = 1; i <= n; i++) {
        giai_thua = giai_thua * i;
    }

    printf("Giai thua cua %d la %d\n", n, giai_thua);
    return 0;
}
```

6. Giải thích lỗi bằng lời của mình

Chương trình không bị lỗi cú pháp nên vẫn chạy bình thường, nhưng sai ở **cách suy nghĩ logic**.

Giai thừa là phép **nhân liên tiếp**, nên biến dùng để nhân phải bắt đầu từ 1. Khi khởi tạo bằng 0, mọi phép nhân phía sau đều cho kết quả 0, dẫn đến chương trình in ra kết quả sai.

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

7. Bài học rút ra

- Code **chạy được** ≠ **code đúng**
- Lỗi logic thường đến từ:
 - Khởi tạo biến sai
 - Hiểu sai công thức toán
- Khi debug:
 - So sánh kết quả với giá trị mong đợi
 - Kiểm tra giá trị ban đầu của biến
 - Xem lại công thức

BÀI TẬP 10 :

[BÀI TẬP] KẾT HỢP: VIẾT – SỬA LỖI – REFACTOR – TEST

1. Mục tiêu

- Rèn kỹ năng làm việc **có hệ thống** với AI
- Không chỉ xin code, mà:
 - Viết
 - Kiểm thử
 - Cải tiến
 - Giải thích lý do cải tiến

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

2. Đề bài ban đầu

```
// Yêu cầu: Viết hàm tính tổng các số Fibonacci từ 1 đến n
int fibonacciSum(int n) {
    // Code còn thiếu
}

int main() {
    int n = 5;
    printf("Tổng Fibonacci den %d la %d\n", n, fibonacciSum(n));
    return 0;
}
```

GIAI ĐOẠN 1 – VIẾT CODE

Prompt 1 – Viết code còn thiếu

Prompt 1

Viết hàm C fibonacciSum(int n) để tính tổng các số Fibonacci từ 1 đến n.
Giả sử $n \geq 0$.

Kết quả AI (phiên bản ban đầu – CHƯA tối ưu)

```
int fibonacciSum(int n) {
    if (n <= 0) return 0;

    int a = 0, b = 1;
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        int c = a + b;
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
        a = b;  
        b = c;  
        sum += b;  
    }  
  
    return sum;  
}
```

Nhận xét nhanh

- Code chạy được
- Logic **chưa rõ ràng**
- Dễ gây nhầm lẫn Fibonacci tính theo **vị trí hay giá trị**

GIAI ĐOẠN 2 – THÊM TEST CASE

Prompt 2 – Yêu cầu thêm test case

Prompt 2

Hãy viết thêm test case để kiểm thử hàm fibonacciSum với các giá trị:

n = 0, n = 1, n = 10.

In kết quả bằng printf.

Code test (AI + chỉnh cho rõ)

```
#include <stdio.h>
```

```
int fibonacciSum(int n) {  
    if (n <= 0) return 0;
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

```
int a = 0, b = 1;
int sum = 0;

for (int i = 1; i <= n; i++) {
    int c = a + b;
    a = b;
    b = c;
    sum += b;
}

return sum;
}

int main() {
    printf("n = 0, sum = %d\n", fibonacciSum(0));
    printf("n = 1, sum = %d\n", fibonacciSum(1));
    printf("n = 10, sum = %d\n", fibonacciSum(10));
    return 0;
}
```

Vấn đề phát hiện qua test

- Không rõ “từ 1 đến n” là:
 - Tổng **n** số Fibonacci đầu tiên
 - Hay tổng các số Fibonacci $\leq n$

Cần làm rõ yêu cầu → refactor.

GIAI ĐOẠN 3 – REFACTOR & TỐI ƯU

Prompt 3 – Refactor code

Prompt 3

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

Refactor lại hàm `fibonacciSum` để:

- Rõ ràng về ý nghĩa
- Tối ưu logic
- Dễ đọc cho sinh viên năm nhất
- Tổng các số Fibonacci $\leq n$

Code sau khi refactor (TỐT HƠN)

```
int fibonacciSum(int n) {  
    if (n < 1) return 0;  
  
    int a = 0;  
    int b = 1;  
    int sum = 0;  
  
    while (b <= n) {  
        sum += b;  
        int next = a + b;  
        a = b;  
        b = next;  
    }  
  
    return sum;  
}
```

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

GIẢI ĐOẠN 4 – GIẢI THÍCH VÌ SAO CODE REFACTOR TỐT HƠN

Prompt 4 – Yêu cầu giải thích

Prompt 4

Hãy giải thích bằng lời đơn giản tại sao phiên bản code refactor tốt hơn phiên bản ban đầu.

Giải thích (bằng lời – để nộp bài)

- Phiên bản refactor:
 - Làm rõ **Fibonacci** $\leq n$, không mơ hồ
 - Không dùng biến đếm vòng lặp không cần thiết
 - Dùng `while` → đúng bản chất bài toán
- Code:
 - Dễ đọc
 - Dễ test
 - Dễ sửa nếu đổi yêu cầu

Tối ưu không chỉ là nhanh hơn, mà là rõ ràng hơn

5. So sánh trước & sau refactor

Tiêu chí	Code ban đầu	Code refactor
Rõ yêu cầu	Không	Có
Dễ đọc	Trung bình	Cao
Logic	Dễ nhầm	Rõ ràng
Khả năng mở rộng	Thấp	Tốt
Phù hợp học tập	Vừa	Rất tốt

Họ và tên : Lý Gia Huy

BTVN : session 6

Lớp : CNTT 5

6. Tổng kết toàn bộ quá trình

- Prompt 1: Sinh code nhanh
- Prompt 2: Phát hiện vấn đề qua test
- Prompt 3: Refactor để chuẩn hóa
- Prompt 4: Hiểu sâu, không học vẹt

Kết luận:

Làm việc tốt với AI = biết **chia nhỏ yêu cầu**, không hỏi một lần rồi thôi.

7. Đánh giá (để ghi vào bài nộp)

- Hoàn thành đầy đủ các bước
- Có test case
- Có refactor
- Có giải thích bằng lời
- Thể hiện tư duy lập trình & kiểm thử