Unit – 4
Software Coding and Testing

Thanki Kunal
Lecturer, Government Polytechnic-Porbandar

4.1 Coding and Code Review

Coding

- The input to the coding phase is the design document.
- During coding phase:
  - modules identified in the design document are coded according to the module specifications.
  - Unit test modules
- At the end of the design phase we have:
  - module structure (e.g. structure chart) of the system
  - module specifications:
    - data structures and algorithms for each module.
- **Objective of coding phase:**
  - transform design into code
  - unit test the code.

Coding Standards

- Good software development organizations require their programmers to adhere to some standard style of coding: called ***coding standards***.
- Many software development organizations: formulate their own coding standards that suits them most
- **Advantages** of using Coding Standards:
  - it gives a uniform appearance to the codes written by different engineers,
  - it enhances code understanding,
  - encourages good programming practices.

Coding Standards and Guidelines

- A coding standard sets out standard ways of doing several things:
  - Header information,
  - the way variables are named,
  - code is properly indented
  - maximum number of source lines allowed per function, etc.
  - Use of proper comment

Representative Coding Standards

- Contents of headers for different modules:
  - The headers of different modules should be standard for an organization.
  - The exact format for header information is usually specified.

- Header data:
  - Name of the module,
  - date on which the module was created,
  - author's name,
  - modification history,
  - synopsis of the module,
  - different functions supported, along with their input/output parameters,
  - global variables accessed/modified by the module.

Representative Coding Standards

- Rules for limiting the use of globals:
  - what types of data can be declared global and what can not.
- Naming conventions for
  - global variables,
  - local variables, and
  - constant identifiers.
- Error return conventions and exception handling mechanisms.
  - the way error and exception conditions are handled should be standard within an organization.
  - For example, when different functions encounter error conditions
  - should either return a 0 or 1 consistently.

Representative Coding Guidelines

- Do not use too clever and difficult to understand coding style:
  - Code should be easy to understand.
- Avoid obscure side effects:
  - one that is not obvious from a casual examination of the code.
  - makes later maintenance difficult.
  - The side effects of a function call include:
    - modification of parameters passed by reference,
    - modification of global variables,
    - I/O operations.
  - For example,
    - if a global variable is changed obscurely in a called module, it becomes difficult for anybody trying to understand the code.

Representative Coding Guidelines

- Do not use an identifier (variable name) for multiple purposes:
  - Programmers often use the same identifier for multiple purposes.
  - For example, some programmers use a temporary loop variable also for storing the final result.

```
for(i=1;i<100;i++)
        {…..}
      i=2*p*q;
return(i);
```

  - There are several things wrong with this approach, hence should be avoided.
  - Each variable should be given a name indicating its purpose:
    - This is not possible if an identifier is used for multiple purposes.

Representative Coding Guidelines

- Code should be well-documented.
  - on the average there must be at least one comment line for every three source lines.
  - The length of any function should not exceed 10 source lines.
- Avoid Lengthy functions:
  - usually very difficult to understand
  - probably do too many different things.
- Do not use goto statements.
  - Use of go to statements:
    - make a program unstructured
    - make it very difficult to understand.

Code Review

- After a module has been coded,
  - ***code inspection*** and ***code walk through*** are carried out
  - ensures that coding standards are followed
  - helps detect as many errors as possible before testing.

- Detect as many errors as possible during inspection and walkthrough:
  - detected errors require less effort for correction
  - much higher effort needed if errors were to be detected during integration or system testing.

Code Review

- Reviewer checks for :
  - Potential flaws of code
  - Consistency with overall program design
  - Quality of comment
  - General rules of coding standard

- Two types of code review
  - Code walkthrough
  - Code inspection

Code Walk Through

- An **informal code analysis** technique.
  - author presents the code to **peers for feedback**.
  - It is more **focused on learning**, discussion, and identifying potential improvements **rather than detecting defects.**
- **Key Features**
  - **Less Formal**: No strict roles or predefined checklists.
  - **Interactive:** The author explains the code, and reviewers provide feedback.
  - **Focus on Learning**: Helps team members understand the codebase and discuss potential improvements.
  - **Minimal Preparation**: No extensive analysis before the meeting is required.
- The main objectives of the code walk through is to discover the **algorithmic and logical errors in the code.**

Code Walk Through

Process of Code Walkthrough

- **Preparation (Optional)** – The author may provide background on the code.
- **Presentation** – The author walks the team through the code.
- **Discussion & Feedback** – Peers provide feedback on logic, efficiency, and clarity.
- **Revision** – The author incorporates the suggested changes.

Code Walk Through

- □ **Advantages of Code Walkthrough**

✓Encourages collaboration and knowledge sharing.

✓Helps new developers understand the project.

✓Provides quick feedback with minimal overhead.

- □ **Disadvantages of Code Walkthrough**

✕ May miss defects due to its informal nature.

✕ Lacks structured defect tracking and metrics.

Code Inspection

- In contrast to code walk through,
  - Code Inspection is a **formal, rigorous review process** where a team thoroughly examines the code against predefined standards
  - Code inspection aims mainly at **discovery of commonly made errors.**
- During code inspection:
  - the code is examined for the **presence of certain kinds of errors,**
  - in contrast to the hand simulation of code execution done in code walk through. **Standards are checked**

Code Inspection

□ Common specific Errors:

- Use of uninitialized variables.
- Non terminating loops.
- Array indices out of bounds.
- Incompatible assignments.
- Improper storage allocation and deallocation.
- Actual and formal parameter mismatch in procedure calls.
- Jumps into loops.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values, etc.

Code Inspection

- For instance, consider:
  - classical error of writing a **procedure that modifies a formal parameter**
  - while the **calling routine** calls the procedure **with a constant actual parameter.**
- It is more likely that such an error will be discovered:
  - by looking for this kind of mistakes in the code,
  - rather than by simply hand simulating execution of the procedure.

Code Inspection

- **Key Features**
  - **Highly Structured**: Follows a predefined process with specific roles.
  - **Formal Process**: Uses checklists, predefined review criteria, and detailed documentation.
  - **Focus on Defects**: The primary goal is to find issues, not to discuss alternative solutions.
  - **Requires Preparation**: Reviewers must analyze the code before the inspection meeting.
  - **Metrics-Based**: Collects data on defects and the review process for future improvements.

Code Inspection

- **Roles in Code Inspection**
  - **Moderator (Facilitator)** – Organizes the review process and ensures adherence to guidelines.
  - **Author (Developer)** – Writes the code and provides necessary documentation.
  - **Reviewer (Inspector)** – Checks for defects based on coding standards and checklists.
  - **Recorder (Scribe)** – Documents the issues found during the inspection.
  - **Manager (Optional)** – Oversees the review process and ensures compliance.

Code Inspection

- **Process of Code Inspection**
  - **Planning** – Moderator schedules the review and selects the team.
  - **Preparation** – Reviewers analyze the code and use checklists to identify defects.
  - **Inspection Meeting** – The team discusses identified issues and logs defects.
  - **Rework** – The author corrects the identified defects.
  - **Follow-up** – The moderator verifies whether the fixes are implemented correctly.

Code Inspection

- **Advantages of Code Inspection**

  ✓ Detects a high number of defects early.
  ✓ Improves software quality and maintainability.
  ✓ Enhances compliance with coding standards.
  ✓ Facilitates knowledge sharing among team members.

- **Disadvantages of Code Inspection**

  ✕ Time-consuming due to its structured nature.
  ✕ Requires significant preparation and resources.

Examples:

🔷 1️⃣ **Code Walkthrough Example**

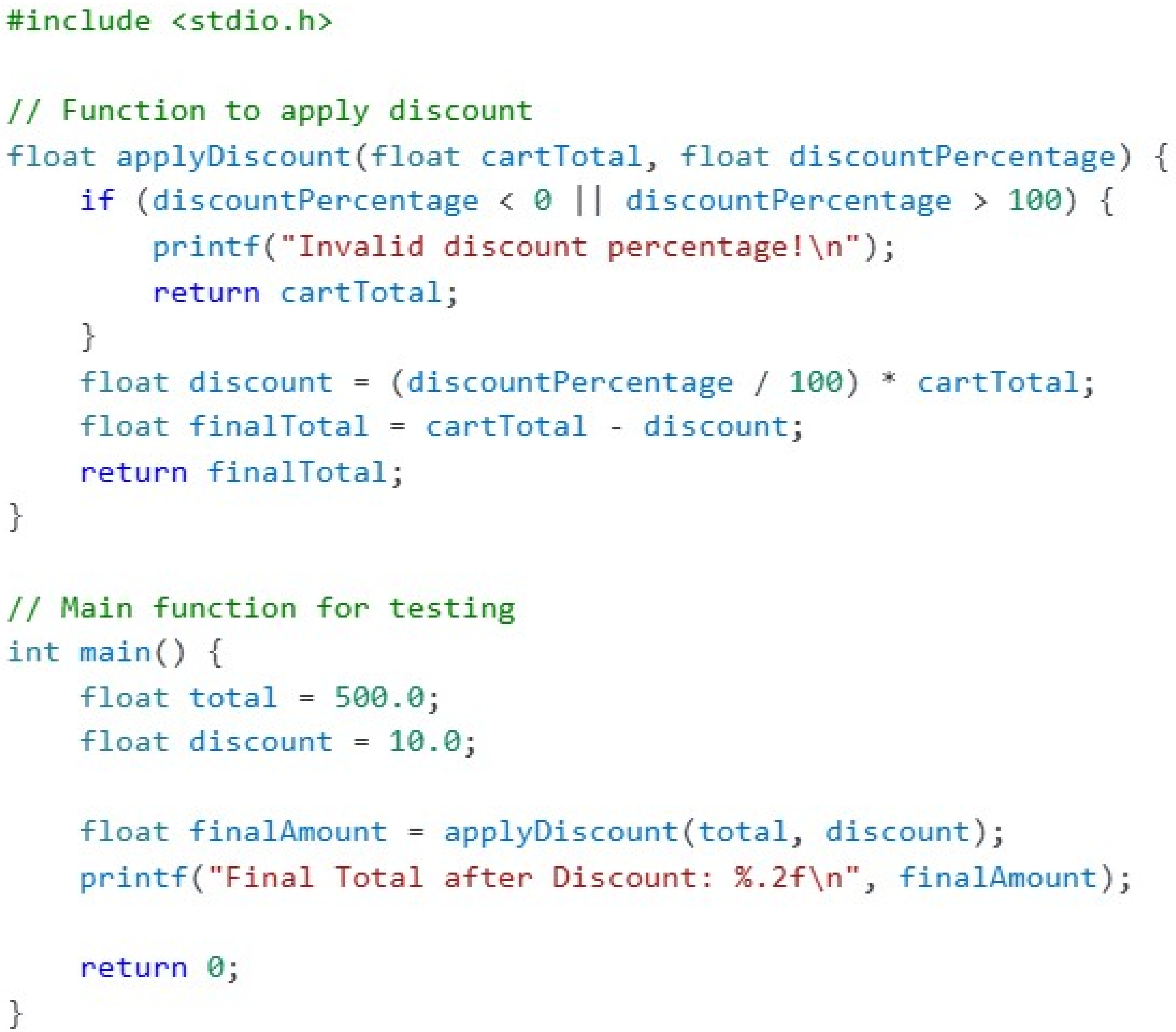
📌 **What is a Code Walkthrough?**

A **code walkthrough** is an informal review process where the developer explains their code to peers or a reviewer to get feedback. The goal is to improve logic, readability, and maintainability.

👨‍💻 **Scenario:**

A developer has implemented a **cart discount feature** in a C-based code. They walk through the code with a senior developer to ensure correctness.

✅ **Code to Walk Through (Coupon Discount in Checkout)**



```c
#include <stdio.h>

// Function to apply discount
float applyDiscount(float cartTotal, float discountPercentage) {
    if (discountPercentage < 0 || discountPercentage > 100) {
        printf("Invalid discount percentage!\n");
        return cartTotal;
    }
    float discount = (discountPercentage / 100) * cartTotal;
    float finalTotal = cartTotal - discount;
    return finalTotal;
}

// Main function for testing
int main() {
    float total = 500.0;
    float discount = 10.0;

    float finalAmount = applyDiscount(total, discount);
    printf("Final Total after Discount: %.2f\n", finalAmount);

    return 0;
}
```

🔍 **Walkthrough Discussion & Feedback:**

✅ **Correctly calculates discount** ✅

❌ **No validation for negative cart total** – Needs a check.

❌ **No unit tests provided** – Should be tested with various values.

✅ **Good function separation** – applyDiscount() is independent of main().

🔷 **Suggested Improvement:**

- **Add validation** to check if cartTotal is negative.
- **Write unit tests** to check cases like 0% and 100% discounts.

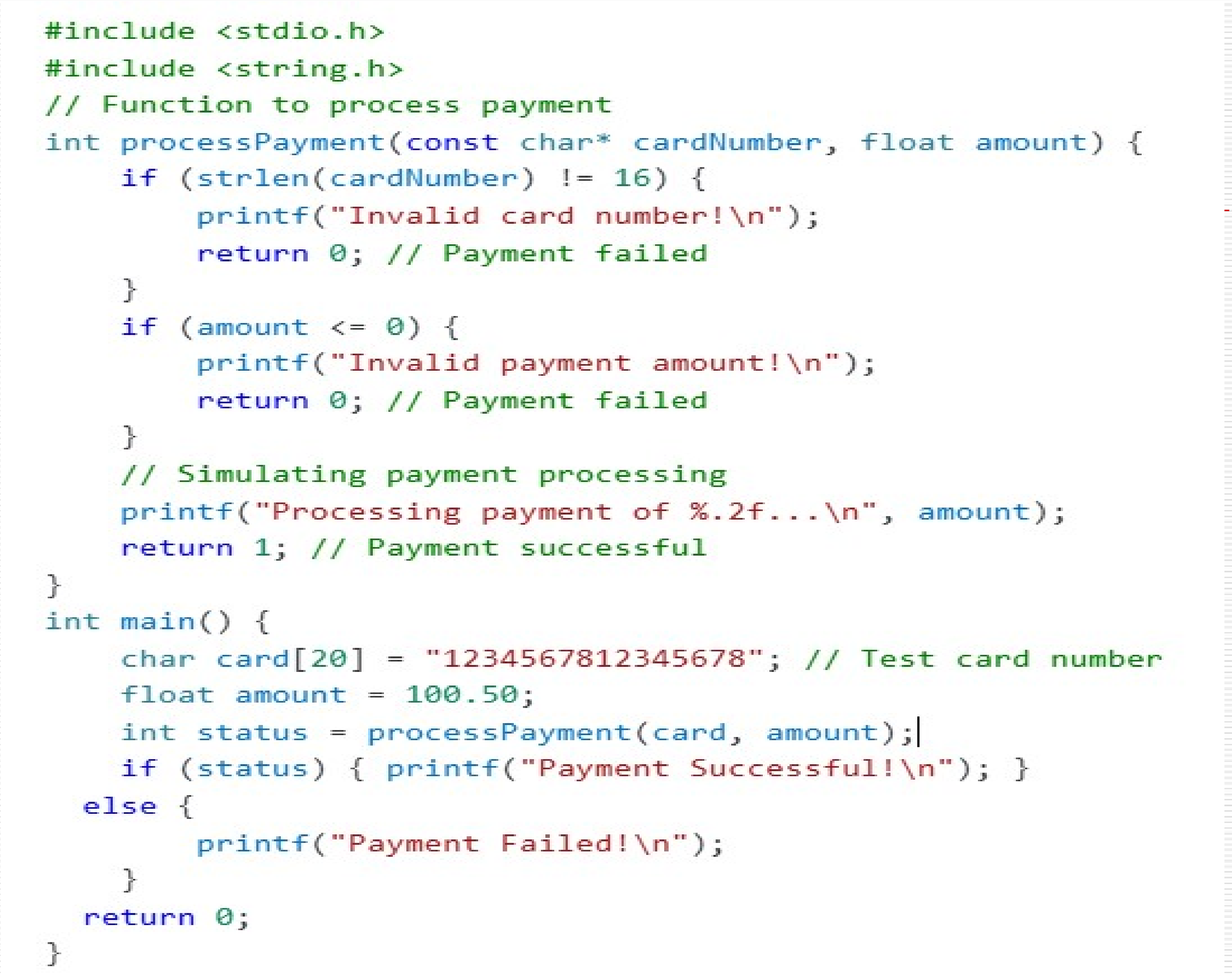
🔷 2️⃣ Code Inspection Example

📌 What is Code Inspection?

A **code inspection** is a **formal** review where experienced developers inspect the code against **coding standards, security, and performance issues**. The process is usually **documented**.

🧑‍💻 Scenario:

A security expert and senior developer inspect the **payment processing code** for vulnerabilities before deployment.



```c
#include <stdio.h>
#include <string.h>
// Function to process payment
int processPayment(const char* cardNumber, float amount) {
    if (strlen(cardNumber) != 16) {
        printf("Invalid card number!\n");
        return 0; // Payment failed
    }
    if (amount <= 0) {
        printf("Invalid payment amount!\n");
        return 0; // Payment failed
    }
    // Simulating payment processing
    printf("Processing payment of %.2f...\n", amount);
    return 1; // Payment successful
}
int main() {
    char card[20] = "1234567812345678"; // Test card number
    float amount = 100.50;
    int status = processPayment(card, amount);
    if (status) { printf("Payment Successful!\n"); }
  else {
        printf("Payment Failed!\n");
    }
  return 0;
}
```

🔍 Code Inspection Findings & Issues:

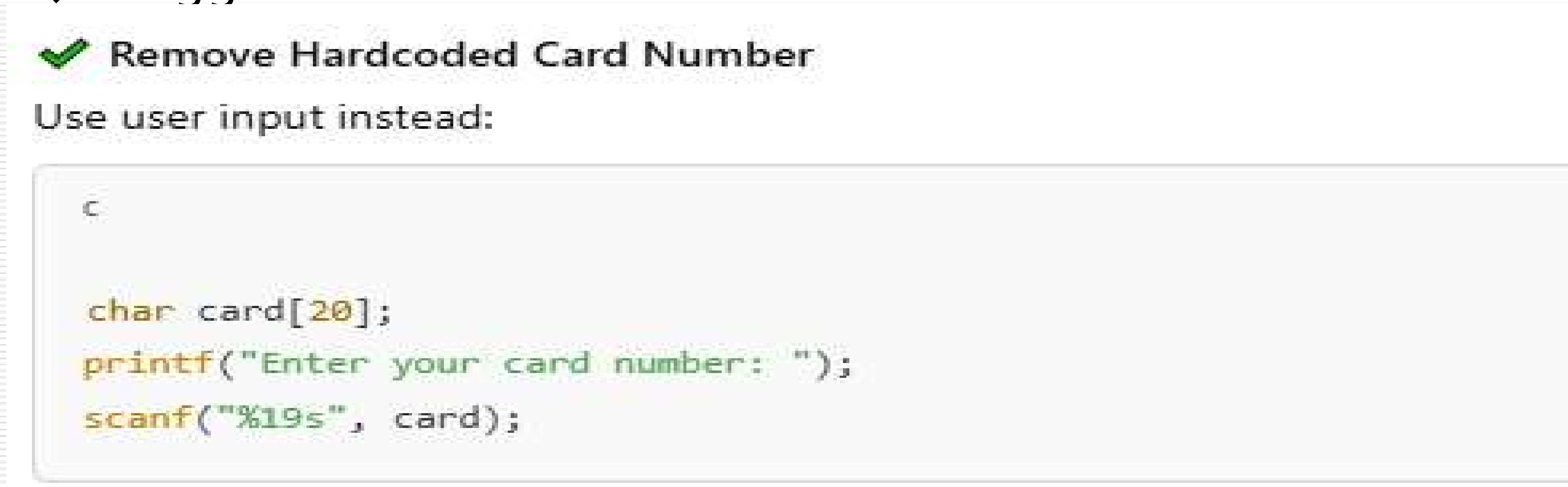
🚨 Security Issues:

❌ **No encryption for card details** – Sensitive data should never be stored or processed in plain text.
❌ **Hardcoded card number for testing** – Risky practice, should be dynamically inputted.
❌ **No logging of failed payments** – Debugging will be difficult.

🚀 Performance & Best Practices:

✅ **Validates card length correctly** ✅
❌ **No input sanitization** – Should check for non-numeric characters.
❌ **Lack of modular design** – processPayment() should be part of a separate payment module.

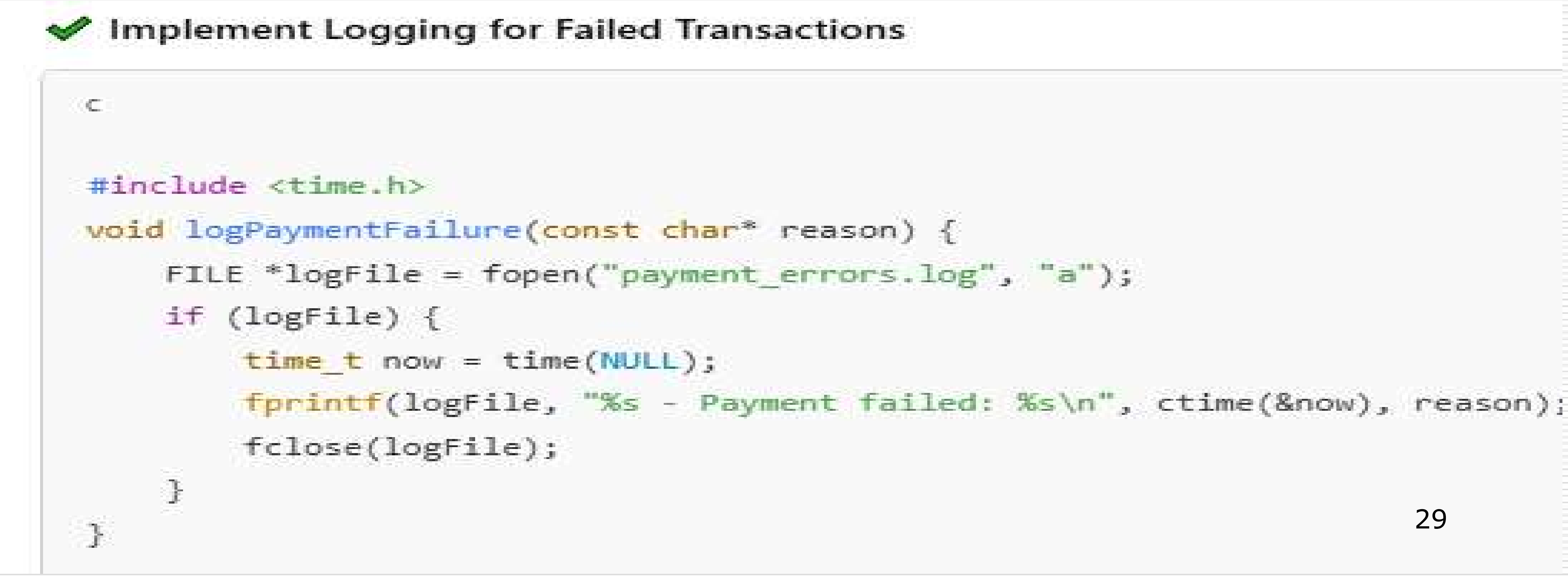
◆ Suggested Fixes:



✓ Remove Hardcoded Card Number

Use user input instead:

```c
char card[20];
printf("Enter your card number: ");
scanf("%19s", card);
```



✓ Implement Logging for Failed Transactions

```c
#include <time.h>
void logPaymentFailure(const char* reason) {
    FILE *logFile = fopen("payment_errors.log", "a");
    if (logFile) {
        time_t now = time(NULL);
        fprintf(logFile, "%s - Payment failed: %s\n", ctime(&now), reason);
        fclose(logFile);
    }
}
```

📌 Key Differences Between Walkthrough & Inspection

Feature	Code Walkthrough	Code Inspection
Type	Informal	Formal
Goal	Improve code logic, readability	Find defects, security issues
Participants	Developer + Peers	Experts (Senior Devs, QA, Security)
Output	Suggestions for improvement	Documented report with fixes