Tower of Hanoi Assignment

Class : Advanced C Programming

Submission Date : 2024/04/05

Student ID : 2022742021

Affiliation : Kwangwoon University

Full name : Juho Kim

Major : Software Engineering

Abstract

This report provides a comprehensive walkthrough of the development process for the Tower of Hanoi implemented in C++. Initially, the fulfillment of each requirement is discussed, accompanied by descriptive code snippets. Subsequently, the report delves into the implementation of additional features. Furthermore, the correctness of the code is demonstrated through an explanation of the underlying mechanisms supplemented by flowcharts and the presentation of error handling processes via code snippets. Lastly, the report concludes by addressing the challenges encountered during development and outlining the limitations of the current code, which presents opportunities for future improvement.

The game was developed applying the knowledge of vectors, error handling methods, Object Oriented Programming (OOP) and other concepts from advanced c programming class. For more information, please visit my GitHub repository.

For the sake of simplicity, some functions discussed in this report may be presented without parameters.

Requirements Fulfillment

Following are the assignment requirements from the slides along with its corresponding code implementation. Error handling processes are not discussed in detail as it will be talked about in the correctness of the code.

 Requirement 1: Only one disk can be moved at a time (Used push_back, pop_back)

```
void move_disk(vector<vector<int>>& v, int from, int to) {
    v[to - 1].push_back(v[from - 1].back());
    v[from - 1].pop_back();
}
```

Instead of pass by value, rods with disks are passed by reference in **move_disk()** for efficient processing of large numbers of rods and disks.

Before implementing pass by reference, the performance of pass by value and pass by reference were compared to select the most efficient method for our assignment.

	Pass by Value	Pass by Reference
Time Complexity	O(n)	O(1)
Memory Usage	Additional memory for copy	No additional memory required
Function Call	Copying data	Direct access to data
Impact on Stack	Increased stack usage	No impact on stack
Suitable for	Small data or immutable	Large data or mutable
	objects	objects

As the number of rods and disks are predefined by the user, the user may choose to play the game with large numbers of rods and disks (for example, 1000 rods and 1000disks!) and if the entire rods are copied every time a user enters a move, we're not only wasting a lot of memory but most importantly increasing the time complexity by O(n). This may cause delays when user decides to play the game with a lot of rods and disks.

To avoid such hazards, rods are passed by reference for directly accessing the data every time user makes a move.

 Requirement 2 : Each move consists of taking the upper disk from one of the stack and placing it on top of another stack or on an empty rod. No larger disk may be placed on top of a smaller disk.

The following logics were implemented in **IsMoveAllowed()** to meet the requirements:

1. Checking the highest disk on a rod

```
int fromPeakElement = get_peak_disk(v[from - 1]);
int toPeakElement = get_peak_disk(v[to - 1]);
```

The highest disk placed on the rod moving from and to are checked for comparing the size of the disks before moving the disk.

2. Preventing user from placing disk on the same rod

```
if (from == to)
    error("( Attempted to place disk on the same rod )");
```

When the indices moving from is equal to that of moving to, runtime error is thrown with the error message displayed. The error is handled by the catch statement inside **IsMoveAllowed()** to redirect user for reentering their move.

3. Preventing user from moving a nonexistent disk

```
else if (fromPeakElement == 0)
    error("( Cannot move from rod with no disks )");
```

If the highest disk on the rod moving from is zero, there is no disks on the rod. Hence if such scenario occurs, we'll throw a run time error with an error message. The error is handled by the catch statement inside **IsMoveAllowed()** to redirect user for reentering their move.

4. Ensuring that a larger disk is not placed on top of smaller disk

```
else if (toPeakElement != 0 && toPeakElement < fromPeakElement)
    error("( Destination rod has a smaller disk than the one being moved )");
```

The highest disks on the rod moving to is checked if it's not zero because we do not want to trigger the error when trying to move a disk to a rod with no disks.

Finally, the size of topmost disk on rod moving to is compared with that of rod moving from to display error message when user attempts to place larger disk on top of a smaller disk.

Requirement 3 : All disks are stacked on the first rod in a decreasing order of sizes

The following code were implemented in **initialize_disks()** :

1.Initializing towers with r rods

```
vector<vector<int>> towers(r);
```

A 2D vector, towers created with its rows initialized as the number of rods.

2.Creating initial tower with disks stacked in descending order

```
vector<int> initial_tower(d);
for (int i = 0; i < d; i++)
{
    initial_tower[i] = d - i;
}

towers[0] = initial_tower;
```

initial_tower (1D vector) is initialized with disks stacked in descending order of sizes using for loop and transferred to the first rod on **towers** vector. This ensures that all disks are stacked on the first rod in decreasing order of sizes.

 Requirement 4 : Indices of rods (starts from 1, not 0)

The following has been implemented in **print_towers()** :

```
cout << "[" << r + 1 << "] ";
```

By adding 1 to indices of rods, the indices of the rods are displayed to start from 1, not 0.

 Requirement 5 : Number of moves (starts from 1 again, not 0)

Initial number of moves is set to zero in **play_game()** because

```
int num_moves = 0;
```

The number of moves will be updated everytime before showing a prompt for entering user movement in **ask_move().**

```
pair<int, int> ask_move(int rods, int& num_moves){
    num_moves += 1;
    display_prompt(num_moves, rods);
    return get_move(rods, num_moves);
}
```

 Requirement 5 : The allowed indices of the rod from which a disk will be moved

In **display_prompt(),** all allowed indice are added to **indices** using for loop.

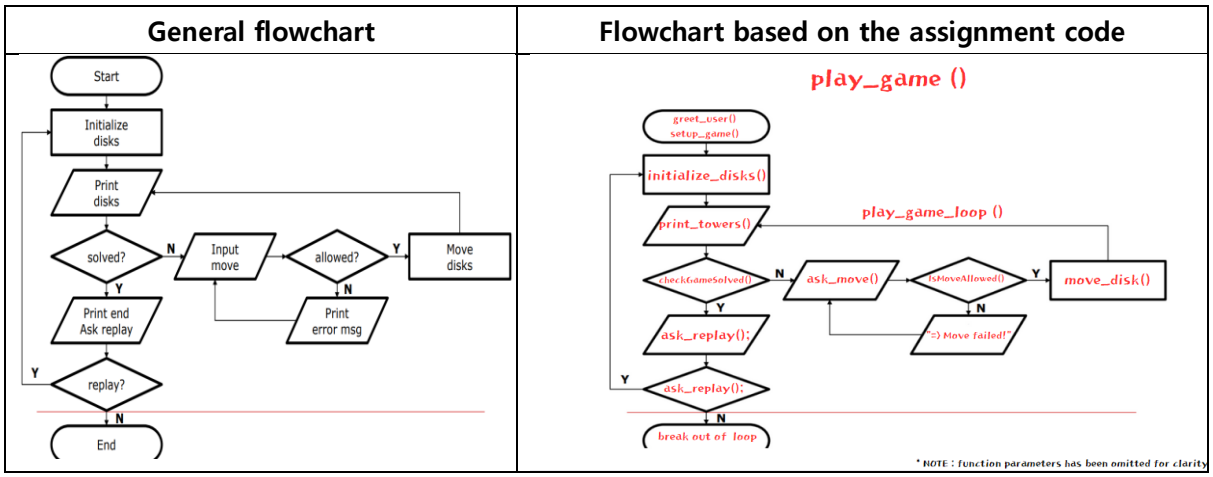
```
string indices = "[1";
for (int i = 1; i < rods; i++) {
    indices += "|" + to_string(i + 1);
}
indices += "]";
```

The **indices** obtained are displayed on the prompt

```
cout << "[" << num_moves  << "] From which tower will you move a disk to which tower? (from="
    << indices << ", to=" << indices << "): ";
```

Flow chart of play_game()

The general flow chart of hanoi tower has been specified based on the assignment code as follows :



The game begins with **greet_user()** displaying welcome message to the user and **setup_game()** asking input for number of rods and disks. Once the number of rods and disks is set by the user via **initialize_disks()**, its corresponding initial rods and disks are displayed by **print_towers()**.

Then **ask_move()** prompts the user to enter the rod they're moving the disk from and to. When movement is allowed, **IsMoveAllowed()** returns true enabling the user's movement to be entered as a parameter of **move_disk()** for transferring the disk to a specific rod; otherwise, an error message is displayed and false is returned.

Once the disks have been moved, **checkGameSolved()** checks if game has been solved. If user hasn't completed the game or entered an invalid move, the user enters a loop to repeat the game solving process starting from **ask_move()** until game is solved.

Once all the disks has been transferred successfully to another rod, the game ends showing number of moves by user and **ask_replay()** asking for replay. If user decides to replay by entering "y" **play_game()** is called again to repeat all the steps mentioned above. Otherwise, when "n" is entered, the user breaks out of the loop causing the game to end.

Additional Features

1. Displaying welcome message

An ASCII art of the phrase "Welcome to tower of Hanoi" was created using online Text to ASCII art generator. Then under **greet_user()**, the ascii art was saved as string into the variable asciiArt and displayed.

2. Getting the number of rods and disks from user

Under **play_game()**, two variables **rods** and **disks** were created for storing number of rods and disks.

```
void play_game() {
    int rods, disks;
```

The variables has been passed by reference to update its values within **setup_game()**

```
setup_game(rods, disks);
```

Upon getting the user input, **rods** and **disks** are updated

```
void setup_game(int& rods, int& disks) {
    rods = get_rods();
    disks = get_disks();
```

Code Correctness Demonstration

Error handling of the code was done as follows :

 Error handling 1: Receiving valid number of rods and disks

The code for **get_rods()** and **get_disks()** has the same code implementation. The explanation for this example is based on **get_rods()**, but the mechanism is the same for **get_disk()** as well.

1.Preventing non integer input

```
int rods;
try {
    cout << "Total number of rods: ";
    cin >> rods;

    if (!cin)
        error("( Please enter an integer :D )");
```

If user tries to enter non integer input, !cin is true throwing a runtime error.

```
catch (runtime_error& e) {
    cout << "INVALID INPUT "<< e.what() << endl << endl;
    clearInputBuffer();// Clear the input buffer
    return get_rods(); // Retry input
}
```

The error is caught within **get_rods()**, message is displayed, input buffer is cleared and **get_rods()** is called again to ask user for input again.

2. Limiting user from entering below minimum

```
else if (rods < 3)
    throw out_of_range("( Number of rods must be 3 or more )");
```

The game is set to prevent user from entering rods of less than 3 because if user enters 2 rods and 2 disks, the game is essentially unsolvable. Upon entering number of rods less than 3, out of range error is thrown.

```
else if (disks <= 0)
    throw out_of_range("Please enter a positive integer");
```

When user attempts to enter number of disks less than or equal to zero, out of range error is thrown.

```
catch (out_of_range& e) {
    cout << "OUT_OF_RANGE : " << e.what() << endl << endl;
    clearInputBuffer();
    return get_rods();
}
```

The error is caught within **get_rods()**, out of range error message is displayed, input buffer is cleared and **get_rods()** is called to ask user for input again.

3. Prevent number of rods being less than that of disks

```
try {
    // Check if number of rods is less than the number of disks
    if (rods < disks) {
        error("( Number of rods must be greater than or equal to the number of disks!! )");
```

Under **setup_game(),** if number of rods is less than that of disks, a runtime error is thrown.

```
catch (runtime_error& e) {
    cout << "INVALID INPUT " << e.what() << endl << endl;
    setup_game(rods, disks);
}
```

The error is handled within the **setup_game()** to recall **setup_game()** for asking user for number of rods and disks once again.

Error handling 2: Receiving valid moves from user

The code for **get_moveTo()** and **get_moveFrom()** has the same code implementation. In this example, explanation is based on **get_moveTo()**, but the mechanism is the same for **get_moveFrom()** as well.

1.Preventing non integer input

```
if (!cin)
    error("( Please enter an integer :D )");
```

If user tries to enter non integer input, !cin is true throwing a runtime error outside the **get_moveTo()**.

```
catch (runtime_error& e) {
    cout << "=> Move failed! " << e.what() << endl;
    clearInputBuffer();
    return ask_move(rods, num_moves);
}
```

The error is caught within **get_move()**, error message is displayed, input buffer is cleared and **ask_move()** is called again to ask user for input once again.

2. Limiting user from entering below minimum

```
else if (to <= 0)
    throw out_of_range("( Please enter move within the range )");
```

The game is set to prevent user from entering move less than equal to zero as such indices is out of range. Hence any move less than or equal to zero will throw an out of range error.

```
catch (out_of_range& e) {
    cout << "=> Move failed! " << e.what() << endl;
    clearInputBuffer();
    return ask_move(rods, num_moves);
}
```

The out_of_range error from **get_moveTo()** is caught at **get_move(),** error message is displayed, input buffer is cleared, and **ask_move()** is recalled to ask user for move once again.

3. Checking if move is allowed

```
if (from <= 0 || from > rods || to <= 0 || to > rods)
    throw out_of_range("( Rod is out of range )");
```

At **IsMoveAllowed()**, users move to and from are checked and if it's not within available indices, an out of range error is thrown.

```
catch (out_of_range& e) { // Out of range
    cout << "=> Move failed! " << e.what() << endl;
    return false;
```

Error is caught at **IsMoveAllowed(),** error message is displayed, and false is returned.

```
bool moveable = IsMoveAllowed(to, from, rods, towers);
```

This sets the moveable variable to false and prevents the disk from being moved

Conclusion

□ Key challenges tackled

Some of the hardest parts were:

1. Developing code logic following the flowchart

Initially the code was written sequentially without any specific flowchart. To repeat certain parts of the game, all steps were broken down into small subsets and certain functions were included in the **play_game_loop()** to keep up with the flowchart.

2. Handling errors for each input without disrupting the flowchart

The first attempt was handle all the errors in the **play_game_loop().** However, such a naïve attempt deteriorated the readability, semantics and structure of the code with overcomplicated logics. To

overcome such limitations, all the error handling were added at each subprocesses, or functions.

3. Keeping track of moves

Based on the lecture, I tried entering the number of moves via passing by value into **ask_move()** and returning the updated value to the variable, **move**. But if I were to do so, I had to take out **display_prompt()** and create another variable for receiving the current number of moves. This caused a lot of code smell within the **play_loop().**

Hence **ask_move()** was modified to pass **num_moves** by reference and keep updating number of moves while simultaneously returning the movement entered by the user.

4. Stopping myself from over engineering

As the code grew larger, I found myself constantly over engineering the process for achieving the simplest task. For instance, when attempting to create a loop for maintaining gameplay, all the subprocesses or functions were written inside a main function with nested while loops for **play_game()** and **play_game_loop().** This not only made it hard to read, but also time consuming for debugging.

The problem was tackled using the concept of extraction, inversion and abstraction learned from CodeAesthetic. Using the approaches learned, the code was simplified with any redundant or unnecessary code removed and nested loops removed.

5. Coding with good semantics and code structure

Finally, the code itself had all the necessary logic for successfully running the Tower of Hanoi. However, it was very hard even for myself to look at the code and understand how it worked. I added a lot of comments to each lines of code but this further deteriorated the readability of my code.

So I had to watch a controversial video "Don't Write Comments" by CodeAesthetics for improving the semantics and code structure. According to the video, the best coding style is to make the code self-explanatory so that it doesn't need a comment in the first place.

Applying what I've learned, steps were further broken down with subfunctions. This simplified the code so that readers could understand without the having to see all the details. Furthermore, Additional functions were created to aggregate any subfunctions.

Finally with the aggregated functions, the main function was written with much simpler code with better structure. Now the reader doesn't have to read all the details and figure out how the code works! The code itself is a comment! It's self-explanatory!

Limitations

1. No user friendly text visuals of rods and disks
2. Recursively calling function when asking user for input again

Future improvements

1. Adding text visuals of rod and disks
2. Using alternative methods to recursively call a function

References

- How to simplify your code
- How to stop confiscating code with comments
- ASCII art of welcome message
- Call by reference vs Call by value
- Aggregation : Creating function with subfunctions.