**Main.cpp:**

#include "Disk.h" // Disk class

#include "HanoiStack.h" // HanoiStack class

#include <iostream> // cout, endl

#include <stdlib.h> // atoi

#include <stdio.h> // printf

void initializeGame(HanoiStack&, int); // Initializes stackA with how many disks are entered by the user

int pickUpDisk(); // shows choice for which stack to pick a disk up

int putDownDisk(); // shows choice for which stack to place a disk

void moveDisk(HanoiStack&, HanoiStack&, HanoiStack&); // UI for moving disks from stack to stack

void printDirections(); // prints the directions for the game

bool checkWin(HanoiStack&, HanoiStack&, int); // checks if stackB or stackC are winning stacks

void rePush(HanoiStack&, HanoiStack&, HanoiStack&, int, Disk); // pushes a disk back onto its

// orignial stack to if choice of disk placement was

// incorrect acording to the rules of the game

void solveHanoi(int, int, int, int); // Shows the moves that are needed to complete the game correctly

int main(int argc, char\*\* argv) {

int count = argc; // amount of arguments at start of the program

if (count != 2) {

std::perror("Cannot Execute Program: Error Code\n\t--Amount of arguments incorrect\n\t--To run:\n\t\t./main numOfDisks\n");

exit(1); // checks for correct argument input at start of program

}

int numOfDisks = atoi(argv[1]); // takes the second argument as the # of disks to start with

bool done = false; // game flag

int choice;

if (numOfDisks < 1 || numOfDisks > 10) {

std::perror("Cannot Execute Program: Error Code\n\t--Second Argument Invalid Input\n\t--To run:\n\t\tnumOfDisks must be between 1 and 10 inclusive\n");

exit(1); // checks for correct argument input at start of program

}

HanoiStack stackA, stackB, stackC; // three stacks for game

initializeGame(stackA, numOfDisks); // initializes stackA from input

std::cout << "\tWelcome to the Towers of Hanoi!\n\n";

std::cout << "Here is how to play: " << std::endl;

printDirections(); // prints the directions

do {

std::cout << "A." << std::endl;

std::cout << stackA;

std::cout << "B." << std::endl;

std::cout << stackB;

std::cout << "C." << std::endl;

std::cout << stackC;

std::cout << "Options:" << std::endl;

std::cout << "1. Move a disk" << std::endl;

std::cout << "2. Print Optimal Solution" << std::endl;

std::cout << "3. Quit" << std::endl;

do {

std::cout << "Enter your choice (1-3): ";

std::cin >> choice;

std::cout << std::endl;

} while (choice < 1 || choice > 3); // input validation

switch (choice) {

case 1:

moveDisk(stackA, stackB, stackC); // main game UI

break;

case 2:

std::cout << "Here is the solution: Take notes!!!!!" << std::endl;

solveHanoi(numOfDisks, 1, 2, 3);

break;

case 3:

done = true; // game quit

break;

}

if (checkWin(stackB, stackC, numOfDisks)) { // checks to see if win condition satisfied

std::cout << stackA;

std::cout << stackB; // prints the stacks

std::cout << stackC;

std::cout << "YOU WIN!!!!!!" << std::endl;

done = true;

}

} while (!done);

return 0;

}

/\*

\* solveHanoi Function:

\* Recursive algorithm that tells the user the optimal moves in order in order to complete the game

\*/

void solveHanoi(int n, int start, int end, int tmp) {

if (n >= 1) {

solveHanoi(n - 1, start, tmp, end);

printf("Move disk %d from stack %d to stack %d\n", n, start, end);

solveHanoi(n - 1, tmp, end, start);

}

}

/\*

\* initializeGame Function:

\* Takes the input of how many disks to start with and initializes stackA with that many disks

\*/

void initializeGame(HanoiStack &s, int numOfDisks) {

for (int i = numOfDisks; i >= 1; i--) {

Disk dk(i);

s.push(dk);

}

}

/\*

\* moveDisk Function:

\* Takes all three stacks and starts the UI for the user to move a disk

\*/

void moveDisk(HanoiStack &a, HanoiStack &b, HanoiStack &c) {

int choiceStack = pickUpDisk(); // Returns what stack a disk is being taken from

int lastChoice = 0;

Disk hold;

if (choiceStack == 1) {

hold = a.pop();

lastChoice = 1;

if (hold.getSize() == -1) { // Checks if the stack is empty

return;

}

} else if (choiceStack == 2) {

hold = b.pop();

lastChoice = 2;

if (hold.getSize() == -1) { // Checks if the stack is empty

return;

}

} else if (choiceStack == 3) {

hold = c.pop();

lastChoice = 3;

if (hold.getSize() == -1) { // Checks if the stack is empty

return;

}

}

std::cout << "Current disk being held: " << hold.getSize() << std::endl; // shows user disk

choiceStack = putDownDisk(); // Returns what stack a disk a being placed on

if (choiceStack == 1) {

if (a.seeTop() > hold.getSize()) {

a.push(hold);

return;

} else

rePush(a, b, c, lastChoice, hold); // Pushes a disk back on its original stack

}

else if (choiceStack == 2) {

if (b.seeTop() > hold.getSize()) {

b.push(hold);

return;

} else

rePush(a, b, c, lastChoice, hold); // Pushes a disk back on its original stack

} else if (choiceStack == 3) {

if (c.seeTop() > hold.getSize()) {

c.push(hold);

return;

} else

rePush(a, b, c, lastChoice, hold); // Pushes a disk back on its original stack

}

}

/\*

\* printDirections Function:

\* Prints the directions of the game for the user to understand how to play

\*/

void printDirections() {

std::cout << "Rules of the Game:" << std::endl;

std::cout << "1. You can only move one disk at a time (taking the top disk from a single " <<

"stack and you must place it onto a stack before taking another disk)" << std::endl;

std::cout << "2. No disk can be placed on a disk smaller than itself" << std::endl;

std::cout << "How to Win:" << std::endl;

std::cout << "--Create the original pyramid again on the second (Stack B) or the third " <<

"(Stack C) stack to finish the game" << std::endl;

std::cout << "--Or just select option to show optimal solution" << std::endl;

}

/\*

\* pickUpDisk Function:

\* UI for user to input what stack to take a disk from

\*/

int pickUpDisk() {

int choice;

do {

std::cout << "Which stack would you like to take a disk from? (1-3)" << std::endl;

std::cout << "1. A\t2. B\t3. C" << std::endl;

std::cin >> choice;

if (choice == 1)

return 1;

else if (choice == 2)

return 2;

else if (choice == 3)

return 3;

} while (choice < 1 || choice > 3);

}

/\*

\* putDownDisk Function:

\* UI for user to input what stack to place a disk on

\*/

int putDownDisk() {

int choice;

do {

std::cout << "Which stack would you like to place the disk? (1-3)" << std::endl;

std::cout << "1. A\t2. B\t3. C" << std::endl;

std::cin >> choice;

if (choice == 1)

return 1;

else if (choice == 2)

return 2;

else if (choice == 3)

return 3;

} while (choice < 1 || choice > 3);

}

/\*

\* checkWin Function:

\* Uses internal function for stackB and stackC to see if they are winning formations

\*/

bool checkWin(HanoiStack &b, HanoiStack &c, int numOfDisks) {

if (b.win(numOfDisks) || c.win(numOfDisks))

return true;

return false;

}

/\*

\* rePush Function:

\* Pushes the disk being held back on to its orignial stack because stack that was input by user

\* was incorrect

\*/

void rePush(HanoiStack &a, HanoiStack &b, HanoiStack &c, int lastChoice, Disk hold) {

if (lastChoice == 1) {

a.push(hold);

} else if (lastChoice == 2) {

b.push(hold);

} else if (lastChoice == 3) {

c.push(hold);

}

std::cout << "Cannot place bigger disk on top of a smaller disk" << std::endl;

}

**Disk.h:**

#ifndef \_DISK\_H\_

#define \_DISK\_H\_

class Disk {

private:

int size;

public:

Disk(); // Default Ctor

Disk(int); // Constructor to initialize the disk

int getSize(); // Returns the size of the disk

};

#endif

**Disk.cpp:**

#include "Disk.h"

/\*

\* Default Constructor:

\* Takes no arguments and initializes the disk to its default value

\*/

Disk::Disk() {

size = 0;

}

/\*

\* Overloade Constructor:

\* Takes an int and initializes the disk to the passed parameter

\*/

Disk::Disk(int s) {

size = s;

}

/\*

\* getSize Function:

\* Returns the size of the disk (diameter)

\*/

int Disk::getSize() {

return size;

}

**HanoiStack.h:**

#ifndef \_HANOISTACK\_H\_

#define \_HANOISTACK\_H\_

#include "Disk.h" // Disk class

#include <iostream> // cout, cin

class HanoiStack {

friend std::ostream &operator<<(std::ostream&, const HanoiStack&); // Prints the stack

private:

struct DiskNode {

Disk d;

DiskNode\* next;

};

DiskNode\* head;

public:

HanoiStack(); // Default Ctor

HanoiStack(const HanoiStack&); // Copy Ctor

~HanoiStack(); // Destructor

void push(Disk); // Pushes a disk onto the stack

Disk pop(); // Pops the top disk off of the stack and returns a pointer to the

// popped DiskNode

void operator+=(const Disk&); // an alternative to pushing a disk onto the stack

HanoiStack operator=(const HanoiStack&); // overloaded assignment

bool win(int); // Checks if the stack matches the win condition

int seeTop(); // Returns the top disks size of the stack

};

#endif

**HanoiStack.cpp:**

#include "HanoiStack.h"

/\*

\* Default Constructor:

\* Initializes head of stack list

\*/

HanoiStack::HanoiStack() {

head = nullptr;

}

/\*

\* Copy Constructor:

\* Allows for deep copy of stack list

\*/

HanoiStack::HanoiStack(const HanoiStack &rhs) {

head = nullptr;

DiskNode\* rhsCursor = rhs.head;

while (rhsCursor) {

push(rhsCursor->d);

rhsCursor = rhsCursor->next;

}

}

/\*

\* Destructor:

\* Deletes dynamically allocated memory

\*/

HanoiStack::~HanoiStack() {

DiskNode\* cursor = head;

while (cursor) {

cursor = cursor->next;

delete head;

head = cursor;

}

}

/\*

\* push(Disk) Function:

\* Pushes a disk onto the stack

\*/

void HanoiStack::push(Disk newDisk) {

DiskNode\* newNode = new DiskNode;

newNode->d = newDisk;

newNode->next = nullptr;

if (head == nullptr) {

head = newNode;

return;

}

DiskNode\* cursor = head;

while (cursor->next) {

cursor = cursor->next;

}

if (newNode->d.getSize() > cursor->d.getSize()) {

std::cout << "Cannot place disk, disk is larger than disk below it" << std::endl;

delete newNode;

return;

}

cursor->next = newNode;

}

/\*

\* pop() Function:

\* Pops a DiskNode off of the stack and returns a DiskNode pointer of the popped DiskNode

\*/

Disk HanoiStack::pop() {

if (head == nullptr) {

std::cout << "No disks on current stack" << std::endl;

Disk bad(-1);

return bad;

}

if (head->next == nullptr) {

Disk rtnMe = head->d;

delete head;

head = nullptr;

return rtnMe;

}

DiskNode\* cursor = head;

DiskNode\* prev = head;

while (cursor->next) {

prev = cursor;

cursor = cursor->next;

}

prev->next = nullptr;

Disk rtnMe = cursor->d;

delete cursor;

return rtnMe;

}

/\*

\* Overloaded operator+=:

\* Allows a different way for a disk to be pushed onto the stack

\*/

void HanoiStack::operator+=(const Disk &rhs) {

push(rhs);

}

/\*

\* Overloaded operator<<:

\* Acts as the print function for the stack, prints the stack visually and all of its contents

\*/

std::ostream& operator<<(std::ostream &os, const HanoiStack &obj) {

if (obj.head == nullptr) {

for (int i = 0; i < 7; i++) {

os << " " << std::endl;

}

os << "=============" << std::endl;

return os;

}

HanoiStack::DiskNode\* cursor = obj.head;

int counter = 0;

while (cursor) {

cursor = cursor->next;

counter++;

}

for (int i = 0, j = counter; i < counter; i++, j--) {

cursor = obj.head;

for (int l = 1; l < j; l++) {

cursor = cursor->next;

}

int limit = (13 - cursor->d.getSize()) / 2;

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

os << " ";

}

for (int q = 0; q < cursor->d.getSize(); q++) {

os << "#";

}

for (int m = 0; m < limit; m++) {

os << " ";

}

os << std::endl;

}

os << "=============" << std::endl;

return os;

}

/\*

\* Overloaded operator=:

\* Allows for different stack lists to be copied over one another

\*/

HanoiStack HanoiStack::operator=(const HanoiStack &rhs) {

DiskNode\* cursor = head;

while (cursor) {

cursor = cursor->next;

delete head;

head = cursor;

}

head = nullptr;

if (this == &rhs) {

std::cout << "Cannot assign, Objects are identical" << std::endl;

return \*this;

}

DiskNode\* rhsCursor = rhs.head;

while (rhsCursor) {

push(rhsCursor->d);

rhsCursor = rhsCursor->next;

}

return \*this;

}

/\*

\* win() Function:

\* Checks to see if the stack matches the win condition and then returns is true or not

\*/

bool HanoiStack::win(int numOfDisks) {

if (head == nullptr) {

return false;

}

int count = 0;

DiskNode\* cursor = head;

while (cursor) {

cursor = cursor->next;

count++;

}

if (count == numOfDisks) {

cursor = head;

for (int i = numOfDisks; i > 0; i--) {

if (cursor->d.getSize() != i) {

return false;

}

cursor = cursor->next;

}

return true;

} else {

return false;

}

}

/\*

\* seeTop() Function:

\* Returns the size of the top disk on the stack

\*/

int HanoiStack::seeTop() {

if (head == nullptr) {

return 11;

}

DiskNode\* cursor = head;

while (cursor->next) {

cursor = cursor->next;

}

return cursor->d.getSize();

}

**Sample Output:**

1.

Welcome to the Towers of Hanoi!

Here is how to play:

Rules of the Game:

1. You can only move one disk at a time (taking the top disk from a single stack and you must place it onto a stack before taking another disk)

2. No disk can be placed on a disk smaller than itself

How to Win:

--Create the original pyramid again on the second (Stack B) or the third (Stack C) stack to finish the game

--Or just select option to show optimal solution

A.

#

##

###

=============

B.

=============

C.

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 1

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

A.

##

###

=============

B.

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 2

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

Cannot place bigger disk on top of a smaller disk

A.

##

###

=============

B.

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 2

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

2

A.

###

=============

B.

##

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 3

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

Cannot place bigger disk on top of a smaller disk

A.

###

=============

B.

##

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

2

Current disk being held: 2

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

1

A.

##

###

=============

B.

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 2

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

2

A.

###

=============

B.

##

=============

C.

#

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

3

Current disk being held: 1

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

2

A.

###

=============

B.

#

##

=============

C.

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 3

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

A.

=============

B.

#

##

=============

C.

###

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

2

Current disk being held: 1

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

1

A.

#

=============

B.

##

=============

C.

###

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

2

Current disk being held: 2

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

A.

#

=============

B.

=============

C.

##

###

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

2

No disks on current stack

A.

#

=============

B.

=============

C.

##

###

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 1

Which stack would you like to take a disk from? (1-3)

1. A 2. B 3. C

1

Current disk being held: 1

Which stack would you like to place the disk? (1-3)

1. A 2. B 3. C

3

=============

=============

#

##

###

=============

YOU WIN!!!!!!

2.

Welcome to the Towers of Hanoi!

Here is how to play:

Rules of the Game:

1. You can only move one disk at a time (taking the top disk from a single stack and you must place it onto a stack before taking another disk)

2. No disk can be placed on a disk smaller than itself

How to Win:

--Create the original pyramid again on the second (Stack B) or the third (Stack C) stack to finish the game

--Or just select option to show optimal solution

A.

#

##

###

####

=============

B.

=============

C.

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 2

Here is the solution: Take notes!!!!!

Move disk 1 from stack 1 to stack 3

Move disk 2 from stack 1 to stack 2

Move disk 1 from stack 3 to stack 2

Move disk 3 from stack 1 to stack 3

Move disk 1 from stack 2 to stack 1

Move disk 2 from stack 2 to stack 3

Move disk 1 from stack 1 to stack 3

Move disk 4 from stack 1 to stack 2

Move disk 1 from stack 3 to stack 2

Move disk 2 from stack 3 to stack 1

Move disk 1 from stack 2 to stack 1

Move disk 3 from stack 3 to stack 2

Move disk 1 from stack 1 to stack 3

Move disk 2 from stack 1 to stack 2

Move disk 1 from stack 3 to stack 2

A.

#

##

###

####

=============

B.

=============

C.

=============

Options:

1. Move a disk

2. Print Optimal Solution

3. Quit

Enter your choice (1-3): 3