stack

学籍番号:22120  
組番号:222  
名前:塚田 勇人

2025-06-23

# 目的

スタックのプッシュ、ポップの動作を、与えられた課題をこなすことによって理解することを このレポートの目的とします。

# 原理

この章では、スタックとハノイの塔の原理について説明します。

## スタック

スタックは、データを一時的に保存するためのデータ構造です。 スタックは、後入れ先出し（LIFOと呼ばれる。Last-In-First-Outの略） の原則に基づいて動作します。 要素の追加は「プッシュ」、要素の取り出しは「ポップ」と呼ばれます。

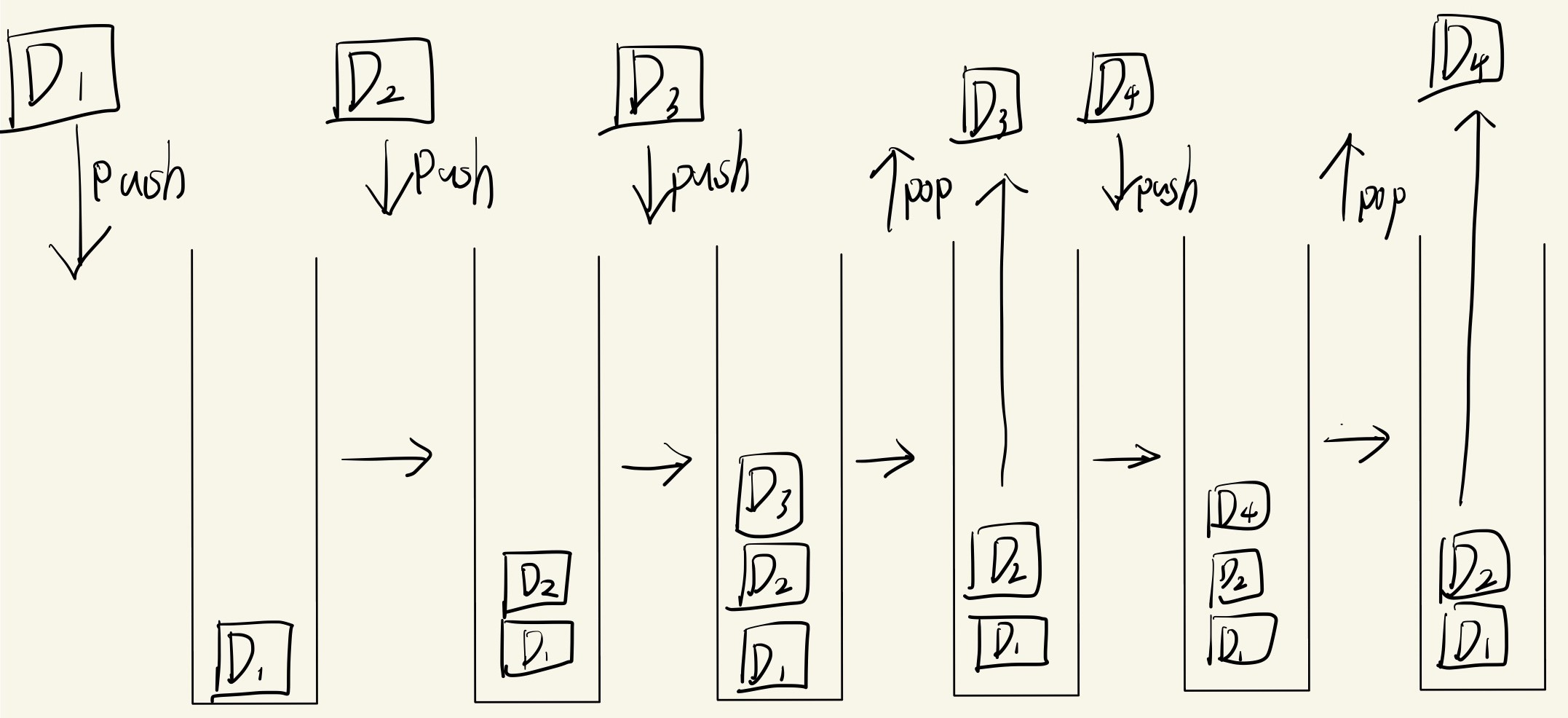


Figure 1: スタックのイメージ

図[1](#fig:stack)は、スタックのイメージを示しています。 スタックに要素を追加すると、新しい要素がスタックの一番上に追加されます。 スタックから要素を取り出すと、一番上の要素が取り出されます。 スタックは、データの一時的な保存や逆順の処理が必要な場合に便利です。  
   スタックと対になるデータ構造として、キューがあります。 キューは、先入れ先出し（FIFOと呼ばれる。First-In-First-Outの略） の原則に基づいて動作します。要素の追加は「エンキュー」、 要素の取り出しは「デキュー」と呼ばれます。

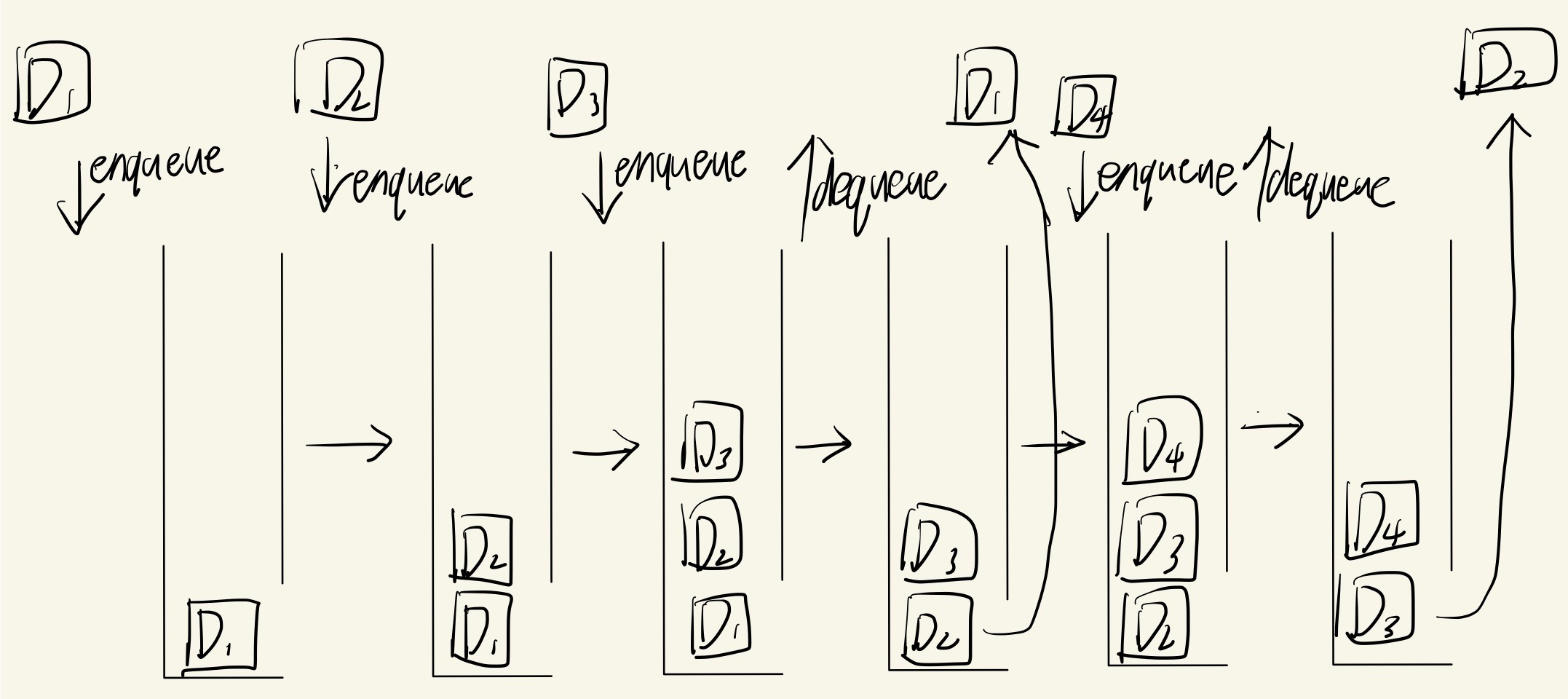


Figure 2: キューのイメージ

図[2](#fig:queue)は、キューのイメージを示しています。 キューに要素を追加すると、新しい要素がキューの一番上に追加されます。 キューから要素を削除すると、一番下の要素が取り出されます。  
   スタックとキューの違いとして、 スタックは後入れ先出しの原則に基づいて、要素を取り出すときに一番上の 要素が取り出されるのに対して、キューは先入れ先出しの原則に基づいて、 要素を取り出すときに一番下の要素が取り出されることが挙げられます。

## ハノイの塔

ハノイの塔は、フランスの数学者エドゥアール・リュカによって考案されたパズルです。 ハノイの塔は、3本の杭と、いくつかの円盤からなります。 円盤は、大きいものから小さいものまで順番に積まれています。 目的は、最初の杭に積まれた円盤を、できるだけすくない手数で他の杭に移動することです。 ただし、次のルールに従う必要があります。

* 1回に1枚の円盤のみを移動できる。
* 小さい円盤の上に大きい円盤を積んではいけない。

今回はスタックを用いて、ハノイの塔のパズルを実装するプログラムを作成します。

# 実験環境

stackの課題のプログラムを実装した環境を表[1](#tb:environment)に示します。

実験環境

| OS | Windows 10 |
| --- | --- |
| CPU | AMD Ryzen 7 5800H |
| メモリ | 16GB |
| コンパイラ | gcc version 11.4.0 |
| エディタ | Visual Studio Code |

# プログラムの設計と説明

このプログラムでは統一してリスト[[define]](#define)のリストのように、 塔の高さと数の値と構造体を定義しています。

#define HEIGHT 5  
#define TOWERS 3  
  
typedef struct {  
 int data[HEIGHT];  
 int volume;  
} Stack;

プログラムで使われている関数について説明します。

## スタックプログラムの関数

スタックプログラムでは表[2](#tb:stackfunction)の関数を使用しています。

スタックプログラムで使用されている関数

| 関数名 | 処理内容 |
| --- | --- |
| init | 構造体を初期化する |
| push | 構造体にデータをpushする |
| pop | 構造体からデータをpopする |
| printStack | 構造体の中身を表示する |
| pushTest | pushできるかどうかを判定する |
| popTest | popできるかどうかを判定する |

それぞれの関数について説明します。

### init関数

init関数は表[3](#tb:init)のように定義され、 リスト[[initcode]](#initcode)のように実装されています。

init関数

| 関数名 | init |
| --- | --- |
| 型 | void |
| 引数1 | \*stack:表示する構造体のポインタ |
| 戻り値 | なし |

void init(Stack \*stack) {  
 int i;  
 for (i = 0; i < HEIGHT; i++) {  
 stack->data[i] = 0; // スタックのすべての要素を0にする  
 }  
 stack->volume = 0; // スタックに格納されているデータ数を0にする  
 }

init関数は次の手順の処理が実装されています。

1. スタックのすべての要素を0にする。
2. スタックに格納されているデータ数を0にする。

### push関数

push関数は表[4](#tb:push)のように定義され、 リスト[[pushcode]](#pushcode)のように実装されています。

push関数

| 関数名 | push |
| --- | --- |
| 型 | int |
| 引数1 | \*stack:pushする構造体のポインタ |
| 引数2 | number:pushするデータ |
| 戻り値 | 成功したら0, 失敗したら-1 |

int push(Stack \*stack, int number) {  
 if (stack->volume == HEIGHT) { // これ以上スタックできないなら-1を返す  
 return -1;  
 }  
 stack->data[stack->volume] = number; // データを最上位に積み込む  
 stack->volume++; // データの個数を増やす  
 return 0;  
 }

push関数は次の手順の処理が実装されています。

1. スタックが満杯かどうかを判定し、満杯なら-1を返し関数を終了する。
2. スタックにデータを積み込む。
3. データの個数を増やす。
4. 0を返し、関数を終了する。

### pop関数

pop関数は表[5](#tb:pop)のように定義され、 リスト[[popcode]](#popcode)のように実装されています。

pop関数

| 関数名 | pop |
| --- | --- |
| 型 | int |
| 引数1 | \*stack:popする構造体のポインタ |
| 戻り値 | 成功したらpopしたデータ, 失敗したら-1 |

int pop(Stack \*stack) {  
 int num;  
 if (stack->volume == 0) {  
 return -1;  
 }  
 stack->volume--; // 格納されているデータ個数のカウントを減らす  
 num = stack->data[stack->volume]; // 取り出すデータを取り出す  
 stack->data[stack->volume] = 0; // 取り出した場所を初期化する  
 return num;  
}

pop関数は次の手順の処理が実装されています。

1. スタックが空かどうかを判定し、空なら-1を返し関数を 終了する。
2. 格納されているデータ個数のカウントを減らす。
3. スタックからデータを取り出す。
4. データを取り出した場所を初期化する。
5. 取り出したデータを返し、関数を終了する。

### printStack関数

printstack関数は表[6](#tb:printstack)のように定義され、 リスト[[printstackcode]](#printstackcode)のように実装されています。

printStack関数

| 関数名 | printStack |
| --- | --- |
| 型 | void |
| 引数1 | \*stack:表示する構造体のポインタ |
| 戻り値 | なし |

void printStack(Stack \*stack) {  
 int i;  
 printf("[ ");  
 for (i = 0; i < stack->volume; i++) { // スタックに格納されている値を  
 printf("%d ", stack->data[i]); // スタックされている順番に1行に表示  
 }  
 printf("]\n");  
}

printstack関数は次の手順の処理が実装されています。

1. スタックに格納されている値を表示する。

### pushTest関数

pushTest関数は表[7](#tb:pushtest)のように定義され、 リスト[[pushtestcode]](#pushtestcode)のように実装されています。

pushTest関数

| 関数名 | pushTest |
| --- | --- |
| 型 | bool |
| 引数1 | \*stack:判定する構造体 |
| 戻り値 | pushできるならtrue, できないならfalse |

bool pushTest(Stack stack) {  
 if (stack.volume == HEIGHT) {  
 return false;  
 } else {  
 return true;  
 }  
}

pushTest関数は次の手順の処理が実装されています。

1. スタックが満杯かどうかを判定する。
2. 満杯ならfalseを返し、そうでなければtrueを返し、関数を終了する。

### popTest関数

poptest関数は表[8](#tb:poptest)のように定義され、 リスト[[poptestcode]](#poptestcode)のように実装されています。

popTest関数

| 関数名 | popTest |
| --- | --- |
| 型 | int |
| 引数1 | \*stack:popする構造体のポインタ |
| 戻り値 | 成功したらpopしたデータ, 失敗したら-1 |

bool popTest(Stack stack) {  
 if (stack.volume == 0) {  
 return false;  
 } else {  
 return true;  
 }  
}

poptest関数は次の手順の処理が実装されています。

1. スタックが空かどうかを判定する。
2. 空ならfalseを返し、そうでなければtrueを返し、関数を終了する。

## ハノイの塔プログラムの関数

ハノイの塔プログラムでは章[4.1](#sec:stack)の関数に加えて表[9](#tb:hanoifunction)の関数を使用している。

ハノイの塔プログラムで使用されている関数

| 関数名 | 処理内容 |
| --- | --- |
| top | 塔の最上位の値を返す関数 |
| enableStack | 値を移動できるかどうかを判定する関数 |
| checkFinish | クリアしているかどうかを判定する関数 |

それぞれの関数について説明する。

### top関数

top関数は表[10](#tb:top)のように定義され、 リスト[[topcode]](#topcode)のように実装されています。

top関数

| 関数名 | top |
| --- | --- |
| 型 | int |
| 引数1 | tower:塔の構造体 |
| 戻り値 | 塔の最上位の値 |

int top(Stack tower) {  
 return tower.data[tower.volume - 1];  
}

top関数は次の手順の処理が実装されています。

1. 塔の最上位の値を返し、関数を終了する。

### enableStack関数

enableStack関数は表[11](#tb:enableStack)のように定義され、 リスト[[enableStackcode]](#enableStackcode)のように実装されています。

enableStack関数

| 関数名 | enableStack |
| --- | --- |
| 型 | int |
| 引数1 | fromTower:移動元の塔の構造体 |
| 引数2 | toTower:移動先の塔の構造体 |
| 戻り値 | 可能なら1, 不可能なら0 |

int enableStack(Stack fromTower, Stack toTower) {  
 if (fromTower.volume != 0 && toTower.volume != HEIGHT &&  
 top(fromTower) < top(toTower)) {  
 return 1; /\* 移動可能である条件に応じて返り値を返す \*/  
 } else {  
 return 0;  
 }  
}

enableStack関数は次の手順の処理が実装されています。

1. 移動可能かどうか判定する。移動可能かどうかの条件は次の通りである。
   * 移動元の塔が空でない。
   * 移動先の塔が満杯でない。
   * 移動元の塔の最上位の値が移動先の塔の最上位の値より小さい。
2. 移動可能なら1を返し、そうでなければ0を返し、関数を終了する。

### checkFinish関数

checkFinish関数は表[12](#tb:checkFinish)のように定義され、 リスト[[checkFinishcode]](#checkFinishcode)のように実装されています。

checkFinish関数

| 関数名 | checkFinish |
| --- | --- |
| 型 | int |
| 引数1 | tower:塔の構造体 |
| 引数2 | blocks:ブロックの数 |
| 戻り値 | クリアしているなら1, していないなら0 |

int checkFinish(Stack tower, int blocks) {  
 int i;  
 int check = blocks;  
 for (i = 0; i < blocks; i++) {  
 if (tower.data[i] == check--) {  
 } else {  
 return 0;  
 }  
 }  
 return 1;  
 // ブロックが初期状態と同じ状態かチェックする  
}

checkFinish関数は次の手順の処理が実装されています。

1. ブロックが初期状態と同じ状態かチェックする。
2. ブロックが初期状態と同じ状態なら1を返し、 そうでなければ0を返し、関数を終了する。

# プログラム

この章では、それぞれのプログラムのメイン関数を示し、 処理内容について説明します。 また、それぞれのプログラムの全文はここに記すと長くなるのでレポートの 最後に付録として示します。

## スタックプログラム

リスト[[stackmaincode]](#stackmaincode)は、スタックプログラムのメイン関数を示しています。

int main(void) {  
 Stack stack;  
 int i, check;  
 init(&stack);  
 printf("MAX STACK NUM:%d\n", HEIGHT);  
 for (i = 10; i < 40; i += 10) {  
 if (pushTest(stack) == true) {  
 printf("pushed %d\n", i);  
 push(&stack, i);  
 } else {  
 printf("error\n");  
 }  
 }  
  
 printf("The data is stacked in order?\n");  
 printStack(&stack);  
  
 printf("The data is retrieved in order?\n");  
 for (i = 0; i < 4; i++) {  
 if (popTest(stack) == true) {  
 printf("popped %d\n", pop(&stack));  
 } else {  
 printf(  
 "What happens when you try to retrieve data when it is empty"  
 "\n");  
 printf("error\n");  
 }  
 }  
  
 for (i = 10; i < 70; i += 10) {  
 if (pushTest(stack) == true) {  
 printf("pushed %d\n", i);  
 push(&stack, i);  
 } else {  
 printf(  
 "What happens when you try to add data when it is full"  
 "\n");  
 printf("error\n");  
 }  
 }  
 return 0;  
}

メイン関数では次の処理を実装しています。

1. スタックを初期化する。
2. スタックに入るデータの最大数を表示する。
3. スタックにデータを10から30プッシュする。
4. スタックの要素を表示する。
5. スタックからデータを4回ポップする。
6. スタックにデータを10から60プッシュする。

それぞれの事項について説明する。

### スタックを初期化する処理

リスト[[stackmaincode]](#stackmaincode)の4行目で、スタックを初期化しています。 init関数（節[4.1.1](#sec:init)）を呼び出すことで、スタックを初期化して います。初期化を行わないとスタックの最初の値が不定値になるため、 この動作は重要です。

### スタックに入るデータの最大数を表示する処理

リスト[[stackmaincode]](#stackmaincode)の5行目で、スタックに入るデータの最大数を表示しています。 これをすることで、実行結果を見たときにスタックにどれだけデータを プッシュできるかがわかります。

### スタックにデータを10から30プッシュする処理

リスト[[stackmaincode]](#stackmaincode)の6行目から13行目で、10から30までの データをpush関数（節[4.1.2](#sec:push)）を呼び出すことでプッシュ しています。プッシュする前にpushTest関数（節[4.1.5](#sec:pushTest)）を呼び出し、 プッシュできるかどうか判定することでスタックの最大値を超えて プッシュすることを防いでいます。この部分の処理では、3つのデータしか プッシュしないのでエラーメッセージが表示されることはありません。

### スタックの要素を表示する処理

リスト[[stackmaincode]](#stackmaincode)の15, 16行目で、printStack関数 （節[4.1.4](#sec:printStack)）を呼び出すことでスタックの要素を 表示しています。  
  
   後の節[6.1](#sec:stackresult)スタックプログラム の実行結果 で挙げられている、確認する事項の内の以下の事項を満たす ように 節[5.1.3](#sec:pushprocess1)スタックにデータを10から30プッシュ する処理 、節[5.1.4](#sec:printStackprocess)スタックの要素を表示する 処理 を実装しています。

* データ順にスタックされているか。

### スタックからデータを4回ポップする処理

リスト[[stackmaincode]](#stackmaincode)の18行目から28行目で、4回pop関数 （節[4.1.3](#sec:pop)）を呼び出し上記の処理でプッシュした30から10までの データをポップしています。 また、ポップする前にpopTest関数（節[4.1.6](#sec:popTest)）を呼び出し、 ポップできるかどうか判定することで、スタックが空の時にデータを ポップしようとしたときにスタックの範囲外のデータをポップしないように しています。なので4回目のポップでは、スタックが空の時にデータを ポップしようとしているので、popTest関数がfalseを返し、 エラーメッセージを表示するように実装されています。  
  
   後の節[6.1](#sec:stackresult)スタックプログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすようにこの処理を 実装しています。

* データ順に取り出せているか。
* スタックが空の時にデータを取り出そうとしたときにどのような 動作をするか。

### スタックにデータを10から60プッシュする処理

リスト[[stackmaincode]](#stackmaincode)の30行目から40行目で、10から60までの データを節[5.1.3](#sec:pushprocess1)で説明した処理と同様にプッシュ しています。60のデータをプッシュする際には、スタックが満杯の状態 でプッシュしようとするため、pushTest関数がfalseを返し、 エラーメッセージを表示するように実装されています。  
  
   後の節[6.1](#sec:stackresult)スタックプログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすようにこの処理を 実装しています。

* 満杯の時にデータを追加しようとしたときにどのような動作を するか。

   以上の処理を行うことで、スタックプログラムのメイン関数を 実装しています。 また、このメイン関数の工夫点としてテキストで示されていた関数の内容は テキスト通りで中身を一切変えずに、他に独自の関数を追加して確認 すべき事項を確認できるようにしている点です。 これをすることでメイン関数で何をしているかがわかりやすく、よりテキスト に沿ったプログラムを実装できるようになります。

## ハノイの塔プログラム

リスト[[hanoimaincode]](#hanoimaincode)は、ハノイの塔プログラムのメイン関数を 示しています。

int main(void) {  
 int i;  
 int count = 1;  
 int fromNumber, toNumber;  
 int tempNumber;  
 int blocks;  
 Stack tower[TOWERS];  
  
 printf("Select the number of steps(3, 4, 5):");  
 while (scanf("%d", &blocks) != 1 || blocks < 3 || blocks > 5) {  
 while (getchar() != '\n');  
 printf("error please rewrite\n");  
 printf("Select the number of steps(3, 4, 5):");  
 }  
 /\*3 塔を初期化する\*/  
 for (i = 0; i < TOWERS; i++) {  
 init(&tower[i]);  
 }  
 /\*第１塔に決められた個数をスタックする\*/  
 i = blocks;  
 while (i != 0) {  
 push(&tower[0], i--);  
 }  
  
 /\*塔の初期状態を表示する\*/  
 for (i = 0; i < TOWERS; i++) {  
 printf("%d:", i + 1);  
 printStack(&tower[i]);  
 }  
 while (1) {  
 // 今，何回目の移動であるかを数える．  
 printf("%dth\n", count++);  
 // 移動元と移動先を受け取る  
 while (1) {  
 printf("enter the source and destination towers.[? ?]:");  
 while (scanf("%d%d", &fromNumber, &toNumber) != 2 || toNumber > 3 ||  
 toNumber < 1 || fromNumber > 3 || fromNumber < 1) {  
 while (getchar() != '\n');  
 printf("error please rewrite\n");  
 printf("enter the source and destination towers.[? ?]:");  
 }  
 // 移動元の塔から移動先の塔にデータを移動させる  
 if (enableStack(tower[fromNumber - 1], tower[toNumber - 1]) == 1) {  
 tower[toNumber - 1].data[tower[toNumber - 1].volume] =  
 pop(&tower[fromNumber - 1]);  
 tower[toNumber - 1].volume++;  
 break;  
 } else {  
 printf("Cannot move\n");  
 }  
 }  
 // 現在の塔の状態を表示する  
 for (i = 0; i < TOWERS; i++) {  
 printf("%d:", i + 1);  
 printStack(&tower[i]);  
 }  
 // クリア判定をする  
 for (i = 1; i < TOWERS; i++) {  
 if (checkFinish(tower[i], blocks) == 1) {  
 printf("clear!\n");  
 return 0;  
 }  
 }  
 }  
}

メイン関数では次の処理を実装しています

1. ユーザーにブロックの数の入力を求める。
2. 塔を初期化する。
3. 1. で入力されたブロックの数をプッシュする。
4. 塔の初期状態を表示する。
5. 以下の処理をクリアするまで繰り返す。
   1. 今何回目の移動であるかを数える。
   2. ユーザーに移動元と移動先の入力を求める。
   3. 移動元の塔から移動先の塔にデータを移動させる。
   4. 塔の状態を表示する。
   5. クリアしているかどうか判定する。
   6. クリアしていたらループを抜け、clear!と表示する。

それぞれの事項について説明します。なお、説明の中で [[hanoimaincode]](#hanoimaincode)の7行目で定義されている構造体配列tower[0]、 tower[1]、tower[2]をそれぞれ第1塔、第2塔、第3塔と呼ぶこととします。

### ユーザーにブロックの数の入力を求める処理

リスト[[hanoimaincode]](#hanoimaincode)の9行目から14行目で、ユーザーにブロックの 数の入力を求める処理を実装しています。 この処理では、3から5の数を入力するように求めています。そのために 9行目に3, 4, 5の数を入力するように求めるメッセージを表示しています。 また、10行目のwhileの条件式で3, 4, 5以外の数を入力した場合、エラー メッセージを表示し、再度3, 4, 5の数を入力するように求めるように 実装しています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* 段数を正しく受け取れているか。
* ユーザーから入力を受け取る際に、適当ではない入力を 拒否できているか。

### 塔を初期化する処理

リスト[[hanoimaincode]](#hanoimaincode)の15行目から18行目で、塔を初期化する 処理を実装しています。init関数（節[4.1.1](#sec:init)）をfor文で使い、 すべての塔が初期化されるように呼び出すことで、 3つの塔を初期化しています。初期化を行わないとそれぞれの塔の値が 不定値になるため、この処理は必須です。

### 1. で入力されたブロックの数をプッシュする処理

リスト[[hanoimaincode]](#hanoimaincode)の19行目から23行目で、第1塔に 節[5.2.1](#sec:blockinput)ユーザーにブロックの数の入力を求める関数  で入力されたブロックの数をプッシュする処理を 実装しています。この処理で重要な点は、ブロックを数が大きい方から プッシュしていることです。これをすることで、ハノイの塔のパズルの ルールに従い、大きいブロックが下に、小さいブロックが上に積まれる という状態を再現しています。

### 塔の初期状態を表示する処理

リスト[[hanoimaincode]](#hanoimaincode)の25行目から29行目で、塔の初期状態を 表示する処理を実装しています。printStack関数 （節[4.1.4](#sec:printStack)）をfor文で使い、すべての塔の状態を 表示するように呼び出すことで、3つの塔の状態を表示しています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* 塔の状態を正しく表示できているか。

### 以下の処理をクリアするまで繰り返す処理

リスト[[hanoimaincode]](#hanoimaincode)の30行目から64行目で、以下の節で 説明する処理をクリアするまで繰り返す処理を実装しています。

### 今何回目の移動であるかを数える処理

リスト[[hanoimaincode]](#hanoimaincode)の32行目で、今何回目の移動であるかを 数え、表示する処理を実装しています。この処理は、ハノイの塔の パズルを解く際に何回目の移動であるかを数えるために実装しています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* 回数を正しく数えられているか。

### ユーザーに移動元と移動先の入力を求める処理

リスト[[hanoimaincode]](#hanoimaincode)の30行目から41行目で、ユーザーに移動元と 移動先の入力を求める処理を実装しています。この処理では、 1から3の数のみを入力するように求めています。そのために36行目と 37行目のwhileの条件式で1から3以外の数を入力した場合、エラー メッセージを表示し、再度1から3の数を入力するように求めるように 実装しています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* ユーザーから入力を受け取る際に、適当ではない入力を 拒否できているか。

### 移動元の塔から移動先の塔にデータを移動させる処理

リスト[[hanoimaincode]](#hanoimaincode)の42行目から51行目で、 節[5.2.7](#sec:movetowerinput)で入力された移動元の塔から移動先の塔に データを移動させる処理を実装しています。この処理では、 enableStack関数（節[4.2.2](#sec:enableStack)）を呼び出し、移動できるか どうか判定しています。移動できる場合は、pop関数（節[4.1.3](#sec:pop)） を使い移動元の塔からデータをポップし、ポップしたデータを push関数（節[4.1.2](#sec:push)）を使い移動先の塔にプッシュすること でデータを移動させています。 移動できない場合は、エラーメッセージを表示するように実装しています。 また、構造体の添え字に入力された数から1を引いた数を使っています。 これは、塔の番号は1から始まるのに対して、構造体配列の添え字は 0から始まるためです。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* 移動元と移動先を受け取り、データを移動させることができるか。

### 塔の状態を表示する処理

リスト[[hanoimaincode]](#hanoimaincode)の52行目から56行目で、塔の状態を 表示する処理を実装しています。節[5.2.4](#sec:printTower)で説明した処理と 同様に、printStack関数をfor文を使い、すべての塔の状態を表示するように 呼び出すことで、3つの塔の状態を表示しています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* 塔の状態を正しく表示できているか。

### クリアしているかどうか判定する処理

リスト[[hanoimaincode]](#hanoimaincode)の57行目から63行目で、クリアしているか どうか判定する処理を実装しています。この処理では、checkFinish関数 （節[4.2.3](#sec:checkFinish)）をfor文を使い、第1塔以外の塔の状態を チェックしています。クリアしている場合は、clear!と表示して プログラムを終了します。ものによってはパズルのルールとして 第1塔のブロックを第3塔に移動させることがクリアとされているものも ありますが、ここでは第2塔と第3塔どちらに移動させてもクリアになるよう にしています。  
  
   後の節[6.2](#sec:hanoiresult)ハノイの塔プログラムの実行結果 で 挙げられている、確認する事項の内の以下の事項を満たすように この処理を実装しています。

* クリアしているかどうかを判定できているか。

# 実行結果

この章では、それぞれのプログラムの実行結果を示す。

## スタックプログラムの実行結果

スタックのプログラムを 以下の事項を確認できるように実行した結果をリスト[[resultstack]](#resultstack)に示す。

* データ順にスタックされているか。
* データ順に取り出せているか。
* スタックが空の時にデータを取り出そうとしたときにどのような動作をするか。
* 満杯の時にデータを追加しようとしたときにどのような動作をするか。

./result\_stack.txt

上記の事項について説明をします。

### データ順にスタックされているか

リスト[[resultstack]](#resultstack)の2行目から6行目までの出力を見ると、 10, 20, 30がプッシュされた後、スタック内でその順番で表示されているので データ順にスタックされていることがわかります。

### データ順に取り出せているか

リスト[[resultstack]](#resultstack)の7行目から10行目までの出力を見ると、 30, 20, 10が順にポップされているので、 データ順に取り出せていることがわかります。

### スタックが空の時にデータを取り出そうとしたときにどのような動作をするか

リスト[[resultstack]](#resultstack)の11行目から12行目までの出力を見ると、 スタックが空の時にデータを取り出そうとするとエラーが表示されるので、 エラーになることがわかります。しかし、この動作はプログラムでスタックが 空の時にポップしないようにしているため起こる動作です。もし、そのように プログラムしていなかった場合、配列の範囲外のポインタにアクセスするため プログラムがクラッシュしたり、まったく別のデータをポップしてしまう 可能性があるため、エラーを表示するようにプログラムしないと 大変危険です。

### 満杯の時にデータを追加しようとしたときにどのような動作をするか

リスト[[resultstack]](#resultstack)の13行目から19行目までの出力を見ると、 10から50までの5つの値をプッシュした後、もう一つの値をプッシュ しようとしたときにエラーが表示されるので、 エラーになることがわかります。 しかし、この動作もプログラムでスタックが満杯の時にプッシュしないように しているため起こる動作です。もし、そのようにプログラムしていなかった 場合、配列の範囲外のポインタにアクセスするため、プログラムが クラッシュしたり、まったく別のプログラムにプッシュして他のデータを 書き換えてしまう可能性があるため、エラーを表示するようにプログラム しないと大変危険です。  
  
   また、工夫点として実行結果を英語で表示することで、だれが見ても わかりやすくグローバル化している社会に対応するようにした。

## ハノイの塔プログラムの実行結果

ハノイの塔のプログラムを以下の事項を確認できるように実行した結果を リスト[[resulthanoi1]](#resulthanoi1), [[resulthanoi2]](#resulthanoi2)に示す。

* 段数を正しく受け取れているか。
* 回数を正しく数えられているか。
* 移動元と移動先を受け取り、データを移動させることができるか。
* 塔の状態を正しく表示できているか。
* クリアしているかどうかを判定できているか。
* ユーザーから入力を受け取る際に、適当ではない入力を拒否できているか。

./result\_tower.1.txt

./result\_tower.2.txt

上記の事項について説明をします。

### 段数を正しく受け取れているか

リスト[[resulthanoi1]](#resulthanoi1)の5行目から8行目までの出力を見ると、 段数で3段を選んだ時に、初期状態として、3, 2, 1がスタックされた塔が 表示されています。また、リスト[[resulthanoi2]](#resulthanoi2)の1行目から4行目まで の出力をみると、段数で4段を選んだ時に、初期状態として、4, 3, 2, 1が スタックされた塔が表示されています。したがって、 段数を正しく受け取れていることがわかります。

### 回数を正しく数えられているか

リスト[[resulthanoi1]](#resulthanoi1)の9行目、20行目や25行目などの出力を見ると、 1回目から順に数えられているので、回数を正しく数えられていることが わかります。

### 移動元と移動先を受け取り、データを移動させることができるか

リスト[[resulthanoi1]](#resulthanoi1)の16行目から19行目までの出力を見ると、 1  3と入力を受け取ると、1から3にデータが移動されているので、 移動元と移動先を受け取り、データを移動させることができているのが わかります。

### 塔の状態を正しく表示できているか

リスト[[resulthanoi1]](#resulthanoi1)の17行目から19行目までの出力を見ると、 初期状態と比べて、1つのデータを1から3に移動させた後の塔の状態が 表示されているため、塔の状態を正しく表示できていることがわかります。

### クリアしているかどうかを判定できているか

リスト[[resulthanoi1]](#resulthanoi1)の46行目から50行目までの出力を見ると、 1以外の塔で3, 2, 1の順でスタックされたときにクリアと表示され プログラムが終了しているので、クリアしているかどうかを判定できている ことがわかります。

### ユーザーから入力を受け取る際に、適当ではない入力を拒否できているか

リスト[[resulthanoi1]](#resulthanoi1)の1行目から5行目までの出力を見ると、 与えられている選択肢の3, 4, 5以外の数字を入力すると、もう一度入力を 促すようになっています。また、 リスト[[resulthanoi1]](#resulthanoi1)の10行目から16行目までの出力を見ると、 [4.2.2](#sec:enableStack) enableStack関数節で説明した条件を満たさない移動元と移動先を 選んでいない場合や、数字以外のものを入力したときに、移動できないという 旨のメッセージが表示されてるようになっています。したがって、 ユーザーから入力を受け取る際に、適当ではない入力を拒否できていることが わかります。  
  
   また、工夫点として[6.1](#sec:stackresult) スタックプログラム節と 同じように、実行結果を英語で表示することで、だれが見ても わかりやすくグローバル化している社会に対応するようにしました。

# 考察

この章では、スタックの利点と欠点に関する考察を述べます。

## スタックの利点

スタックの先入れ後出しの性質を利用することで、ハノイの 塔における、下にあるデータは上のデータを移動させないと、 取り出すことができない、というゲーム性を実装することが できます。このように、スタックは、データの順番を保持する 必要がある際に、非常に有用です。

## スタックの欠点

スタックは空の状態でポップする、満杯の状態でプッシュする、 という動作をした場合にプログラムがクラッシュしたり、他の データを書き換えてしまうという欠点があります。これは、 動作をする前に、スタックが空かどうか、満杯かどうかを判定 することで解決することができます。

## まとめ

上記より、課題をこなすことでスタックに関する理解を深める ことができた。 またプログラムを作成する時に、ユーザーからの 入力を受け取る際に適当ではない入力を拒否することや、 オーバーフローを防ぐようにすることで、プログラムの堅牢性を 守ることが重要だと学ぶことができました。

# 付録:プログラム全文

この章では、レポート本文で示したプログラムの全文を 示します。

## スタックプログラム

スタックプログラムのソースコードを リスト[[stackcode]](#stackcode)に示す。

#include <stdbool.h>  
#include <stdio.h>  
#define HEIGHT 5  
  
typedef struct {  
 int data[HEIGHT];  
 int volume;  
} Stack;  
  
/// @brief 構造体を初期化する  
/// @param stack 初期化する構造体  
void init(Stack \*stack) {  
 int i;  
 for (i = 0; i < HEIGHT; i++) {  
 stack->data[i] = 0; // スタックのすべての要素を0にする  
 }  
 stack->volume = 0; // スタックに格納されているデータ数を0にする  
}  
  
/// @brief 構造体にデータをpushする  
/// @param stack pushする構造体  
/// @param number pushするデータ  
/// @return 成功したら0, 失敗したら-1  
int push(Stack \*stack, int number) {  
 if (stack->volume == HEIGHT) { // これ以上スタックできないなら-1を返す  
 return -1;  
 }  
 stack->data[stack->volume] = number; // データを最上位に積み込む  
 stack->volume++; // データの個数を増やす  
 return 0;  
}  
  
/// @brief 構造体からデータをpopする  
/// @param stack popする構造体  
/// @return 成功したらpopしたデータ, 失敗したら-1  
int pop(Stack \*stack) {  
 int num;  
 if (stack->volume == 0) {  
 return -1;  
 }  
 stack->volume--; // 格納されているデータ個数のカウントを減らす  
 num = stack->data[stack->volume]; // 取り出すデータを取り出す  
 stack->data[stack->volume] = 0; // 取り出した場所を初期化する  
 return num;  
}  
  
/// @brief 構造体に格納されているデータを表示する  
/// @param stack 表示する構造体  
void printStack(Stack \*stack) {  
 int i;  
 printf("[ ");  
 for (i = 0; i < stack->volume; i++) { // スタックに格納されている値を  
 printf("%d ", stack->data[i]); // スタックされている順番に1行に表示  
 }  
 printf("]\n");  
}  
  
/// @brief pushできるかどうかを判定する  
/// @param stack 判定する構造体  
/// @return pushできるならtrue, できないならfalse  
bool pushTest(Stack stack) {  
 if (stack.volume == HEIGHT) {  
 return false;  
 } else {  
 return true;  
 }  
}  
  
/// @brief popできるかどうかを判定する  
/// @param stack 判定する構造体  
/// @return popできるならtrue, できないならfalse  
bool popTest(Stack stack) {  
 if (stack.volume == 0) {  
 return false;  
 } else {  
 return true;  
 }  
}  
  
int main(void) {  
 Stack stack;  
 int i, check;  
 init(&stack);  
 printf("MAX STACK NUM:%d\n", HEIGHT);  
 for (i = 10; i < 40; i += 10) {  
 if (pushTest(stack) == true) {  
 printf("pushed %d\n", i);  
 push(&stack, i);  
 } else {  
 printf("error\n");  
 }  
 }  
  
 printf("The data is stacked in order?\n");  
 printStack(&stack);  
  
 printf("The data is retrieved in order?\n");  
 for (i = 0; i < 4; i++) {  
 if (popTest(stack) == true) {  
 printf("popped %d\n", pop(&stack));  
 } else {  
 printf(  
 "What happens when you try to retrieve data when it is empty"  
 "\n");  
 printf("error\n");  
 }  
 }  
  
 for (i = 10; i < 70; i += 10) {  
 if (pushTest(stack) == true) {  
 printf("pushed %d\n", i);  
 push(&stack, i);  
 } else {  
 printf(  
 "What happens when you try to add data when it is full"  
 "\n");  
 printf("error\n");  
 }  
 }  
 return 0;  
}

## ハノイの塔プログラム

ハノイの塔プログラムのソースコードを [[towercode]](#towercode)に示す。

#include <stdbool.h>  
#include <stdio.h>  
#define HEIGHT 5  
#define TOWERS 3  
  
typedef struct {  
 int data[HEIGHT];  
 int volume;  
} Stack;  
  
void init(Stack \*stack) {  
 int i;  
 for (i = 0; i < HEIGHT; i++) {  
 stack->data[i] = 0; // スタックのすべての要素を0にする  
 }  
 stack->volume = 0; // スタックに格納されているデータ数を0にする  
}  
  
int push(Stack \*stack, int number) {  
 if (stack->volume == HEIGHT) { // これ以上スタックできないなら-1を返す  
 return -1;  
 }  
 stack->data[stack->volume] = number; // データを最上位に積み込む  
 stack->volume++; // データの個数を増やす  
 return 0;  
}  
  
int pop(Stack \*stack) {  
 int num;  
 if (stack->volume == 0) {  
 return -1;  
 }  
 stack->volume--; // 格納されているデータ個数のカウントを減らす  
 num = stack->data[stack->volume]; // 取り出すデータを取り出す  
 stack->data[stack->volume] = 0; // 取り出した場所を初期化する  
 return num;  
}  
  
void printStack(Stack \*stack) {  
 int i;  
 printf("[ ");  
 for (i = 0; i < HEIGHT; i++) { // スタックに格納されている値を  
 printf("%d ", stack->data[i]); // スタックされている順番に1行に表示  
 }  
 printf("]\n");  
}  
  
/// @brief 塔の最上位の値を返す関数  
/// @param tower 塔の構造体  
/// @return 塔の最上位の値  
int top(Stack tower) {  
 return tower.data[tower.volume - 1]; /\* スタックの最上位の値を返す \*/  
}  
  
/// @brief 値を移動できるかどうかを判定する関数  
/// @param fromTower 移動元の塔  
/// @param toTower 移動先の塔  
/// @return 可能なら1, 不可能なら0  
int enableStack(Stack fromTower, Stack toTower) {  
 if (fromTower.volume != 0 && toTower.volume != HEIGHT &&  
 top(fromTower) < top(toTower)) {  
 return 1; /\* 移動可能である条件に応じて返り値を返す \*/  
 } else {  
 return 0;  
 }  
}  
  
/// @brief クリアしているかどうかを判定する関数  
/// @param tower 塔の構造体  
/// @param blocks ブロックの数  
/// @return クリアしているなら1, していないなら0  
int checkFinish(Stack tower, int blocks) {  
 int i;  
 int check = blocks;  
 for (i = 0; i < blocks; i++) {  
 if (tower.data[i] == check--) {  
 } else {  
 return 0;  
 }  
 }  
 return 1;  
 // ブロックが初期状態と同じ状態かチェックする  
}  
  
int main(void) {  
 int i;  
 int count = 1;  
 int fromNumber, toNumber;  
 int tempNumber;  
 int blocks;  
 Stack tower[TOWERS];  
  
 printf("Select the number of steps(3, 4, 5):");  
 while (scanf("%d", &blocks) != 1 || blocks < 3 || blocks > 5) {  
 while (getchar() != '\n');  
 printf("error please rewrite\n");  
 printf("Select the number of steps(3, 4, 5):");  
 }  
 /\*3 塔を初期化する\*/  
 for (i = 0; i < TOWERS; i++) {  
 init(&tower[i]);  
 }  
 /\*第１塔に決められた個数をスタックする\*/  
 i = blocks;  
 while (i != 0) {  
 push(&tower[0], i--);  
 }  
  
 /\*塔の初期状態を表示する\*/  
 for (i = 0; i < TOWERS; i++) {  
 printf("%d:", i + 1);  
 printStack(&tower[i]);  
 }  
 while (1) {  
 // 今，何回目の移動であるかを数える．  
 printf("%dth\n", count++);  
 // 移動元と移動先を受け取る  
 while (1) {  
 printf("enter the source and destination towers.[? ?]:");  
 while (scanf("%d%d", &fromNumber, &toNumber) != 2 || toNumber > 3 ||  
 toNumber < 1 || fromNumber > 3 || fromNumber < 1) {  
 while (getchar() != '\n');  
 printf("error please rewrite\n");  
 printf("enter the source and destination towers.[? ?]:");  
 }  
 // 移動元の塔から移動先の塔にデータを移動させる  
 if (enableStack(tower[fromNumber - 1], tower[toNumber - 1]) == 1) {  
 tower[toNumber - 1].data[tower[toNumber - 1].volume] =  
 pop(&tower[fromNumber - 1]);  
 tower[toNumber - 1].volume++;  
 break;  
 } else {  
 printf("Cannot move\n");  
 }  
 }  
 // 現在の塔の状態を表示する  
 for (i = 0; i < TOWERS; i++) {  
 printf("%d:", i + 1);  
 printStack(&tower[i]);  
 }  
 // クリア判定をする  
 for (i = 1; i < TOWERS; i++) {  
 if (checkFinish(tower[i], blocks) == 1) {  
 printf("clear!\n");  
 return 0;  
 }  
 }  
 }  
}