stack

学籍番号:22120
組番号:222
名前: 塚田 勇人

2024 年 6 月 27 日

目 次

1　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。

2　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。

2.1　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。

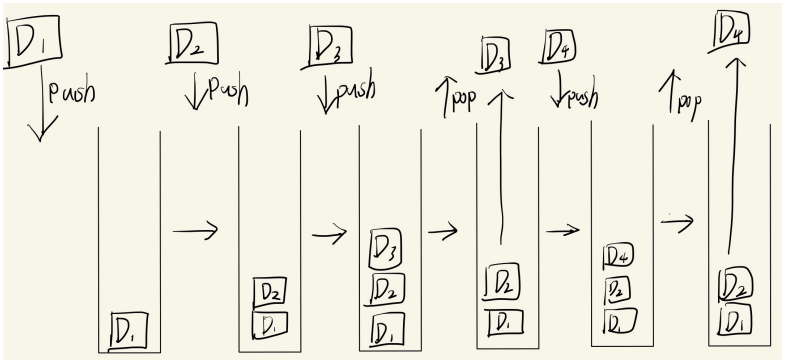

図 1: スタックのイメージ

図 1 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。

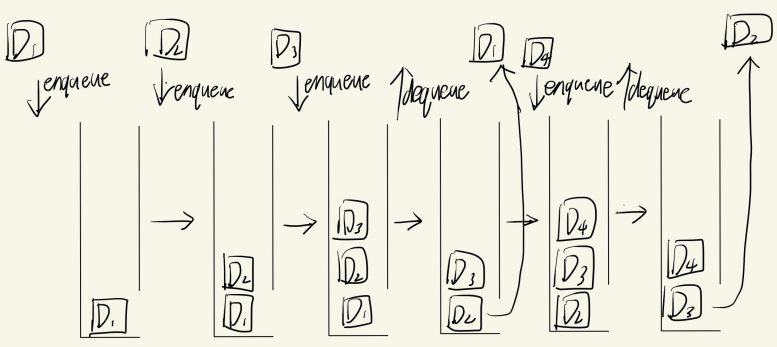

図 2: キューのイメージ

図 2 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。

2.2 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。

3 実験環境

stack の課題のプログラムを実装した環境を表 1 に示します。

表 1: 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

4　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 1 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。

リスト 1: 値と構造体の定義

```
#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。

4.1　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 の関数を使用しています。

表 2: 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。

4.1.1　init 関数

init 関数は表 3 のように定義され、リスト 2 のように実装されています。

表 3: 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

リスト 2: init 関数

```
    void init(Stack *stack) {
      int i;
```

```
3       for (i = 0; i < HEIGHT; i++) {
4           stack->data[i] = 0; // スタックのすべての要素を 0にする
5       }
6       stack->volume = 0; // スタックに格納されているデータ数を 0にする
7   }
```

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。

4.1.2 push 関数

push 関数は表 4 のように定義され、リスト 3 のように実装されています。

表 4: 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

リスト 3: push 関数

```
1   int push(Stack *stack, int number) {
2     if (stack->volume == HEIGHT) { // これ以上スタックできないなら-1を返す
3         return -1;
4     }
5     stack->data[stack->volume] = number; // データを最上位に積み込む
6     stack->volume++; // データの個数を増やす
7     return 0;
8   }
```

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。

4.1.3 pop 関数

pop 関数は表 5 のように定義され、リスト 4 のように実装されています。

表 5: 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

リスト 4: pop 関数

```
1  int pop(Stack *stack) {
2    int num;
3    if (stack->volume == 0) {
4        return -1;
5    }
6    stack->volume--; // 格納されているデータ個数のカウントを減らす
7    num = stack->data[stack->volume]; // 取り出すデータを取り出す
8    stack->data[stack->volume] = 0; // 取り出した場所を初期化する
9    return num;
10 }
```

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。

4.1.4 printStack 関数

printstack 関数は表 6 のように定義され、リスト 5 のように実装されています。

表 6: 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

リスト 5: printstack 関数

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

```
6       }
7       printf("]\n");
8   }
```

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。

4.1.5 pushTest 関数

pushTest 関数は表 7 のように定義され、リスト 6 のように実装されています。

表 7: 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

リスト 6: pushTest 関数

```
1   bool pushTest(Stack stack) {
2     if (stack.volume == HEIGHT) {
3         return false;
4     } else {
5         return true;
6     }
7   }
```

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。

4.1.6 popTest 関数

poptest 関数は表 8 のように定義され、リスト 7 のように実装されています。

表 8: 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

リスト 7: popTest 関数

```
1   bool popTest(Stack stack) {
2     if (stack.volume == 0) {
3         return false;
4     } else {
5         return true;
6     }
7 }
```

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。

4.2 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 の関数に加えて表 9 の関数を使用している。

表 9: 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。

4.2.1 top 関数

top 関数は表 10 のように定義され、リスト 8 のように実装されています。

表 10: 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

リスト 8: top 関数

```
1 int top(Stack tower) {
2   return tower.data[tower.volume - 1];
3 }
```

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。

4.2.2 enableStack 関数

enableStack 関数は表 11 のように定義され、リスト 9 のように実装されています。

表 11: 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

リスト 9: enableStack 関数

```
1  int enableStack(Stack fromTower, Stack toTower) {
2    if (fromTower.volume != 0 && toTower.volume != HEIGHT &&
3        top(fromTower) < top(toTower)) {
4        return 1; /* 移動可能である条件に応じて返り値を返す */
5    } else {
6        return 0;
7    }
8 }
```

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。

4.2.3 checkFinish 関数

checkFinish 関数は表 12 のように定義され、リスト 10 のように実装されています。

表 12: 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

リスト 10: checkFinish 関数

```
1  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。

5 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。

5.1 スタックプログラム

リスト 11 は、スタックプログラムのメイン関数を示しています。

リスト 11: スタックのメイン関数

```
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) {
```

```
21              printf("popped %d\n", pop(&stack));
22          } else {
23              printf(
24                  "What happens when you try to retrieve data when it is empty"
25                  "\n");
26              printf("error\n");
27          }
28      }
29
30      for (i = 10; i < 70; i += 10) {
31          if (pushTest(stack) == true) {
32              printf("pushed %d\n", i);
33              push(&stack, i);
34          } else {
35              printf(
36                  "What happens when you try to add data when it is full"
37                  "\n");
38              printf("error\n");
39          }
40      }
41      return 0;
42  }
```

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。

5.1.1 スタックを初期化する処理

リスト 11 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）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。

5.1.2 スタックに入るデータの最大数を表示する処理

リスト 11 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。

5.1.3　スタックにデータを 10 から 30 プッシュする処理

リスト 11 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）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）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。

5.1.4　スタックの要素を表示する処理

リスト 11 の 15, 16 行目で、printStack 関数（節 4.1.4）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 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 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 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。

5.1.5　スタックからデータを 4 回ポップする処理

リスト 11 の 18 行目から 28 行目で、4 回 pop 関数（節 4.1.3）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）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 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。

5.1.6　スタックにデータを 10 から 60 プッシュする処理

リスト 11 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 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 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。

5.2 ハノイの塔プログラム

リスト 12 は、ハノイの塔プログラムのメイン関数を示しています。

リスト 12: ハノイの塔のメイン関数

```
1  int main(void) {
2    int i;
3    int count = 1;
4    int fromNumber, toNumber;
5    int tempNumber;
6    int blocks;
7    Stack tower[TOWERS];
8
9    printf("Select the number of steps(3, 4, 5):");
10   while (scanf("%d", &blocks) != 1 || blocks < 3 || blocks > 5) {
11       while (getchar() != '\n');
12       printf("error please rewrite\n");
13       printf("Select the number of steps(3, 4, 5):");
14   }
15   /*3 塔を初期化する*/
16   for (i = 0; i < TOWERS; i++) {
17       init(&tower[i]);
18   }
19   /*第１塔に決められた個数をスタックする*/
20   i = blocks;
21   while (i != 0) {
22       push(&tower[0], i--);
23   }
24
25   /*塔の初期状態を表示する*/
26   for (i = 0; i < TOWERS; i++) {
27       printf("%d:", i + 1);
28       printStack(&tower[i]);
29   }
30   while (1) {
31       // 今,何回目の移動であるかを数える.
32       printf("%dth\n", count++);
33       // 移動元と移動先を受け取る
34       while (1) {
35           printf("enter the source and destination towers.[? ?]:");
```

```
36              while (scanf("%d%d", &fromNumber, &toNumber) != 2 || toNumber > 3
                    ||
37                  toNumber < 1 || fromNumber > 3 || fromNumber < 1) {
38                  while (getchar() != '\n');
39                  printf("error please rewrite\n");
40                  printf("enter the source and destination towers.[? ?]:");
41              }
42              // 移動元の塔から移動先の塔にデータを移動させる
43              if (enableStack(tower[fromNumber - 1], tower[toNumber - 1]) == 1) {
44                  tower[toNumber - 1].data[tower[toNumber - 1].volume] =
45                      pop(&tower[fromNumber - 1]);
46                  tower[toNumber - 1].volume++;
47                  break;
48              } else {
49                  printf("Cannot move\n");
50              }
51          }
52          // 現在の塔の状態を表示する
53          for (i = 0; i < TOWERS; i++) {
54              printf("%d:", i + 1);
55              printStack(&tower[i]);
56          }
57          // クリア判定をする
58          for (i = 1; i < TOWERS; i++) {
59              if (checkFinish(tower[i], blocks) == 1) {
60                  printf("clear!\n");
61                  return 0;
62              }
63          }
64      }
65  }
```

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。
   (a) 今何回目の移動であるかを数える。
   (b) ユーザーに移動元と移動先の入力を求める。
   (c) 移動元の塔から移動先の塔にデータを移動させる。
   (d) 塔の状態を表示する。
   (e) クリアしているかどうか判定する。

(f) 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 12 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。

5.2.1 ユーザーにブロックの数の入力を求める処理

リスト 12 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 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。

5.2.2 塔を初期化する処理

リスト 12 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）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。

5.2.3 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 12 の 19 行目から 23 行目で、第 1 塔に節 5.2.1 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。

5.2.4 塔の初期状態を表示する処理

リスト 12 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）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 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。

5.2.5　以下の処理をクリアするまで繰り返す処理

リスト 12 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。

5.2.6　今何回目の移動であるかを数える処理

リスト 12 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 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。

5.2.7　ユーザーに移動元と移動先の入力を求める処理

リスト 12 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 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。

5.2.8　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 12 の 42 行目から 51 行目で、節 5.2.7 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）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）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）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 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。

5.2.9　塔の状態を表示する処理

リスト 12 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 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 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。

5.2.10　クリアしているかどうか判定する処理

リスト 12 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）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 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。

6　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。

6.1　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 13 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。

リスト 13: スタックプログラムの実行結果

```
1 MAX STACK NUM:5
2 pushed 10
3 pushed 20
4 pushed 30
5 The data is stacked in order?
```

```
6 [ 10 20 30 ]
7 The data is retrieved in order?
8 popped 30
9 popped 20
10 popped 10
11 What happens when you try to retrieve data when it is empty
12 error
13 pushed 10
14 pushed 20
15 pushed 30
16 pushed 40
17 pushed 50
18 What happens when you try to add data when it is full
19 error
```

上記の事項について説明をします。

6.1.1 データ順にスタックされているか

リスト 13 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。

6.1.2 データ順に取り出せているか

リスト 13 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。

6.1.3 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 13 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。

6.1.4 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 13 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。

6.2 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 14, 15 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。

リスト 14: ハノイプログラムの実行結果.1

```
1  Select the number of steps(3, 4, 5):6
2  error please rewrite
3  Select the number of steps(3, 4, 5):jkfa
4  error please rewrite
5  Select the number of steps(3, 4, 5):3
6  1:[ 3 2 1 0 0 ]
7  2:[ 0 0 0 0 0 ]
8  3:[ 0 0 0 0 0 ]
9  1th
10 enter the source and destination towers.[? ?]:3 1
11 Cannot move
12 enter the source and destination towers.[? ?]:2 3
13 Cannot move
14 enter the source and destination towers.[? ?]:sdafasd sfdafa
15 error please rewrite
16 enter the source and destination towers.[? ?]:1 3
17 1:[ 3 2 0 0 0 ]
18 2:[ 0 0 0 0 0 ]
19 3:[ 1 0 0 0 0 ]
20 2th
21 enter the source and destination towers.[? ?]:1 2
22 1:[ 3 0 0 0 0 ]
23 2:[ 2 0 0 0 0 ]
24 3:[ 1 0 0 0 0 ]
25 3th
26 enter the source and destination towers.[? ?]:3 2
27 1:[ 3 0 0 0 0 ]
28 2:[ 2 1 0 0 0 ]
29 3:[ 0 0 0 0 0 ]
```

```
4th
enter the source and destination towers.[? ?]:1 3
1:[ 0 0 0 0 0 ]
2:[ 2 1 0 0 0 ]
3:[ 3 0 0 0 0 ]
5th
enter the source and destination towers.[? ?]:2 1
1:[ 1 0 0 0 0 ]
2:[ 2 0 0 0 0 ]
3:[ 3 0 0 0 0 ]
6th
enter the source and destination towers.[? ?]:2 3
1:[ 1 0 0 0 0 ]
2:[ 0 0 0 0 0 ]
3:[ 3 2 0 0 0 ]
7th
enter the source and destination towers.[? ?]:1 3
1:[ 0 0 0 0 0 ]
2:[ 0 0 0 0 0 ]
3:[ 3 2 1 0 0 ]
clear!
```

リスト 15: ハノイプログラムの実行結果.2

```
Select the number of steps(3, 4, 5):4
1:[ 4 3 2 1 0 ]
2:[ 0 0 0 0 0 ]
3:[ 0 0 0 0 0 ]
1th
enter the source and destination towers.[? ?]:1 3
1:[ 4 3 2 0 0 ]
2:[ 0 0 0 0 0 ]
3:[ 1 0 0 0 0 ]
2th
enter the source and destination towers.[? ?]:1 2
1:[ 4 3 0 0 0 ]
2:[ 2 0 0 0 0 ]
3:[ 1 0 0 0 0 ]
3th
 /* 省略 */
```

上記の事項について説明をします。

6.2.1 段数を正しく受け取れているか

リスト 14 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 15 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。

6.2.2　回数を正しく数えられているか

リスト 14 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。

6.2.3　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 14 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。

6.2.4　塔の状態を正しく表示できているか

リスト 14 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。

6.2.5　クリアしているかどうかを判定できているか

リスト 14 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。

6.2.6　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 14 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 14 の 10 行目から 16 行目までの出力を見ると、4.2.2 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 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。

7　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。

7.1　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。

7.2　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。

7.3　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。

8　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。

8.1　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 16 に示す。

リスト 16: スタックプログラム

```
#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) {
```

```
25      if (stack->volume == HEIGHT) { // これ以上スタックできないなら-1を返す
26          return -1;
27      }
28      stack->data[stack->volume] = number; // データを最上位に積み込む
29      stack->volume++; // データの個数を増やす
30      return 0;
31  }
32
33  /// @brief 構造体からデータをpopする
34  /// @param stack popする構造体
35  /// @return 成功したらpopしたデータ，失敗したら-1
36  int pop(Stack *stack) {
37      int num;
38      if (stack->volume == 0) {
39          return -1;
40      }
41      stack->volume--; // 格納されているデータ個数のカウントを減らす
42      num = stack->data[stack->volume]; // 取り出すデータを取り出す
43      stack->data[stack->volume] = 0; // 取り出した場所を初期化する
44      return num;
45  }
46
47  /// @brief 構造体に格納されているデータを表示する
48  /// @param stack 表示する構造体
49  void printStack(Stack *stack) {
50      int i;
51      printf("[ ");
52      for (i = 0; i < stack->volume; i++) { // スタックに格納されている値を
53          printf("%d ", stack->data[i]); // スタックされている順番に1行に表示
54      }
55      printf("]\n");
56  }
57
58  /// @brief pushできるかどうかを判定する
59  /// @param stack 判定する構造体
60  /// @return pushできるならtrue，できないならfalse
61  bool pushTest(Stack stack) {
62      if (stack.volume == HEIGHT) {
63          return false;
64      } else {
65          return true;
66      }
67  }
68
69  /// @brief popできるかどうかを判定する
70  /// @param stack 判定する構造体
71  /// @return popできるならtrue，できないならfalse
72  bool popTest(Stack stack) {
73      if (stack.volume == 0) {
74          return false;
```

```
75      } else {
76          return true;
77      }
78  }
79
80  int main(void) {
81      Stack stack;
82      int i, check;
83      init(&stack);
84      printf("MAX STACK NUM:%d\n", HEIGHT);
85      for (i = 10; i < 40; i += 10) {
86          if (pushTest(stack) == true) {
87              printf("pushed %d\n", i);
88              push(&stack, i);
89          } else {
90              printf("error\n");
91          }
92      }
93
94      printf("The data is stacked in order?\n");
95      printStack(&stack);
96
97      printf("The data is retrieved in order?\n");
98      for (i = 0; i < 4; i++) {
99          if (popTest(stack) == true) {
100             printf("popped %d\n", pop(&stack));
101         } else {
102             printf(
103                 "What happens when you try to retrieve data when it is empty"
104                 "\n");
105             printf("error\n");
106         }
107     }
108
109     for (i = 10; i < 70; i += 10) {
110         if (pushTest(stack) == true) {
111             printf("pushed %d\n", i);
112             push(&stack, i);
113         } else {
114             printf(
115                 "What happens when you try to add data when it is full"
116                 "\n");
117             printf("error\n");
118         }
119     }
120     return 0;
121 }
```

8.2 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 17 に示す。

リスト 17: ハノイの塔プログラム

```
#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");
```

```
46 }
47
48 /// @brief 塔の最上位の値を返す関数
49 /// @param tower 塔の構造体
50 /// @return 塔の最上位の値
51 int top(Stack tower) {
52     return tower.data[tower.volume - 1]; /* スタックの最上位の値を返す */
53 }
54
55 /// @brief 値を移動できるかどうかを判定する関数
56 /// @param fromTower 移動元の塔
57 /// @param toTower 移動先の塔
58 /// @return 可能なら 1，不可能なら 0
59 int enableStack(Stack fromTower, Stack toTower) {
60     if (fromTower.volume != 0 && toTower.volume != HEIGHT &&
61         top(fromTower) < top(toTower)) {
62         return 1; /* 移動可能である条件に応じて返り値を返す */
63     } else {
64         return 0;
65     }
66 }
67
68 /// @brief クリアしているかどうかを判定する関数
69 /// @param tower 塔の構造体
70 /// @param blocks ブロックの数
71 /// @return クリアしているなら 1，していないなら 0
72 int checkFinish(Stack tower, int blocks) {
73     int i;
74     int check = blocks;
75     for (i = 0; i < blocks; i++) {
76         if (tower.data[i] == check--) {
77         } else {
78             return 0;
79         }
80     }
81     return 1;
82     // ブロックが初期状態と同じ状態かチェックする
83 }
84
85 int main(void) {
86     int i;
87     int count = 1;
88     int fromNumber, toNumber;
89     int tempNumber;
90     int blocks;
91     Stack tower[TOWERS];
92
93     printf("Select the number of steps(3, 4, 5):");
94     while (scanf("%d", &blocks) != 1 || blocks < 3 || blocks > 5) {
95         while (getchar() != '\n');
```

```
96          printf("error please rewrite\n");
97          printf("Select the number of steps(3, 4, 5):");
98      }
99      /*3 塔を初期化する*/
100     for (i = 0; i < TOWERS; i++) {
101         init(&tower[i]);
102     }
103     /*第１塔に決められた個数をスタックする*/
104     i = blocks;
105     while (i != 0) {
106         push(&tower[0], i--);
107     }
108
109     /*塔の初期状態を表示する*/
110     for (i = 0; i < TOWERS; i++) {
111         printf("%d:", i + 1);
112         printStack(&tower[i]);
113     }
114     while (1) {
115         // 今,何回目の移動であるかを数える.
116         printf("%dth\n", count++);
117         // 移動元と移動先を受け取る
118         while (1) {
119             printf("enter the source and destination towers.[? ?]:");
120             while (scanf("%d%d", &fromNumber, &toNumber) != 2 || toNumber > 3
                    ||
121                 toNumber < 1 || fromNumber > 3 || fromNumber < 1) {
122                 while (getchar() != '\n');
123                 printf("error please rewrite\n");
124                 printf("enter the source and destination towers.[? ?]:");
125             }
126             // 移動元の塔から移動先の塔にデータを移動させる
127             if (enableStack(tower[fromNumber - 1], tower[toNumber - 1]) == 1) {
128                 tower[toNumber - 1].data[tower[toNumber - 1].volume] =
129                     pop(&tower[fromNumber - 1]);
130                 tower[toNumber - 1].volume++;
131                 break;
132             } else {
133                 printf("Cannot move\n");
134             }
135         }
136         // 現在の塔の状態を表示する
137         for (i = 0; i < TOWERS; i++) {
138             printf("%d:", i + 1);
139             printStack(&tower[i]);
140         }
141         // クリア判定をする
142         for (i = 1; i < TOWERS; i++) {
143             if (checkFinish(tower[i], blocks) == 1) {
144                 printf("clear!\n");
```

```
145                 return 0;
146             }
147         }
148     }
149 }
```