

4-1

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。最後に入れたデータが最初に取り出されます。

■ スタックとは

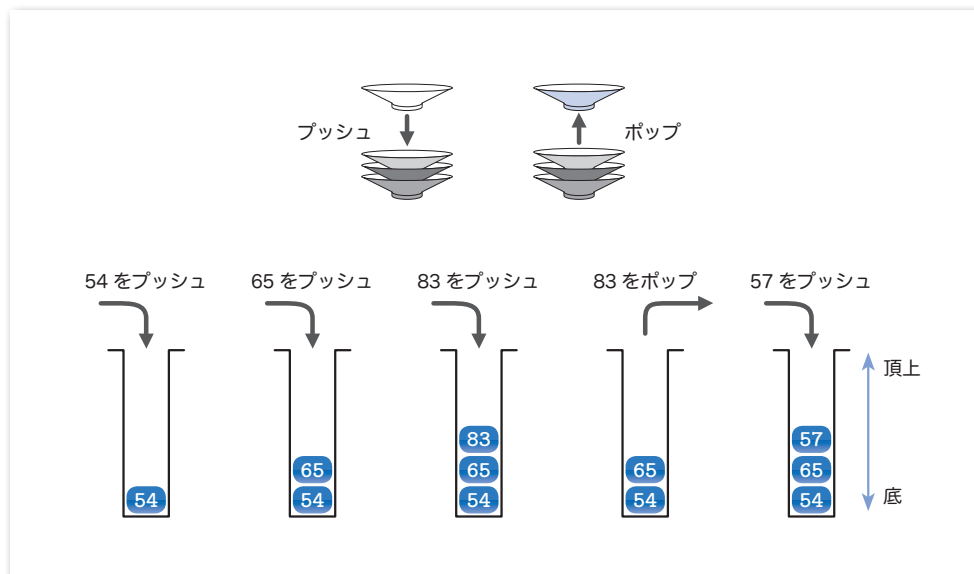
スタック (stack) は、データを一時的に蓄えるためのデータ構造の一つです。データの出し入れは**後入れ先出し** (LIFO / Last In First Out) で行われます。すなわち、最後に入れたデータが最初に取り出されます。

なお、スタックにデータを入れる操作を**プッシュ** (push) と呼び、スタックからデータを取り出す操作を**ポップ** (pop) と呼びます。

スタックにデータをプッシュ／ポップする一例を示したのが、**Fig.4-1** です。テーブルに積み重ねた皿のように、データを入れるのも、取り出すのも、最も“上側”で行います。

なお、プッシュとポップが行われる側を**頂上** (top) と呼び、その反対側を**底** (bottom) と呼びます。

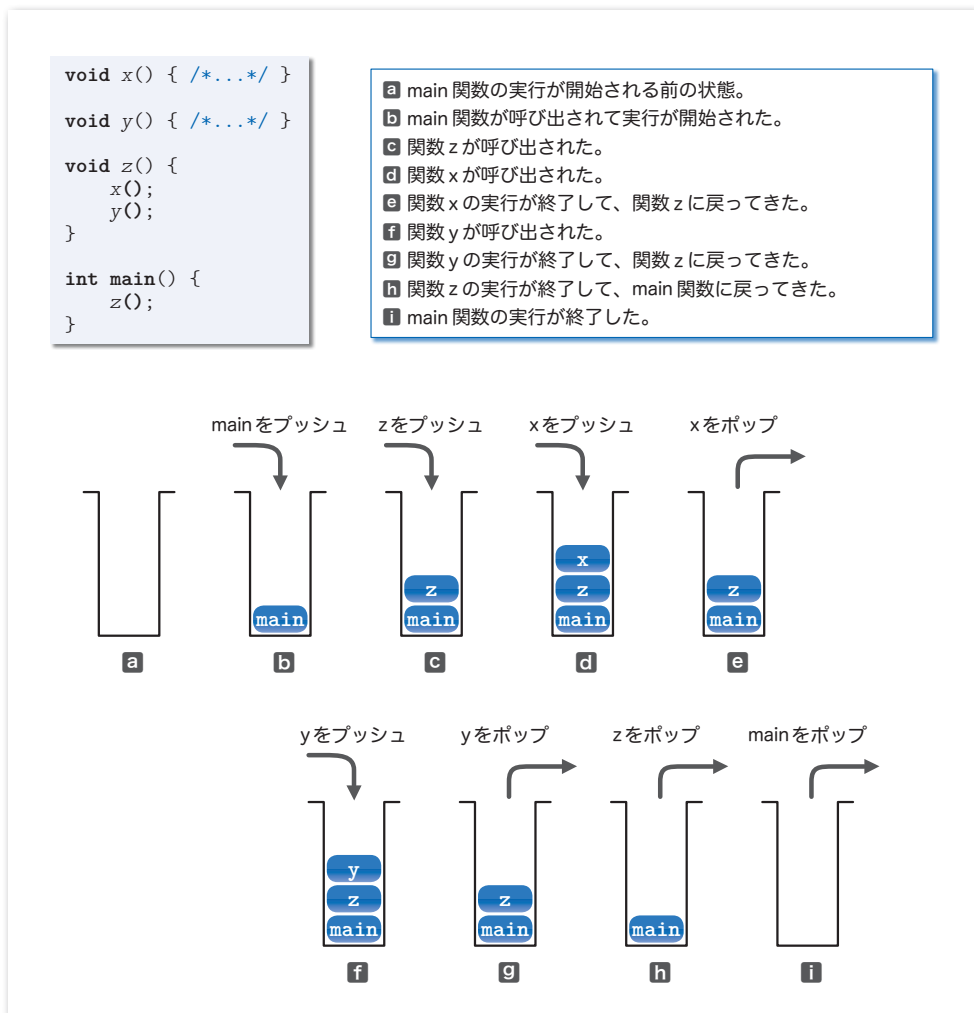
- ▶ stack は、『干し草を積んだ山』『堆積』という意味の語句です。そのため、プッシュすることを“**積む**”ともいいます。



● **Fig.4-1** スタックへのプッシュとポップ

C 言語のプログラムでの関数呼出しや、その実行にあたっては、プログラムの内部でスタックが使われています。そのイメージを簡略的に表したのが **Fig.4-2** です。

この図に示しているプログラムは、**main** を含めて四つの関数から構成されます。



● Fig.4-2 関数呼出しとスタック

まず最初に、**main** 関数の実行が開始されます。それから、**main** 関数が関数 **z** を呼び出します。呼び出された関数 **z** は、関数 **x** と関数 **y** を順次呼び出します。

この図には、呼び出される際に関数がプッシュされ、実行が終了して呼出し元に戻る際に関数がポップされる様子が描かれています。

たとえば、図**d**に着目してみます。この図は、**main** ⇨ **z** ⇨ **x** と関数が呼び出されていて、関数呼出しが階層構造となっていることを示しています。

この状態で関数 **x** の実行が終了したときに、関数 **x** と **z** の二つがポップされて、いきなり **main** 関数に戻るといったことはありません。

- ▶ ここに示したのは、関数呼出しのイメージを理解するための概略図です。実際にはもっと複雑な構造となっています。

■ スタックの実現

スタックを実現するプログラムを作りましょう。ここでは、スタックに格納するデータは、単なる `int` 型の値とします。

List 4-1 の "IntStack.h" がヘッダ部で、**List 4-2** の "IntStack.c" がソース部です。

List 4-1

chap04/IntStack.h

```

/* int型スタックIntStack (ヘッダ部) */
#ifndef ___IntStack
#define ___IntStack

/*--- スタックを実現する構造体 ---*/
typedef struct {
    int max;          /* スタックの容量 */
    int ptr;         /* スタックポインタ */
    int *stk;        /* スタック本体 (の先頭要素へのポインタ) */
} IntStack;

/*--- スタックの初期化 ---*/
int Initialize(IntStack *s, int max);

/*--- スタックにデータをプッシュ ---*/
int Push(IntStack *s, int x);

/*--- スタックからデータをポップ ---*/
int Pop(IntStack *s, int *x);

/*--- スタックからデータをピーク ---*/
int Peek(const IntStack *s, int *x);

/*--- スタックを空にする ---*/
void Clear(IntStack *s);

/*--- スタックの容量 ---*/
int Capacity(const IntStack *s);

/*--- スタック上のデータ数 ---*/
int Size(const IntStack *s);

/*--- スタックは空か ---*/
int IsEmpty(const IntStack *s);

/*--- スタックは満杯か ---*/
int IsFull(const IntStack *s);

/*--- スタックからの探索 ---*/
int Search(const IntStack *s, int x);

/*--- 全データの表示 ---*/
void Print(const IntStack *s);

/*--- スタックの後始末 ---*/
void Terminate(IntStack *s);

#endif

```

■ スタック構造体：IntStack

スタックを管理するための構造体です。三つのメンバから構成されます。

■ スタック本体用の配列：stk

プッシュされたデータを格納するスタック本体用の配列です。**Fig.4-3** に示すように、添字が0の要素をスタックの底とします。そのため、最初にプッシュされたデータの格納先は `stk[0]` となります。

- ▶ メンバ `stk` は、配列の先頭要素を指すポインタであり、配列本体ではありません。配列本体は、関数 `Initialize` で生成します。

■ スタックの容量：max

スタックの容量（スタックに積める最大のデータ数）を表すメンバです。この値は、配列 `stk` の要素数と一致します（図の例では、`max` の値は8です）。

■ スタックポインタ：ptr

スタックに積まれているデータの個数を表

すメンバです。この値は、**スタックポインタ** (*stack pointer*) と呼ばれます。もちろん、スタックが空であれば `ptr` の値は0となり、満杯であれば `max` と同じ値になります。

最初にプッシュされた“底”のデータが `stk[0]` に格納されているのに対し、最後にプッシュされた“頂上”のデータは `stk[ptr - 1]` に格納されていることとなります。

```

/* int型スタックIntStack (ソース部) */

#include <stdio.h>
#include <stdlib.h>
#include "IntStack.h"

/*--- スタックの初期化 ---*/
int Initialize(IntStack *s, int max)
{
    s->ptr = 0;
    if ((s->stk = calloc(max, sizeof(int))) == NULL) {
        s->max = 0;                               /* 配列の確保に失敗 */
        return -1;
    }
    s->max = max;
    return 0;
}

```

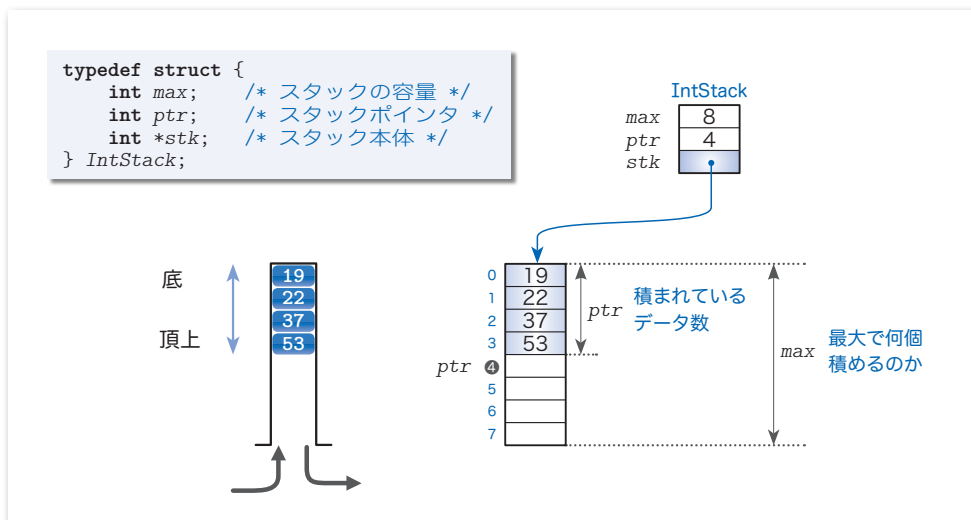
■ 初期化：Initialize

関数 `Initialize` は、スタック本体用の配列領域を確保などの準備処理を行うための関数です。

生成時のスタックは空（データが1個も積まれていない状態）ですから、スタックポインタ `ptr` の値を `0` にし、要素数が `max` である配列 `stk` の本体を確保します。そして、仮引数 `max` に受け取った値をスタックの容量を表すデータメンバ `max` にコピーします。

そのため、スタック本体の個々の要素は、底から `stk[0]`、`stk[1]`、 \dots 、`stk[max - 1]` となります。

- ▶ 配列本体の確保に失敗した場合は、`max` の値を `0` にしています。存在しない配列 `stk` の本体領域に対して、他の関数が不正にアクセスするのを防止するためです。



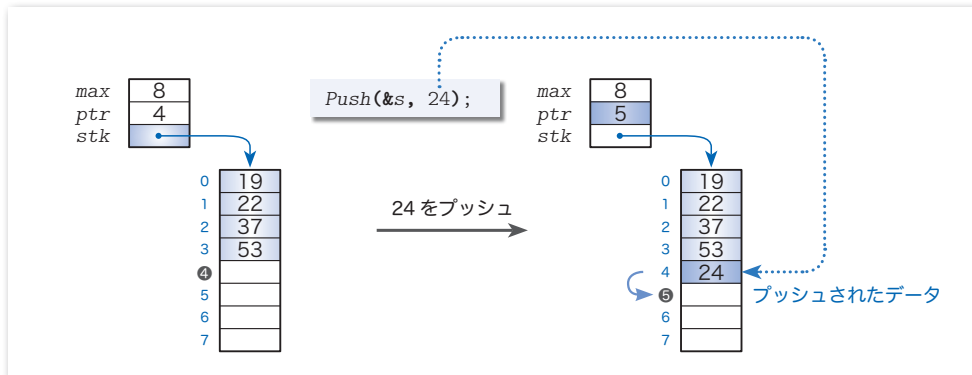
● Fig.4-3 スタックの実現例

■ プッシュ : Push

関数 `Push` は、スタックにデータをプッシュする関数です。

スタックが満杯でプッシュできない場合は `-1` を返却します。満杯でない場合は、受け取ったデータ `x` を、配列の要素 `stk[ptr]` に格納するとともに、スタックポインタをインクリメントした上で、プッシュに成功したことを表す `0` を返却します。

プッシュ操作を行う一例を、**Fig.4-4** に示します。



● **Fig.4-4** スタックへのプッシュ

- ▶ "`IntStack.c`" で提供されている関数のみを利用してスタック操作を行う限り、スタックポインタ `ptr` の値は、必ず `0` 以上かつ `max` 以下となります。したがって、スタックが満杯であるかどうかの判断は、関係演算子 `>=` ではなく、等価演算子 `==` を利用して、以下のように行えます。

```
if (s->ptr == s->max)
```

しかし、プログラムミスなどに起因して `ptr` の値が不正に書き換えられた場合は、`ptr` の値が `max` を超える可能性があります。

本プログラムのように不等号を付けて判断すれば、スタック本体の配列に対する上限や下限を超えたアクセスを防げます。このような些細な工夫で、プログラムの頑健性が向上します。

■ ポップ : Pop

関数 `Pop` は、スタックの頂上からデータをポップする関数です。ポップに成功した場合は `0` を返却しますが、スタックが空でポップできない場合は `-1` を返却します。

ポップ操作を行う一例を、**Fig.4-5** に示しています。まずスタックポインタ `ptr` の値をデクリメントして、それから `stk[ptr]` に格納されている値を、ポインタ `x` が指す変数に格納します。

- ▶ スタックが空であるかどうかを、`ptr == 0` ではなく、不等号を用いた `ptr <= 0` によって判断している理由は、関数 `Push` と同じです。

■ ピーク : Peek

関数 `Peek` は、スタックの頂上のデータ（次にポップを行ったときに取り出されることになるデータ）を“覗き見”するための関数です。ピークに成功した場合は `0` を返却しますが、スタックが空のときは `-1` を返却します。

List 4-2 [B]

chap04/IntStack.c

```

/*--- スタックにデータをプッシュ ---*/
int Push(IntStack *s, int x)
{
    if (s->ptr >= s->max)                /* スタックは満杯 */
        return -1;
    s->stk[s->ptr++] = x;
    return 0;
}

/*--- スタックからデータをポップ ---*/
int Pop(IntStack *s, int *x)
{
    if (s->ptr <= 0)                      /* スタックは空 */
        return -1;
    *x = s->stk[--s->ptr];
    return 0;
}

/*--- スタックからデータをピーク ---*/
int Peek(const IntStack *s, int *x)
{
    if (s->ptr <= 0)                      /* スタックは空 */
        return -1;
    *x = s->stk[s->ptr - 1];
    return 0;
}

/*--- スタックを空にする ---*/
void Clear(IntStack *s)
{
    s->ptr = 0;
}

```

4-1

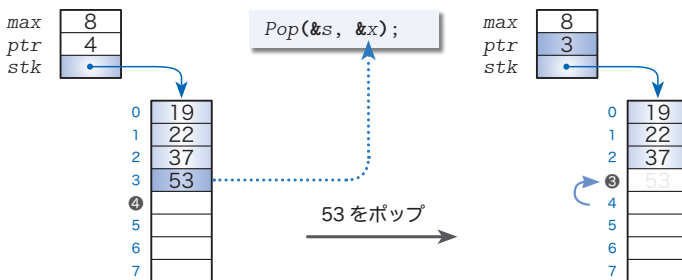
スタック

スタックが空でなければ、頂上の要素 `stk[ptr - 1]` の値を、ポインタ `x` が指す変数に格納します。なお、データの出し入れはしませんので、スタックポインタは変化しません。

■ 全要素の削除：Clear

関数 `Clear` は、スタックに積まれている全データを削除する関数です。

- ▶ スタックに対するプッシュやポップなどのすべての操作は、スタックポインタに基づいて行われるため、スタック本体用の配列要素の値を変更する必要はありません。全要素の削除は、スタックポインタ `ptr` の値を `0` にするだけで完了します。



● Fig.4-5 スタックからのポップ

List 4-2 [C]

chap04/IntStack.c

```

/*--- スタックの容量 ---*/
int Capacity(const IntStack *s)
{
    return s->max;
}

/*--- スタックに積まれているデータ数 ---*/
int Size(const IntStack *s)
{
    return s->ptr;
}

/*--- スタックは空か ---*/
int IsEmpty(const IntStack *s)
{
    return s->ptr <= 0;
}

/*--- スタックは満杯か ---*/
int IsFull(const IntStack *s)
{
    return s->ptr >= s->max;
}

/*--- スタックからの探索 ---*/
int Search(const IntStack *s, int x)
{
    int i;
    for (i = s->ptr - 1; i >= 0; i--) /* 頂上→底に線形探索 */
        if (s->stk[i] == x)
            return i; /* 探索成功 */
    return -1; /* 探索失敗 */
}

/*--- 全データの表示 ---*/
void Print(const IntStack *s)
{
    int i;
    for (i = 0; i < s->ptr; i++) /* 底→頂上に走査 */
        printf("%d ", s->stk[i]);
    putchar('\n');
}

/*--- スタックの後始末 ---*/
void Terminate(IntStack *s)
{
    if (s->stk != NULL)
        free(s->stk);
    s->max = s->ptr = 0;
}

```

■ 容量を調べる：Capacity

関数Capacityは、スタックの容量を返す関数です。メンバmaxの値をそのまま返します。

■ データ数を調べる：Size

関数Sizeは、スタックに積まれているデータ数を返す関数です。メンバptrの値をそのまま返します。

■ 空であるかを判定する：IsEmpty

関数 *IsEmpty* は、スタックが空（データが一つも積まれていない状態）であるかどうかを判定する関数です。空であれば1を、そうでなければ0を返します。

■ 満杯であるかを判定する：IsFull

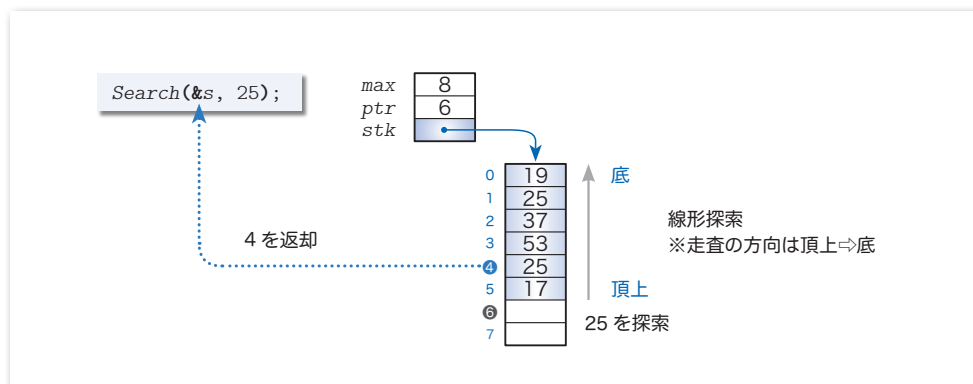
関数 *IsFull* は、スタックが満杯（それ以上データをプッシュできない状態）であるかどうかを判定する関数です。満杯であれば1を、そうでなければ0を返します。

■ 探索：Search

関数 *Search* は、任意の値のデータがスタックに積まっているかどうか、積まれていれば配列内のどこに積まっているのかを調べる関数です。

この関数によって、スタックからの探索を行う例を、**Fig.4-6** に示しています。探索は、**頂上側から底側**への線形探索によって行います。すなわち、配列の添字の**大きいほうから小さいほう**へと走査します。探索に成功した場合は、見つけた要素の添字を返し、失敗した場合は-1を返します。

- ▶ 図に示しているスタックには、添字1の要素と4の要素の2箇所に25があります。このスタックから25を探索すると、**頂上側**の25の添字である4を返します。頂上側から走査するのは、《先にポップされることになるデータ》を優先的に見つけるためです。



● **Fig.4-6** スタックからの探索

■ 全データの表示：Print

関数 *Print* は、スタック本体用の配列内の全データを表示する関数です。スタックに積まれている *ptr* 個すべてのデータを底から頂上へと順に表示します。

■ 後始末：Terminate

関数 *Terminate* は、後始末用の関数です。関数 *Initialize* で確保していたスタック本体用の配列を解放・破棄し、容量 *max* とスタックポインタ *ptr* の値を 0 にします。

■ スタックを利用するプログラム例

スタックを利用するプログラムを作りましょう。**List 4-3** にプログラム例を示します。

- ▶ 本プログラムのコンパイルには、"IntStack.h" と "IntStack.c" が必要です。

List 4-3

chap04/IntStackTest.c

```

/* int型スタックIntStackの利用例 */
#include <stdio.h>
#include "IntStack.h"

int main(void)
{
    IntStack s;
    if (Initialize(&s, 64) == -1) {
        puts("スタックの生成に失敗しました。");
        return 1;
    }
    while (1) {
        int menu, x;

        printf("現在のデータ数 : %d / %d\n", Size(&s), Capacity(&s));
        printf("(1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了 : ");
        scanf("%d", &menu);
        if (menu == 0) break;
        switch (menu) {
            case 1: /*--- プッシュ ---*/
                printf("データ : ");
                scanf("%d", &x);
                if (Push(&s, x) == -1)
                    puts("\aエラー : プッシュに失敗しました。");
                break;
            case 2: /*--- ポップ ---*/
                if (Pop(&s, &x) == -1)
                    puts("\aエラー : ポップに失敗しました。");
                else
                    printf("ポップしたデータは%dです。 \n", x);
                break;
            case 3: /*--- ピーク ---*/
                if (Peek(&s, &x) == -1)
                    puts("\aエラー : ピークに失敗しました。");
                else
                    printf("ピークしたデータは%dです。 \n", x);
                break;
            case 4: /*--- 表示 ---*/
                Print(&s);
                break;
        }
    }
    Terminate(&s);
    return 0;
}

```

本プログラムは、容量が64であるスタックを生成し、プッシュ、ポップ、ピーク、スタック内データの表示を対話的に行います。

実行例

現在のデータ数：0 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：1 データ：1	1をプッシュ。
現在のデータ数：1 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：1 データ：2	2をプッシュ。
現在のデータ数：2 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：1 データ：3	3をプッシュ。
現在のデータ数：3 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：1 データ：4	4をプッシュ。
現在のデータ数：4 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：3 ピークしたデータは4です。	4をピーク。
現在のデータ数：4 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：4 1 2 3 4	スタックの中身を表示。
現在のデータ数：4 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：2 ポップしたデータは4です。	4をポップ。
現在のデータ数：3 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：2 ポップしたデータは3です。	3をポップ。
現在のデータ数：2 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：4 1 2	スタックの中身を表示。
現在のデータ数：2 / 64 (1)プッシュ (2)ポップ (3)ピーク (4)表示 (0)終了：0	

4-1

スタック

演習 4-1

List 4-3 で利用しているのは、"IntStack.c" で提供される関数のうちの一部である。すべての関数を利用するプログラムを作成せよ。

演習 4-2

一つの配列を共有して二つのスタックを実現するスタックのプログラムを作成せよ。スタックに格納するデータは `int` 型の値とし、図のように配列の先頭側と末尾側の両側を利用すること。

