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。データとして格納するのは、単一の **int** 型の値とし、容量（スタックに積める最大のデータ数）を生成時に決定する**固定長のスタック**とします。

List 4-1 の "IntStack.h" がヘッダ部で、右ページ **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あり、3個のメンバで構成されます。

▪ 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。

右ページの**Fig.4-3**に示すように、添字**0**の要素をスタックの**底**とします。

▶ *stk*は、配列の本体ではなく、配列の先頭要素を指すポインタです。配列本体の生成は、関数 *Initialize* で行います

▪ スタックの容量：max

スタックの容量を表す **int** 型のメンバです。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あり、その値は**スタックポインタ**（stack pointer）と呼ばれます。

この *ptr* の値は、スタックが空であれば **0** で、満杯であれば *max* です。

▶ 最初にプッシュされた**底**が *stk[0]* で、最後にプッシュされた**頂上**が *stk[ptr - 1]* です。

List 4-2 [A] chap04/IntStack.c

```
// 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;                                          ← 1
    if ((s->stk = calloc(max, sizeof(int))) == NULL) {
        s->max = 0;                                      ← 2
        return -1;                    // 配列の生成に失敗
    }
    s->max = max;                                        ← 3
    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するなどの準備処理を行います。

▶ 第1引数 *s* は、処理の対象となる**スタック構造体オブジェクトへのポインタ**です（これ以降のほとんどの関数も同様です）。

1 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 にします。

2 要素数 *max* の配列 *stk* の本体を生成します。スタックの個々の要素をアクセスする添字式は、底から *stk*[0], *stk*[1], …, *stk*[*max* - 1] となります。

3 仮引数 *max* に受け取った値を、スタックの容量用データメンバ *max* にコピーします。

▶ 配列本体の確保に失敗した場合は、**2** で *max* の値を 0 にします。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。

```
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
} IntStack;
```

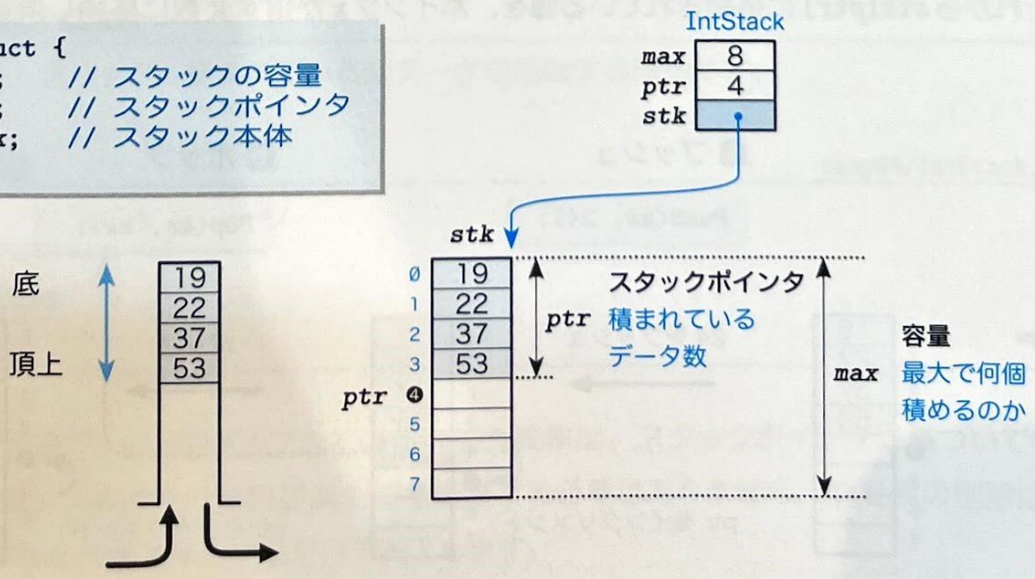


Fig.4-3 スタックの実現例

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。

▶ プッシュに成功すると 0 を返却し、スタックが満杯でプッシュできなければ -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;
}
```

Fig.4-4 a に示すのが、プッシュ操作の一例です。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 *ptr* をインクリメントします。

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。

▶ ポップに成功すると 0 を返却し、スタックが空でポップできなければ -1 を返却します。

List 4-2 [C] chap04/IntStack.c

```
/*--- 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;
}
```

図 b に示すのが、ポップ操作の一例です。まずスタックポインタ *ptr* の値をデクリメントし、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。

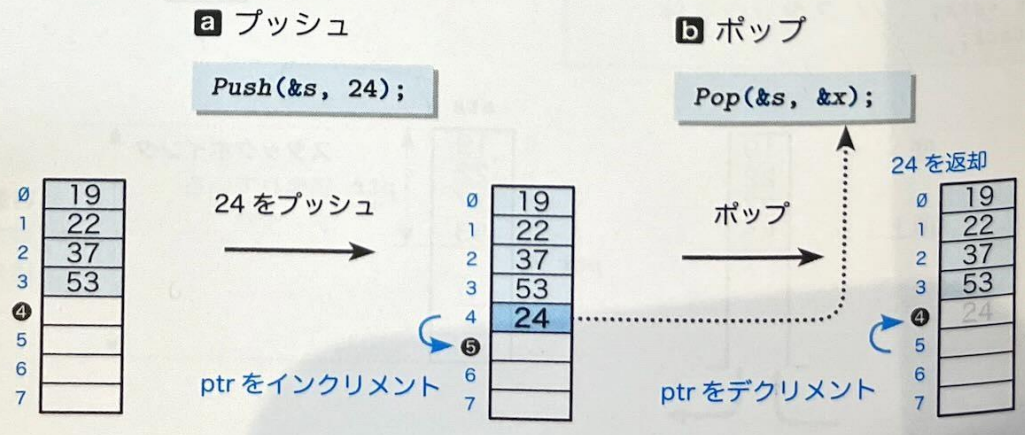


Fig.4-4 スタックへのプッシュとポップ

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データ）を“覗き見”する関数です。

▶ ピークに成功すると 0 を返却し、スタックが空でピークできなければ -1 を返却します。

List 4-2 [D] chap04/IntStack.c

```
/*--- 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、データの出し入れがないため、スタックポインタは変化しません。

▶ 関数 *Push* と、*Pop* と *Peek* では、スタックが満杯であるか、あるいは、スタックが空であるかどうかを関数冒頭の if 文で判定しています。その判定で使っているのが、>= 演算子と <= 演算子です。

スタックが満杯である／空であるかどうかの判定は、等価演算子 == あるいは != を利用して、次のように行えるはずです。

```
if (s->ptr == s->max)       // スタックは満杯か？
if (s->ptr == 0)            // スタックは空か？
```

というのも、"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 *s->max* 以下になるからです。

とはいえ、プログラムミスなどに起因して *ptr* の値が不正に書きかえられた場合、0 より小さくなったり、*s->max* を超えたりす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定すれば、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、**プログラムの頑健さが向上します**

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。

List 4-2 [E] chap04/IntStack.c

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

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。

List 4-2 [F] 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)
{
    for (int 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)
{
    for (int 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);          // 配列を破棄
    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。スタックポインタ *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1個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（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、値 *x* のデータがスタックに積まれているかどうか、積まれていれば配列内のどこに積まれているのかを調べる関数です。

Fig.4-5 に示すのが、探索の一例です。この図に示すように、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の要素から小さいほうの要素へと走査します。

頂上側から走査するのは、《先にポップされるデータ》を優先的に見つけるためです。

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。

▶ 図に示すスタックには、値が 25 の要素が2個あります（添字 1 の要素と 4 の要素です）。ここから 25 を探索すると、**頂上側の 25 の添字 4** を返します。

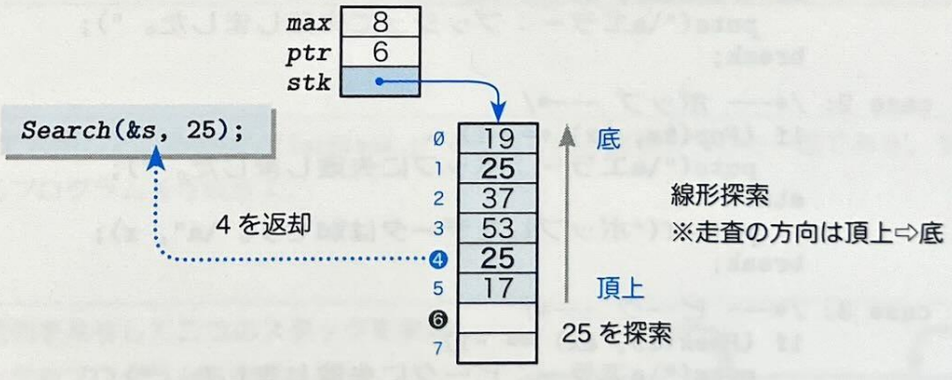


Fig.4-5　スタックからの探索

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* で生成していたスタック本体用の配列を破棄して、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** に示すのが、そのプログラム例です。

▶ 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;
}
```

容量 64 のスタックを生成した上で、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↵
```

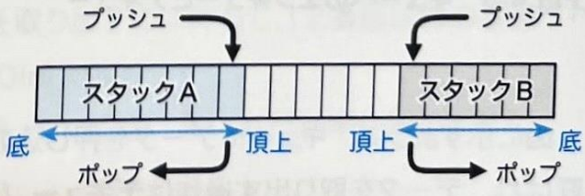
演習 4–1

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。



Column 4-1　関数名について

前章のハッシュ、本章のスタック、キュー、第8章の線形リストなどのプログラムには、同じ名前の関数があります（たとえば、*Initialize* など）。

そのため、それらのプログラムを併用するプログラムは作成不能です。もし併用の必要があれば、*Stack_Initialize*、*Queue_Initialize* など、異なる名前に置きかえるとよいでしょう。