CプログラミングI

Swap 関数を作る

Stack データ構造のための準備

整数変数 x と y の値を取り替える関数 swap を作る

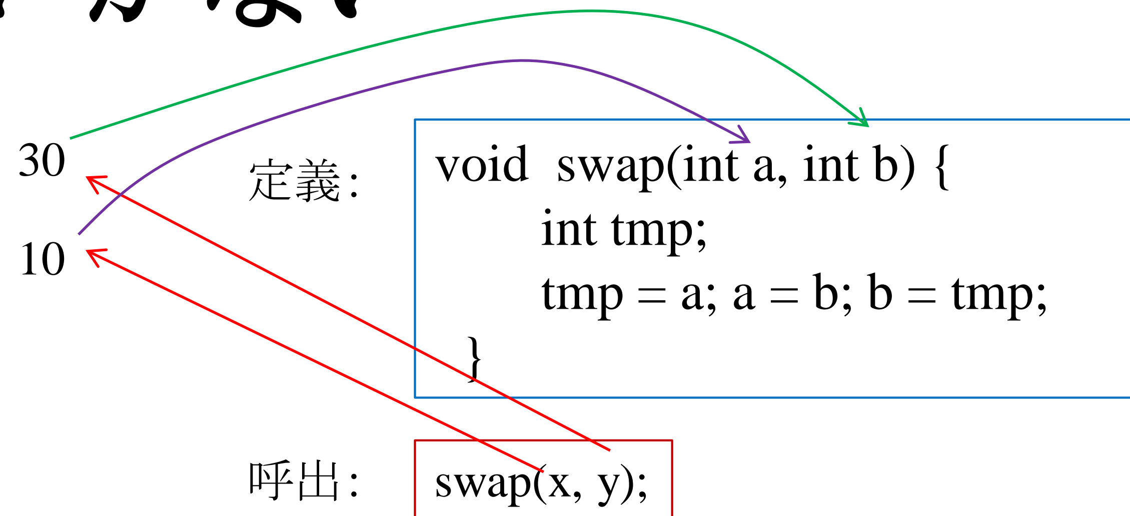
最初の試み: swap-01.c

```
#include <stdio.h>

void  swap(int a, int b) {
      int tmp;
      tmp = a; a = b; b = tmp;
}

int main(void) {
    int x=10, y=30;
    printf("Before: x = %d, y = %d¥n", x, y);  // 10, 30と表示される
    swap(x, y);
    printf("After: x = %d, y = %d¥n", x, y);   // 30, 10と表示される?
    return 0;
 }
```

これではうまくいかない



原因は、(swap)関数に渡される値にある。

a, bそれぞれに xとyの値が渡される。swap関数の中では確かに変数 aとbの値はswapされるが、swap関数を呼び出した(main)関数内の変数xとyの値が入れ替わるわけではない。

そこで、xとyのアドレスを渡す

xとyのアドレス値を記憶する変数を px, py とすれば、

px = xのアドレス

なので、*px は xの値となる。

pyとyについても同様。

そう考えると，次のように関数を定義することが考えられる:

```
void  swap(int *a, int *b) {
        int tmp;
        tmp = *a;   // *a(=x)の値をtmpで記憶
        *a = *b;        // *a (=x)の値を *b(=y)で書き換える
        *b = tmp;     // *b (=y)の値を tmp の値で書き換える
  }
```

試してみよう

swap-01.cのswap関数を書き換える: swap-02.c

```
#include <stdio.h>
```

定義:

```
void  swap(int *a, int *b) {
      int tmp;
      tmp = *a;   // *a(=x)の値をtmpで記憶
      *a = *b;          // *a (=x)の値を *b(=y)で書き換える
      *b = tmp;         // *b (=y)の値を tmp の値で書き換える
}
```

呼出:

```
int main(void) {
     int x=10, y=30;
     printf("Before: x = %d, y = %d¥n", x, y);  // 10, 30と表示される
     swap(x,y);
     printf("After: x = %d, y = %d¥n", x, y);   // 30, 10と表示される?
     return 0;
 }
```

その結果は...

コンパイルすると、次のようなエラーがでる(はず) :

エラー E2342 swapTemplate2.c 13: パラメータ 'a' は int * 型として定義されているので int は渡せない(関数 main)

エラー E2342 swapTemplate2.c 13: パラメータ 'b' は int * 型として定義されているので int は渡せない(関数 main)

*** 2 errors in Compile ***

これはswap関数が間違っているからではない。

実は、swap関数の呼び出し

エラーが起こるのは、swap関数の呼び出し

swap(x,y);

が間違いなのである。

このままだと何回やっても xとyの値が swap 関数に渡される。
しかしswap関数に渡してほしいのは xとyのアドレス（ポインタ）である。
そこで、

swap(&x, &y);

と直す必要がある（理由は分かるだろうか？）

さあ書きなおして再度試してみよう。

再チャレンジ

定義:

```
#include <stdio.h>

void  swap(int *a, int *b) {
      int tmp;
      tmp = *a;   // *a(=x)の値をtmpで記憶
      *a = *b;       // *a (=x)の値を *b(=y)で書き換える
      *b = tmp;     // *b (=y)の値を tmp の値で書き換える
}
```

呼出:

```
int main(void) {
     int x=10, y=30;
     printf("Before: x = %d, y = %d¥n", x, y);  // 10, 30と表示される
     swap(&x, &y);
     printf("After: x = %d, y = %d¥n", x, y);   // 30, 10と表示される?
     return 0;
 }
```

結果:

Before: x = 10, y = 30

After: x = 30, y = 10

参考：配列に対しても…

注意1:　浮動小数点数をswapするには、intをすべて float とする
注意2:　文字列(文字列の配列)にはこのままでは使えない

```
#include <stdio.h>
```

定義は同じ:

```
void  swap(int *a, int *b) {
      int tmp;
      tmp = *a;   // *a(=x)の値をtmpで記憶
      *a = *b;       // *a (=x)の値を *b(=y)で書き換える
      *b = tmp;     // *b (=y)の値を tmp の値で書き換える
}
```

呼出:

```
int main(void) {
    int ar[ ] = {10, 30 };  // 配列
    printf("Before: ar[0] = %d, ar[1] = %d¥n", ar[0], ar[1]);  // 10, 30と表示される
    swap(&ar[0], &ar[1]);
    printf("After: ar[0] = %d, ar[1] = %d¥n", ar[0], ar[1]);   // 30, 10と表示される?
    return 0;
}
```

配列に似たデータ構造：スタックとキュー

スタック(stack)とは

スタックは、「データの挿入、およびデータの取り出しが制限された配列」とみなせる

比較:配列 array を例にすれば、

array[10] とすると 添字番号10番目の要素が取り出せ

array[13]=100; とすれば添字番号113番目の要素を書き換えられる(≒データの記憶が可能)

このように、配列では、任意の場所のデータを取り出せ、

任意の場所にデータを記憶できる

それに対しスタックは、特定の位置(多くは「最後」)にしかデータを追加、もしくは取り出せないようになっている

スタックの操作

- データの追加のための関数を push
  最後の要素として追加される
- データの取り出しのための関数を pop
  最後の要素が取り出され、スタックから取り除かれる

見方を変えると、一番最後に入ってきたものが最初に取り出されるため、
Last In First Out (LIFO) ともいう

配列を使ったスタックの実現

必要な物は:

(1) データ記憶用の配列

(2)　スタックに記憶されている要素数(= 最後の要素が入っている場所)

(3)　そのスタックに要素を付け加える関数 push

(4)　そのスタックから要素を取り出す関数 pop

スタックの実現

(1) データ記憶用の配列
(2) スタックに記憶されている要素数
(3) そのスタックに要素を付け加える関数 push
(4) そのスタックから要素を取り出す関数 pop

(1)は（大域変数として）

```
int stack[NUM];  // NUMは #define NUM 1000 というように宣言しておく
```

(2)はこのための（大域）変数 stack_top を宣言しておく

```
int stack_top=0;          // 初めは空っぽなので、0 を初期値としておく
```

(3)の関数pushはスタックと、その要素数を記憶している変数、そのスタックにいれる数を引数とする. 値は返さないので型はvoidである

```
void push(int st[ ], int *top, int item);
// top の値を書き換えないといけないので、ポインタでないといけない*要解説*
```

4)の関数 pop はスタックと、その要素数を記憶している変数とを引数とし、そのスタックの最後の要素を返す。そして、要素数が1減り、元のスタックにおいて最後に記憶された要素を返す

```
void pop(int st[ ], int *top);
```

課題

- 課題1. push 関数の定義は以下：　関数本体は1行で書ける。チャレンジしよう

```
void push(int st[ ], int *top, int item) {
        int x = *top;
        st[x] = item;
        *top = x+1;
}
```

- 課題2. pop 関数の定義は以下：　関数本体は1行で書ける。チャレンジしよう

```
int pop(int st[ ], int *top) {
        *top = *top - 1;
        return st[ *top ];
}
```

課題3. 次のプログラムを完成させ、走らせて期待したような値が表示されるか確かめよう

```
#include <stdio.h>
#define NUM 100
int stack[NUM];
int stack_top = 0;
void push(int st[ ], int *top, int item) {
    // ここに push の定義を書く
}
int pop(int st[ ], int *top) {
    // ここに pop の定義を書く
}
void printStack(int st[], int top) {
    printf("Current state of the stack :");
    for(top-- ; top >= 0;  top--)
        printf("%3d", st[top]);printf("¥n");
}
```

```
int main(void) {
        push(stack, &stack_top, 10);
        push(stack, &stack_top, 20);
        push(stack, &stack_top, 30);
        printStack(stack, stack_top);
        printf("popped --- %d¥n", pop(stack, &stack_top));
        push(stack, &stack_top, 40);
        push(stack, &stack_top, 50);
        printStack(stack, stack_top);
        while (stack_top > 0) {
          printf("popped --- %d¥n", pop(stack, &stack_top));
        }
        return 0;
}
```

期待した実行結果：

```
Current state of the stack :   30   20   10
popped --- 30
Current state of the stack :   50   40   20   10
popped --- 50
popped --- 40
popped --- 20
popped --- 10
```

なぜこうなるはずなのか、説明せよ

課題4. スタックの応用として、逆ポーランド記法がある。逆ポーランド記法とは、普通の数式が

1 + 2 * 3 - 4

というように、演算子(+ - * /)を、演算される要素の間に書く(これを中置記法という)のに対し、

1 2 3 * + 4 -

というように、演算子を演算される要素の後ろに置く(後置記法)。

一見難しそうに見えるかもしれないが、

(1) 日本語として読みやすい(ただし、適当に助詞を入れる必要がある)、

(2) カッコが不要になる

という特徴がある。

たとえば、「1 2 3 * + 4 -」は

1 に 2 と 3 を *した (掛けた)ものを +して(足して) 4 を -する(引く)

と読める。

ちなみに、前置記法であるポーランド記法では、演算子が先頭に書かれる:

- + 1 * 2 3 4

なお、その計算機では = を入力したら、そ これを基礎知識として、数値や演算子を一つ一つ読み込み、それらが逆ポーランド記法として書かれたものとして計算する「電卓」を作れ。

なお、＝が入力されたら、そこまでの結果を表示するものとする

(porlandTemplate.c参照)

課題5. キューについて調べ、配列を用いてキューを実現するためには、どのようにすればよいか、スタックの分析を参考に考えてみよう。