

Раздел «Язык Си» . CoffeStack :

- **Стек**
 - Реализация стека на основе массива фиксированной длины
 - Стек на основе динамического массива
 - Стек с выделением всей необходимой памяти единым куском
 - Задача 1.
 - Задача 2.
 - Задача 3.
 - Перевод из инфиксной записи в постфиксную
 - Вычисление инфиксной записи с помощью стека
 - Ответы на контрольные вопросы
 - Стек, который не переполнится, но теряет значения
 - Закраска объектов картинки
 - Задачи из сборника http://acm.mipt.ru/twiki/pub/Cintro/LongTask_Retro/odarka97.pdf:

Стек

Стек (stack) – абстрактный тип данных, представляющий последовательность элементов, организованных по принципу LIFO ("last in first out" – последним пришел, первым вышел).



Примеры стека в окружающей реальности:

- детская пирамидка из кружков на палочке;
- башенка из камней;
- стопка тарелок;
- электричка в час пик (последним вошел в вагон – первым вышел из вагона, даже если не хотел).

Реализация стека основывается на двух методах (функциях):

- **push(x)** – положить элемент x на верх стека.
- **x = pop()** – удалить элемент с верха стека, и вернуть его значение (запомним в x).

Для удобства работы можно объявить еще функции, например, **print** (распечатать содержимое стека, бесценно для отладки), **is_empty** (это пустой стек?), **top** (вернуть элемент сверху стека, содержимое стека НЕ изменяется) и так далее.

- Создадим стек емкостью 5 элементов. Сначала стек пустой.
- Кладем в стек (push) синий цвет (blue) и смотрим, какой цвет будет на вершине стека.
- Аналогично кладем цвета red, green, purple.
- Кладем цвет orange. Стек полный. Больше в него нельзя ничего класть. Будет переполнение стека (stack overflow).

Поиск

Поиск

Раздел «Язык Си»

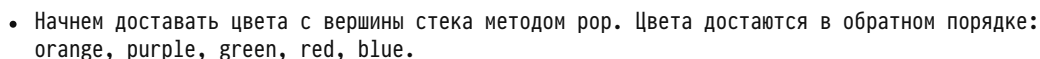
Главная
Зачем учить C?
Определения

Инструменты:
Поиск
Изменения
Index
Статистика

Разделы

Информация
Алгоритмы
Язык Си
Язык Ruby
Язык Ассемблера
EJ Judge
Парадигмы
Образование
Сети
Objective C

Logon>>



Пусть в стеке хранятся данные типа Data (мы будем пока считать и использовать в дальнейших примерах, что это тип int).

В качестве стека объявим массив из 10 элементов типа Data. Сразу положим в стек значения 5, 7, -3

Положим в этот стек числа 5, 7, -3:

Рисунок: стек с числами, вопросительными знаками и индексами.

Решим простейшую задачу – распечатаем содержимое стека.

Но как мы узнаем, где заканчиваются данные и начинается "мусор"? Мы не можем сделать специальное значение, как в строках '\0', чтобы пометить конец данных. Если решим использовать 0 как "конец данных", то как нам хранить значение 0?

Давайте, как в случае длинной арифметики, будем хранить информацию о том, где заканчиваются данные в переменной n.

Можно хранить в n, как в длинной арифметики, индекс последней ячейки с данными. n = 2;

Можно хранить в n индекс первой пустой ячейки или количество ячеек с данными. n = 3;

Что выбрать?

Рассмотрим пустой стек. В индекс "последней ячейки с данными" будет -1, а количество хранимых чисел (или индекс первой пустой ячейки) 0. Выберем второй вариант (он понятнее для пустого стека).

Так как массив a – это ячейки стека, а n – счетчик стека, то стоит эти переменные хранить вместе, в одной структуре. Назовем ее Stack.

```
struct Stack{
    Data a[N];    // элементы стека
    int n;        // сколько элементов хранится в стеке
};
```

Теперь в функции main создадим стек (сразу с данными, как на рисунке выше) и попробуем распечатать его содержимое:

(Мы будем "есть медведя по кускам" – сначала писать наброски кода, потом оформлять их в отдельную функцию. Если вы сразу можете придумать, как написать функцию печати стека, напишите ее и отладьте. Потом читайте дальше.)

Рисунок: массив a и поле n объединены в структуру, переменная называется st

```
int main() {
    struct Stack st = {{5, 7, -3}, 3}; // Data a[N] = {5, 7, -3}, int n = 3;
    // печатаем содержимое стека
    int i;
    for(i = 0; i < st.n; i++) {
        print("%d ", st.a[i]);
    }
    printf("\n");
    return 0;
}
```

Выделим код печати в отдельную функцию stack_print. Она ничего не должна возвращать. Передавать в нее будем стек.

Вопрос – что лучше передать в функцию – копию структуры или ее адрес?

Копировать массив (внутри структуры) – нехорошо. Лучше передадим на него указатель.

```
void stack_print(struct Stack * s) {
    int i;
    for(i = 0; i < s->n; i++) {
        print("%d ", s->a[i]);
    }
    printf("\n");
}

int main() {
    struct Stack st = {{5, 7, -2}, 3};
    stack_print(&st); // 5 7 -2
    return 0;
}
```

Добавим на вершину стека число 11. Стек растет в сторону больших индексов массива.

Поле n – номер первой пустой ячейки, куда будем добавлять новое значение.

Написать функцию нужно так, чтобы после ее использования печать по-прежнему работала правильно. Функция stack_push должна аналогично stack_print ничего не возвращать, в нее должен передаваться стек (передадим его адрес) и туда должно передаваться значение data, которое мы кладем в стек.

Напишем сначала проверку работы этой функции в main:

```
int main() {
    struct Stack st = {{5, 7, -2}, 3};
    stack_print(&st); // 5 7 -2
    stack_push(&st, 11);
    stack_print(&st); // 5 7 -2 11
    return 0;
}
```

После этого реализуем функцию `stack_push`

```
void stack_push(struct Stack * s, Data data) {
    s->a[s->n] = data;
    s->n ++;
}
```

Аналогично допишем тесты на `stack_pop` и реализуем эту функцию.

```
void stack_print(struct Stack * s) {
    int i;
    for(i = 0; i < s->n; i++) {
        print("%d ", s->a[i]);
        printf("\n");
    }
}

int main() {
    Data d;
    struct Stack st = {{5, 7, -2}, 3};
    stack_print(&st);          // 5 7 -2

    stack_push(&st, 11);
    stack_print(&st);          // 5 7 -2 11

    d = stack_pop(&st);
    stack_print(&st);          // 5 7 -2
    printf("d = %d\n", d);    // d = 11

    return 0;
}
```

Заметьте, ни в `stack_push`, ни в `stack_pop` нет проверок – не вышли ли мы за границы массива. Пусть эти проверки делает пользователь нашей библиотеки функций работы со стеком. Для этого дадим ему функции `stack_is_empty` и `stack_is_full`. В функции передают указатель на стек и они возвращают истину или ложь (с точки зрения языка C).

Реализуйте эти функции самостоятельно.

Вопрос: можно ли создать работоспособный стек так:

```
struct Stack st;
stack_print(&st);
```

Быть может, вторая строка приведет к падению программы.

Потому что если `st` – локальная переменная, то в поле `n` может быть любое число. Например, `-147`. Цикл в функции `stack_print` при этом выйдет далеко за границы выделенной памяти.

Значит, нужна функция, которая бы готовила стек к работе. Назовем ее `stack_init`.

```
void stack_init(struct Stack * s) {
    s->n = 0;
}

int main() {
    struct Stack st;

    stack_init(&st);          // без init другие функции работы со стеком будут работать неправильно

    stack_push(&st, 5);
    stack_push(&st, 7);
    stack_push(&st, -2);
    stack_print(&st);          // 5 7 -2

    return 0;
}
```

Контрольный вопрос: В каких функциях передача стека по значению привела бы к полной неработоспособности функции:

- `void stack_print(struct Stack s);`
- `void stack_init(struct Stack s);`
- `void stack_push(struct Stack s, Data d);`
- `Data stack_pop(struct Stack s);`
- `int stack_is_empty(struct Stack s);`
- `int stack_is_full(struct Stack s);`
- `int stack_is_size(struct Stack s);`

Стек на основе динамического массива

Стек, написанный выше, плох тем, что в него нельзя положить больше N элементов. И это N задано на этапе компиляции.

В конкретной задаче можно предположить, какой размер стека взять, но если мы пишем библиотеку функций, которыми будут пользоваться разные программисты, мы ничего не можем сказать об их задачах.

Сделаем N большое, но во-первых, это сегодня N достаточно большое, а через год у нас может появиться задача с еще большим набором данных.

С другой стороны, если в задаче нужно много небольших стеков, то наш стек будет занимать неоправданно много места в памяти.

Хочется, чтобы размер занимаемой памяти увеличивался и уменьшался по необходимости. Для этого будем выделять ее динамически.

Что для этого надо изменить?

- При объявлении структуры нужно заменить массив `Data a[N]` на указатель `Data * a`
- Так как изменение выделенной памяти функцией `realloc` затратно, не будем увеличивать или уменьшать память на каждый `push` и `pop`, а выделим памяти чуть больше. И будем ее увеличивать на еще *несколько* элементов, когда в полный стек попытаются положить еще один элемент. Получим саморасширяющийся стек.

То есть у нас две разные характеристики – сколько данных хранится в стеке (число n) и на сколько данных выделена память (число $size$). $n \leq size$

Структура данных `Stack`:

```
struct Stack {
    Data * a;           // указатель на динамически выделенную память под элементы стека
    unsigned int size;  // на сколько элементов выделена память
    unsigned int n;     // сколько элементов хранится в стеке
};
```

Какие функции не изменятся?

- `void stack_print(struct Stack * s);`
- `unsigned int stack_size(struct Stack * s);`
- `int stack_is_empty(struct Stack * s);`
- `int stack_is_full(struct Stack * s);` – не нужна, стек будет саморасширяться

Пусть функция `stack_init` создает пустой стек на 10 элементов.

```
#define N 10
void stack_init(struct Stack * s) {
    s->n = 0;           // пустой стек
    s->size = N;        // емкостью N элементов
    s->a = malloc(s->size * sizeof(Data));
}
```

В функцию `stack_push` нужно дописать это "саморасширение". Пусть каждый раз стек **увеличивается на N элементов**.

Есть разные стратегии увеличения размера стека – на постоянную величину, на сколько-то процентов и так далее. Выберите сами.

```
void stack_push(struct Stack * s, Data data) {
    // если стек полон, его нужно увеличить на 10 элементов
    if (s->n == s->size) {
        s->size += N;
        s->a = realloc(s->a, s->size * sizeof(Data));
    }
    s->a[s->n] = data;
    s->n ++;
}
```

Напишите функцию `main` для тестирования работы этого стека. Запустите полученный код под `valgrind`. Утечка памяти? Значит в наборе функций не хватает функции создания и удаления стека `stack_new` и `stack_delete`.

В `stack_new` не будем ничего передавать (или будем передавать начальный размер стека). Функция должна возвращать созданный пустой стек, готовый к работе.

Функция `stack_delete` должна получать стек, освободить память (и возвращать `NULL`???)

```
struct Stack * s = stack_new();

stack_push(s, 5);
stack_push(s, 7);
stack_push(s, -2);
stack_print(s);
```

```
stack_delete(s); // или s = stack_delete(s); ????
```

Рисунок: Порядок выделения памяти в stack_new

Как видно из рисунка, выделить память и инициализировать поля нужно в следующем порядке:

1. выделить память под сам стек (поля a, n, size).
2. инициализировать поле n
3. инициализировать поле size
4. выделить память под динамический массив элементов стека и сохранить его адрес в поле a.

```
struct Stack stack_new() {
    struct Stack * s;
    s = malloc(sizeof(struct Stack)); // пункт 1
    stack_init(s);                  // пункты 2, 3, 4
    return s;
}
```

Функцию stack_delete напишите самостоятельно.

Обратите внимание, что если при создании стека функция malloc была использована 2 раза, то и функция free должна быть вызвана тоже 2 раза.

Подумайте в каком порядке надо ее вызвать для указателя на стек s и указателя на массив элементов s->a.

Стек с выделением всей необходимой памяти единым куском

Для решения каких задач может использоваться стек?

Примеры использования стека

Вычисление постфиксной записи

Мы привыкли записывать арифметические выражения в *инфиксной* форме, где оператор (сложить, умножить) стоит между операндами.

2 + 3 =

Это же выражение можно записать в *постфиксной* форме, где оператор стоит после операндов.

2 3 + =

Infix	Postfix
2 + 3	2 3 +
2 + 3 - 4	2 3 + 4 -
2 * 3 + 4 * 5	2 3 * 4 5 * +
2 + 3 * 4 + 5	2 3 4 * + 5 +
(2 + 3) * (4 + 5)	2 3 + 4 5 + *

Заметим, что для указания приоритета операций в инфиксной записи нужны скобки, а в постфиксной скобок не нужно.

Как вычислить значение постфиксного выражения с помощью стека?

Будем класть в стек числа по следующим правилам:

- Пока есть входные данные
 - если это число, кладем (push) в стек.
 - если это '=', заканчиваем разбор входных данных.
 - если это операция (+, -, *), то
 - достаем (pop) нужные операнды из стека,
 - вычисляем значение выражения
 - результат кладем в стек
- После окончания разбора входных данных, результат всего выражения лежит на вершине стека.

Разберем, как работает этот алгоритм и что лежит в стеке на примере вычисления выражения "(1 + 2) * (3 + 4)". В постфиксной форме это "1 2 + 3 4 + *"

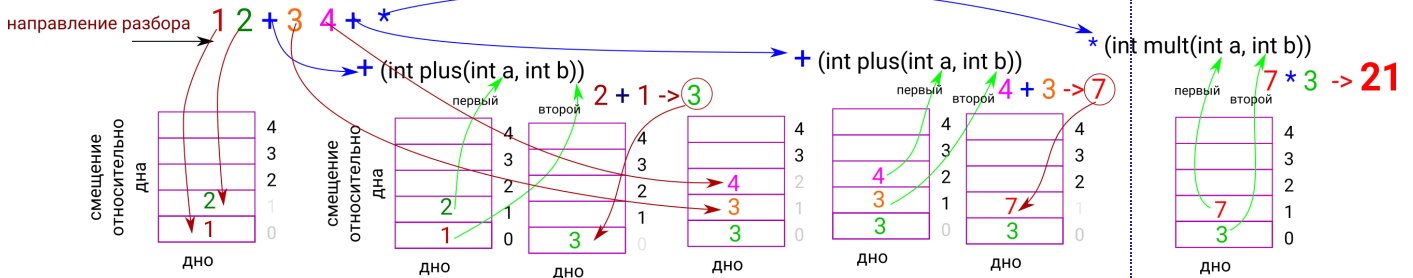


Рисунок3. Вычисление постфиксной записи "1 2 + 3 4 + *"

Задачи

Даны примеры реализации стека

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

typedef int Data;

typedef struct{
    int n; // количество элементов в стеке
    int capacity; // глубина стека (сколько можно поместить)
    Data *dt; // указатель на массив элементов (собственно стек)
} Stack;

// Создание стека (в этом случае сразу выделяется память на
// максимальный размер стека)

Stack * createStack(int capacity){
    // выделили память под объект типа Stack
    // память выделена под:
    // int n, int capacity и для указателя на Data
    // память под массив не выделялась
    Stack * tmp = (Stack*) malloc (sizeof(Stack));
    // заполним поля
    tmp->n = 0; // ничего не лежит
    tmp->capacity = capacity; // размер стека
    // выделение памяти под хранилище стека
    tmp->dt = (Data*)calloc(capacity, sizeof(Data));
    // вернуть указатель на созданный стек
    return tmp;
};

// удаление стека, освобождение памяти, выделенной под стек
void destroy(Stack * st){
    // проверяем не нулевой ли указатель
    if (st){
        // освобождение памяти под хранилище стека
        free(st->dt);
        // освобождение памяти, выделенной под стек
        free(st);
    }
    // для будущего использования этого указателя
    st = 0;
};

// помещаем данные в стек
// если стек полон, то возвращаем -1
int push(Stack* st, Data a){
    // проверка полон ли стек
    if (st->n == st->capacity)
        return -1;
    // добавляем элемент в вершину стека
    st->dt[st->n] = a;
    // увеличиваем количество элементов в стеке
    st->n++;
    return 0;
};

int isStackEmpty(Stack *){
    // проверка не пуст ли стек
}

int isStackFull(Stack *){
    // проверка не полон ли стек
}
```

```

}

void clearStack(Stack *){
// очистка стека - должен быть пуст
};

// достаём данные из стека
// если стек пуст, возвращаем -2

int pop(Stack* st, Data * a){
// написать код с проверкой пуст ли стек
};

void printStack(Stack *){
// код для печати состояния стека:
// размер, количество элементов в стеке
// все элементы стека, начиная со дна
}

// примеры проверки функций работы стека
int main(){
    Stack *st = 0; // указатель на будущий стек
    // создать стек на 5 элементов
    st = createStack(5);
    // распечатать стек
    printStack(st);
    // проверка заполнения стека с попыткой переполнения
    int i;
    for( i = 0; i < 7; i ++){
        int z;
        scanf("%d", &z);
        if (push(st, z) == -1)
            printf("stack overflow\n");
        printStack(st);
    }
    // проверка извлечения одного элемента из стека
    Data a;
    if( pop(st, &a) == -2 )
        printf("stack is empty\n");
    printf("a: %d\n", a);
};

```

Задача 1.

Реализовать и отладить все функции из примера

Задача 2.

С помощью стека решить задачу перевода десятичного числа, записанного как строка символов в двоичный код. Размер строки не превышает 100 символов.

Задача 3.

Решить задачу вычисления выражения, записанного в постфиксной форме.

Для удобства использования "+" и "*" учесть следующий пример использования указателей на функции

```

#include <stdio.h>
#include <stdlib.h>
// указатель на функцию вида int <имя функции>(int, int);
typedef int (*mathAction)(int, int);
// первая функция вида int <имя функции>(int, int);
int plus(int a, int b){
    return a + b;
};
// вторая функция вида int <имя функции>(int, int);
int mult( int a, int b){
    return a * b;
};

int main(){
    int a, b, i, res;
    scanf("%d%d", &a, &b);
    // пример использования указателя на функции
    /*
    mf = plus; // присвоили адрес plus
    printf("%d ", mf(a, b)); // вызов функции (будет plus)
    */
}

```



```

mf = mult; // присвоили указатель на mult
printf("%d ", mf(a, b)); // вызов функции (будет mult)
*/

// массив из двух указателей на функции
// сразу инициализируем plus и minus
mathAction mf[2] = {mult, plus}; // инициализация

//пример использоания массива указателей на функции
for(i = 0; i < 2; i++){
// вызов функций из массива
    res = mf[i](a, b);
    printf("%d ", res);
}
// учтем, что код * - 42, а код + - 43
// Тогда вызов функций из массива в зависимости от символа может быть такой:
// Умножить:
    char c;
    c = '*';
    res = mf[ c - '*' ](a, b);
// Сложить:
    c = '+';
    res = mf[ c - '*' ](a, b);
return 0;
}

```

Перевод из инфиксной записи в постфиксную

TODO

Вычисление инфиксной записи с помощью стека

TODO

Ответы на контрольные вопросы

Стек, который не переполнится, но теряет значения

```

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

typedef struct
{
    int *storage; // место для хранения
    int size; // размер хранилища
    int *head, *last; // голова стека(куда класть) и предыдущий (уже положенный) элемент
    int n; // смещение головы от начала хранилища
}Stack;

// создание стека
Stack* createS(int size)
{
    Stack *tmp = malloc(sizeof(Stack));
    tmp->size = size;
    tmp->storage = calloc(size, sizeof(int));
    tmp->head = tmp->storage;
    tmp->n = 0;
}

// Удаление стека
Stack* destroyS(Stack* st){
    if(st )
    {
        if (st->storage)
            free(st->storage);
        st->storage = st->head = 0;
        free (st);
        st = 0;
    }
    return st;
}

// Кладем элемент в голову
void pushS(Stack * st, int a){
    st->head[0] = a;
    st->last = st->head; // предыдущий элемент получает адрес текущего
// сдвигаем голову
    st->n += (st->size + (st->size - 1));
    st->n %= st->size;
}

```

```

        st->head = st->storage + st->n;
    }

    // достаем элемент
    int popS(Stack* st)
    {
        int ret = st->last[0];
        st->last[0] = 0;
        // голова "сползла" на предыдущий элемент
        // c.lf vj;yj dcnfdkznm
        st->head = st->last;
        //вычисляем где теперь предыдущий
        st->n += (st->size + 1);
        st->n %= st->size;
        st->last = st->storage + (st->n + 1)% st->size;
        return ret;
    }

    // печать стека
    void prSt(Stack* st){
        int i, n;
        int *p = st->storage;
        for( i = 0; i< st->size; i++){
            printf("%d ", p[(st->n + 1 + st->size + i )% st->size]);
        }
        printf("\n");
    }

    int main(){
        Stack *st = 0;
        st = createS(5);
        int a, p;
        char ch = 0;
        pushS(st, 1);
        prSt(st);
        pushS(st, 2);
        prSt(st);
        pushS(st, 3);
        prSt(st);
        pushS(st, 4);
        prSt(st);
        pushS(st, 5);
        prSt(st);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        a = popS(st);
        printf("get: %d\n", a);
        // Пример меню
        // printf("1: вставить\n2: извлечь и напечатать\n3: напечатать все\n");
        /* while ( (p = scanf("%hhd", &ch)) != EOF && p > 0)
            switch (ch)
            {
                case 1: scanf("%d", &a);
                        pushS(st, a);
                        break;
                case 2: a = popS(st);
                        printf("get: %d\n", a);
                        break;
                case 3: prSt(st);
            };
        */
        st = destroyS(st);
        return 0;
    }

```

Закраска объектов картинки

Для задачи "Следствие ведет колобки" из сборника

http://acm.mipt.ru/twiki/pub/Cintro/LongTask_Retro/odarka97.pdf описаны следующие структуры и функции в заголовочном файле **picture.h**:

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
// Хранение картинки
typedef struct
{
    char* pict; // дин. массив для ВСЕЙ картинки
    int n, m; // размер картинки по вертикали и горизонтали
}Picture;

// Координаты ячейки для размещения в стеке
typedef struct cells
{
    int x; // координаты точки на карте
    int y;
}Cell; // Все новые типы данных всегда называются с заглавных букв

// Стек для записи помеченных, но еще необработанных ячеек.
typedef struct{
    Cell *storage; // адрес начала стека
    Cell * sp;      // указатель на вершину стека (куда кладем следующий)
    int size; // размер стека (в Cell)
}Stack;

// Функции

// Для получения и печати картинки:
// предполагается, что картинка хранится в текстовом файле
// в следующем формате:
// первая строка - два целых числа n и m - размеры
// далее n строк по m символов (либо '.', либо '*')
void getPicture(Picture * img, FILE *fl);
// Печать картинки (для отладки и как результат)
void printPicture(const Picture * img);
// Возвращает указатель на начало строки номер row
char * rowN(const Picture * img, int row);

// Для организации стека размера size
// создание стека на size координат
void createS(Stack * st, int size);
// проверка пустой ли стек (0 - если не пуст, 1 - если пуст)
int isEmptySt(const Stack * st);
// проверка полон ли стек: (0 - не полон, 1 - полон)
int isFullSt(const Stack * st);
// добавление элемента в вершину стека
// если добавить не удалось, возвращает 0
int push(Stack * st, Cell cel);

// извлечение (с удалением) элемента из стека
// если извлечь не удалось, возвращает 0
int pop(Stack* st, Cell * cell);
// Печать содержимого стека (для отладки)
void prStack(Stack *st);

// Для "закраски" одного объекта
// НЕРЕКУРСИВНАЯ функция, использующая стек

// Начиная с клетки start, объект закрашивается номером number,
// Функция получает заведомо закрашенную клетку объекта
void fill(Picture * img, Cell start, unsigned char number);

// Закрашиваются все объекты и возвращается их количество
int fillAll(Picture * img);

```

Файл **stack.c** - реализация функций работы со стеком

```

#include "picture.h"

// Функции

// Для организации стека размера size
// создание стека на size координат
void createS(Stack * st, int size){
    st->storage = calloc(size, sizeof(Cell));
    st->sp = st->storage;
    st->size = size;
}

```

```

};

// проверка пустой ли стек (0 - если не пуст, 1 - если пуст)
int isEmptySt(const Stack * st){
    return (st->sp <= st->storage);
};

// проверка полон ли стек: (0 - не полон, 1 - полон)
int isFullSt(const Stack * st){
    // printf("st->sp: %p basa+%d %p\n", st->sp, )
    return (st->sp >= st->storage + st->size)
};
// добавление элемента в вершину стека
int push(Stack * st, Cell cel){
    if(isFullSt(st))
        return 0;
    (st->sp)[0] = cel;
    st->sp++;
    return 1;
};
// извлечение (с удалением) элемента из стека
int pop(Stack* st, Cell * cl){
    if (isEmptySt(st))
        return 0;
    st->sp--;
    cl[0] = st->sp[0];

    return 1;
};

void prStack(Stack *st){
    Cell *p;
    p = st->sp - 1;
    while(p >= st->storage){
        printf("(%d, %d) ", p[0].x, p[0].y);
        p--;
    }
    printf("\n");
};

```

Файл **testSt.c** – проверка работы функций стека

```

#include "picture.h"

void prCell(Cell a){
    printf("(%d, %d) ", a.x, a.y);
};

int main(int argc, char** argv){
    Stack st;
    createS (&st, 10);
    Cell a[5];
    Cell check;
    int i;
    for(i = 0; i < 5; i ++){
        {
            scanf("%d%d", &(check.x), &(check.y));
            if(!push(&st, check ))
            {
                printf("стек полон\n");
                exit(1);
            }
            prStack(&st);
        }
        while(pop(&st, &check))
        {
            printf("get from stack: ");
            prCell(check);
        }
        printf("стек пуст\n");
        return 0;
    }
}

```

Компиляция и запуск

```

>gcc stack.c testSt.c -o tSt
>./tst

```

Файл **picture.c** – реализация некоторых функций для работы с картинкой

```

#include "picture.h"

// Функции

// Возвращает указатель на начало строки номер row
char * rowN(const Picture * img, int row)
{
    return img->pict + row * img->m;
};

// Для получения и печати картинки:
// предполагается, что картинка хранится в текстовом файле
// в следующем формате:
// первая строка - два целых числа n и m - размеры
// далее n строк по m символов (либо '.', либо '*')
void getPicture(Picture * img, FILE *fl){
    // Файл всегда открывается в main
    fscanf(fl, "%d%d\n", &(img->n), &(img->m));
    int i;
    char buf[1000] = {0};
    char *p = 0;
    // выделение памяти для картинки
    img->pict = calloc(img->n * img->m + 1, sizeof(char));
    printf("n: %d m : %d\n", img->n, img->m);
    p = img->pict;
    p[0] = '\0';
    // чтение данных из файла
    for(i = 0; i < img->n; i++)
    {
        //
        fgets(buf, 999, fl);
        int len = strlen(buf);
        buf[len - 1] = '\0';
        //
        printf("%d %s ", i, buf);
        strcat(p, buf);

        printf(" %d: %s\n", i, p);
    }
};

void printPicture(const Picture * img){
    int x, y;
    for ( y = 0; y < img->n; y++)
    {
        for(x = 0; x < img->m ; x++)
        // rowN() возвращает указатель на char? значит можно представить его
        // массивом и обратиться к элементу со смещением x
        printf("%c", rowN(img, y)[x]);
        printf("\n");
    }
}

```

Файл **test.c** - проверка функций для работы с картинкой.

```

#include "picture.h"

int main(int argc, char** argv){
    FILE * fl;
    if(argc < 2)
    {
        printf("Аргументы!!\n");
        exit(1);
    }
    fl = fopen(argv[1], "r");
    if(errno)
    {
        perror("File :(\n");
        exit(1);
    }
    Picture img;
    getPicture(&img, fl);
    printPicture(&img);
    return 0;
}

```

Компиляция и запуск

```
>gcc picture.c test.c -o test
>./test
```

Если написана программа, которая использует функции из **stack.c** и из **picture.c** :

```
>gcc picture.c stack.c myprog.c -o myprog
```




Задачи из сборника

http://acm.mipt.ru/twiki/pub/Cintro/LongTask_Retro/odarka97.pdf :

1. Решить задачу "Следствие ведут колобки" в части разделения объектов. Если кто-то сможет подсчитать разные – это ОЧЕНЬ БОЛЬШОЙ плюс.

2. Решить задачу "Ученая блоха" в части нахождения одного любого пути или его отсутствия. Если кто-то решит всю задачу – ОЧЕНЬ БОЛЬШОЙ плюс Только те функции, которые не изменяют содержимого полей стека. Это `stack_print`, `stack_size`, `stack_is_empty`, `stack_is_full`.

-- TatyanaDerbysheva – 08 Jan 2019

Attachment 	Action	Size	Date	Who	Comment
 piram.png	manage	74.8 K	10 Feb 2019 – 16:54	TatyanaOvsyannikova2011	
 money.png	manage	288.0 K	10 Feb 2019 – 16:54	TatyanaOvsyannikova2011	
 stack_push	manage	196.1 K	10 Feb 2019 – 17:01	TatyanaOvsyannikova2011	
 stack_pop	manage	192.0 K	10 Feb 2019 – 16:57	TatyanaOvsyannikova2011	
 prefix.png	manage	159.1 K	10 Feb 2019 – 18:09	TatyanaOvsyannikova2011	

(с) Материалы раздела "Язык Си" публикуются под лицензией GNU Free Documentation License.