Семинар #7: Указатели и выделение памяти.

Шестнадцатиричная система счисления

Хранение переменных в памяти

Указатели

У каждой переменной есть адрес. Адрес - это номер байта, начиная с которого лежит эта переменная в памяти. Чтобы найти адрес переменной, нужно перед ней поставить знак амперсанда `&`.
Для хранения адресов в языке C введены специальные переменные, которые называются указатели. Тип переменной указателя = тип той переменной, на которую он 'указывает' + звёздочка(`*`) на конце. Например, указатель, который будет хранить адреса переменных типа `int` должен иметь тип `int*`.
Чтобы по указателю получить саму переменную, нужно перед указателем поставить звёздочку(`*`).

```
#include <stdio.h>
int main()
{
    int a = 7;
    printf("Value = %d. Address = %p\n",  a,  &a);

    int* pa = &a;
    printf("Value = %d. Address = %p\n", *pa, pa);
}
```

Выполните задание из файла `0pointers.c`.

Арифметика указателей

С указателями можно производить следующие операции:

- Прибавить или отнять число `p + 2`
- Вычитать 2 указателя `p - q`
- Разыменование (получить то, на что указывает указатель) `*p`
- Квадратные скобки (прибавить число + разыменование): `p[i] == *(p+i)`

Пусть есть одномерный статический массив и указатель на 4-й элемент этого массива:

```
int numbers[6] = {4, 8, 15, 16, 23, 42};
int* p = &numbers[3];
```

Чему равны следующие выражения:

1. `numbers[5]`
2. `*p`
3. `*(p+1)`
4. `*(p-2)`
5. `p[0]`
6. `p[1]`
7. `p[-2]`
8. `*numbers`
9. `*(numbers+5)`
10. `p - numbers`
11. `(short*)p - (short*)numbers`
12. `(char*)p - (char*)numbers`

Подсказка: имя массива во многих случаях ведёт себя как указатель на первый элемент массива.
Выполните задание из файла `1pointerarith.c`.

Malloc и free:

Основные функции для динамического выделения памяти:

- `void* malloc(size_t n)` – выделяет n байт и возвращает указатель **void*** на начало этой памяти
- `void free(void* p)` – освобождает выделенную память
- `void* realloc (void* p, size_t new_n)` – перевыделяет выделенную память

Пример работы с malloc и free:

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

int main()
{
        // Выделяем 50 байт памяти, адрес первого байта будет храниться в указателе p
        void* p = malloc(50);

        // Освободим только - что выделенные 50 байт
        // Память можно освободить в люой момент выполнения, экономя память
        free(p);

        // Выделяем 12 байт памяти, с указателем p1 теперь
        //       можно обращаться как с массивом размера 3
        int* p1 = malloc(12);

        // Выделяем объём памяти достаточный для хранения 15 - ти int - ов
        int* p2 = malloc(15 * sizeof(int));

        // Теперь с p1 и p2 можно работать также как и с массивами типа int
        // То есть можно применять операции типа p1[2]
        // И p1 и p2 будут вести себя как массива размера 3 и 15 соответственно
        for (int i = 0; i < 3; ++i)
                scanf("%d", &p1[i]);
        printf("%d", p1[0] + p1[2]);

        // Увеличим размер нашего массива с 15 до ти25-
        p2 = realloc(p2, 25 * sizeof(int));

        // Не забывайте освобождать ненужную память!
        free(p1);
        free(p2);
}
```

Задачи:

1. **Основы malloc/free**

   - Выделить 123 байта памяти и записать адрес на эту память в указатель типа `void*`. Попробуйте разыменовать этот указатель, что при этом произойдёт?
   - Выделить 100 байт памяти и записать адрес на эту память в новый указатель типа `int*`.
   - Выделить память для хранения 1 элемента типа `int`.
   - Выделить память для хранения 10 элементов типа `unsigned long long`.
   - Выделить память для хранения 100 элементов типа `float*`.
   - Выделить память для хранения 10 элементов типа `double`. Изменить размер этого динамического массива с 10 до 50, используя `realloc`.
   - Освободить всю память, которую вы выделили.

Стек

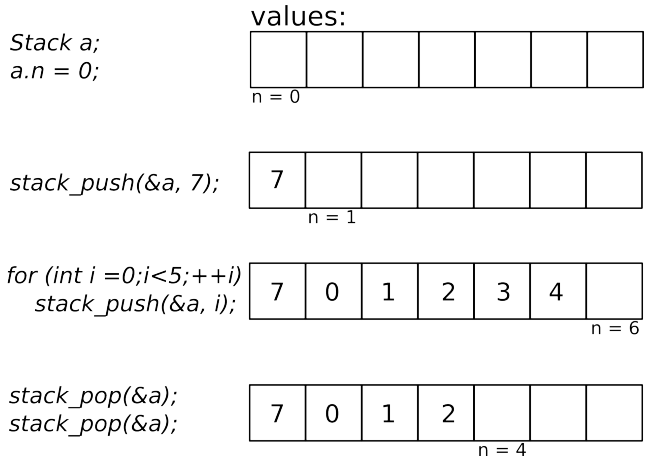
```c
#include <stdio.h>
struct stack
{
    int n;
    int values[100];
};
typedef struct stack Stack;

void stack_push(Stack* s, int x)
{
    s->values[s->n] = x;
    s->n += 1;
}

int main()
{
    Stack a;
    a.n = 0;
    stack_push(&a, 4);
    stack_push(&a, 10);
    stack_pop(&a);
    printf("%d\n", stack_pop(&a));
}
```

Стек — абстрактный тип данных, представляющий собой список элементов, организованных по принципу «последним пришёл — первым вышел».

Реализация с помощью массива:



Задачи:

1. Написать функцию `int stack_pop(Stack* s, int x)`. Протестируёте стек: проверьте, что выведет программа, написанная выше.
2. Написать функцию `int stack_is_empty(Stack* s)`, которая возвращает 1 если стек пуст и 0 иначе.
3. Написать функцию `int stack_get(Stack* s)`, которая возвращает элемент, находящийся в вершине стека, но не изменяет стек.
4. Написать функцию `void stack_print(Stack* s)`, которая распечатывает все элементы стека.
5. Одна из проблем текущей реализации: размер массива 100 задан прямо в определении структуры. Если мы решим изменить максимальный размер стека, то придётся изменять это число по всему коду программы. Чтобы решить эту проблему введите #define-константу CAPACITY:

   ```c
   #define CAPACITY 100
   ```

6. Что произойдёт, если вызвать `stack_push()` при полном стеке? Обработайте эту ситуацию. Программа должна печатать сообщение об ошибке и завершаться с аварийным кодом завершения. Чтобы завершить программу таким образом можно использовать функцию `exit()` из библиотеки `stdlib.h`. Пример вызова: `exit(1);`
7. Аналогично при вызове `stack_pop()` и `stack_get()` при пустом стеке.
8. Введите функцию `stack_init()`, которая будет ответственна за настройку стека сразу после его создания. В данном случае, единственное, что нужно сделать после создания стека это занулить n.
9. Предположим, что вы однажды захотите использовать стек не для целочисленных чисел типа `int`, а для какого-нибудь другого типа (например `char`). Введите синоним для типа элементов стека:

   ```c
   typedef int Data;
   ```

Измените тип элемента стека во всех функциях с int на Data (тип поля n менять не нужно). Теперь вы в любой момент сможете изменить тип элементов стека, изменив лишь одну строчку.

10. Сложные скобки. Решить задачу определения правильной скобочной последовательности, используя стек символов. Виды скобок: () {} [] <>. Пример неправильной последовательности: ({<}>)

11. Стек с динамическим выделением памяти. Описание такого стека выглядит следующим образом:

```
struct stack
{
    int capacity;
    int n;
    Data* values;
};
typedef struct stack Stack;
```

Введено новое поле capacity. В нём будет хранится количество элементов стека, под которые уже выделена память. В отличии от предыдущего варианта стека, это значение будет меняться.
values теперь не статический массив, а указатель. Вы должны выделить необходимое место для стека в функции `stack_init()`. Начальное значение capacity можно выбрать самостоятельно либо передавать на вход функции `stack_init()`. При заполнении стека должно происходить перевыделение памяти с помощью функции `realloc`.