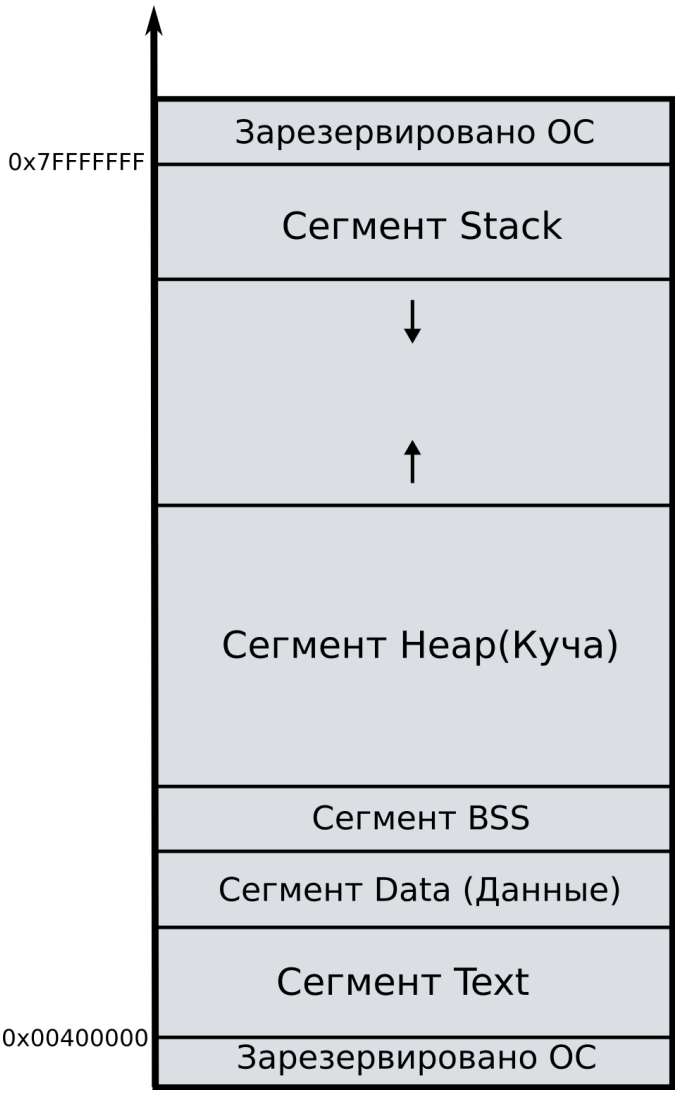
Семинар #9: Сегменты памяти. Динамический массив. Классные задания.

Часть 1: Сегменты памяти



1. **Сегмент памяти Стек (Stack)**

   - При обычном объявлении переменных и массивов все они создаются в стеке:

     ```
     int a;
     int array[10];
     ```

   - Память на локальные переменные функции выделяется при вызове этой функции и освобождается при завершении функции.
   - Маленький размер (несколько мегабайт, зависит от настроек операционной системы).
   - Выделение памяти происходит быстрее чем в куче

2. **Сегмент памяти Куча (Heap)**

   - Выделеть память в куче можно с помощью стандартной функции `malloc`.

     ```
     int* p = malloc(10 * sizeof(int));
     ```

   - Освободить память в куче можно с помощью стандартной функции `free`

     ```
     free(p);
     ```

   - Память можно выделяется/освобождать в любом месте.
   - Размер ограничен свободной оперативной памятью.
   - Выделение памяти происходит медленней чем в стеке

3. **Сегмент памяти Data**

   - В этом сегменте хранятся инициализированные глобальные и статические переменные а также строковые литералы

4. **Сегмент памяти BSS**

   - В этом сегменте хранятся неинициализированные глобальные и статические переменные
   - В большинстве систем все эти данные автоматически инициализируются нулями

5. **Сегмент памяти Text**

   - В этом сегменте хранится машинный код программы (Код на языке C, сначала, переводится в код на языке Ассемблера, а потом в машинный код. Как это происходит смотрите ниже.).
   - Адрес функции - адрес первого байта инструкций в этом сегменте.

Создание массива в разных сегментах памяти

Ниже представлен пример программы в которой создаюся 4 массива в разных сегментах памяти.

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

int array_data[5] = {1, 2, 3, 4, 5};
int array_bss[5];

int main() {
    int array_stack[5];
    int* array_heap = (int*)malloc(5 * sizeof(int));
}
```

- Напечатайте адрес начала каждого из массивов. Помните, что для печати адресов используется спецификатор `%p`.

Переполнение стека – Stackoverflow

- Определите размер стека на вашей системе экспериментальным путём. Создайте массив такого большого размера на стеке, чтобы перестала работать. Минимальный размер массива, при котором падает программа будет примерно равен размеру стека.
- При каждом вызове функции в стеке хранятся локальные переменные функции, аргументы функции а также адрес возврата функции. Даже функция без локальных переменных и аргументов будет хранить на стеке как минимум адрес возврата (8 байт). Определите размер стека на вашей системе экспериментальным путём с помощью рекурсии.

Статические переменные

Помимо глобальных переменных, в сегменте Data хранятся статические переменные. Такие переменные объявляются внутри функций, но создаются в сегменте Data и не удаляются при завершении функции. Вот пример функции со статической переменной.

```
#include <stdio.h>
void counter() {
    static int n = 0;
    n++;
    printf("%i\n", n);
}

int main() {
    counter();
    counter();
    counter();
}
```

Обратите внмание, что в этой функции строка `static int n = 0;` не исполняется при заходе в функцию. Эта строка просто объявляет инициализирует статическую переменную, причём инициализация происходит в самом начале исполнения программы (даже до функции `main`).

- Создайте функцию `adder`, которая будет принимать на вход число и возвращать сумму всех чисел, которые приходили на вход этой функции за время выполнения программы.

```
printf("%i\n", adder(10));  // Напечатает 10
printf("%i\n", adder(15));  // Напечатает 25
printf("%i\n", adder(70));  // Напечатает 95
```

Часть 2: Статический массив внутри структуры

В файле `0array_in_struct.c` содержится минимальный пример массива, который хранится внутри структуры. Максимальная вместимость массива равна `100`. А размер хранится внутри структуры и может принимать значения от 0 до 100. Функция `push_back` принимает на вход адрес на такую структуру и число `value`, а затем добавляет это число в конец массива.

Зачем хранить массив внутри структуры, если можно было бы просто создать его без струтуры? На самом деле у такого подхода много преимуществ:

1. Он позволяет нам самим описать поведение массива при добавлении и удалении элементов.
2. Мы можем передавать такой массив внутри функций также как и обычные переменные.
3. Такой подход распространяется на более сложные структуры данных

Напишите следующие функции для работы с этим массивом:

- `array_print` – эта функция должна принимать на вход адрес структуры `Array` и печатать массив на экран.
- `array_is_empty` – эта функция должна принимать на вход адрес структуры `Array` и возвращать `1`, если массив пуст и `0` иначе.
- `int array_get(const Array* a, int index)` эта функция должна возвращать число, которое лежит по индексу `index` в массиве.
- `void array_set(const Array* a, int index, int value)` эта функция должна устанавливать элемент массива, лежащий по индексу `index` значением `value`.
- Добавьте в функцию

Часть 3: Динамический массив

```
#include <stdio.h>
struct dynarray {
    int size;
    int capacity;
    int* values;
};
typedef struct dynarray Dynarray;
```

В файле `2dynarray.c` содержится код описывающий динамический массив. Измените код так, чтобы происходило перевыделение памяти тогда, когда размер массива начинает превышать вместимость.

Часть 5: Абстрактные типы данных: Стек и Очередь, Дек и Очередь с приоритетом

Абстракстный тип данных (АТД) - это математическая модель для типов данных, которая задаёт поведение этих типов, но не их внутреннею реализацию.

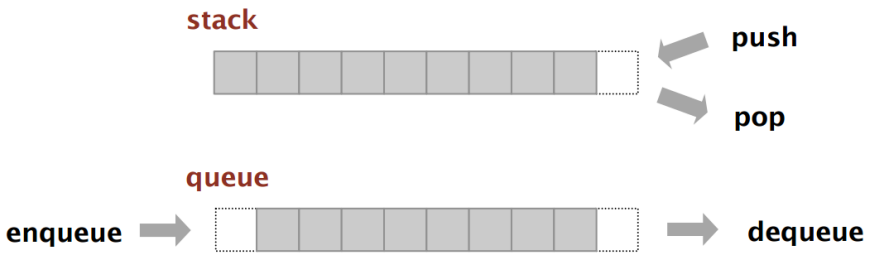
Стек (Stack) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `push` - добавить элемент в стек.
- `pop` - извлечь из стека последний добавленный элемент.

Таким образом, поведение стека задаётся этими двумя операциями. Так как стек - это абстрактный тип данных, то его внутренняя реализация на языке программирования может быть самой разной. Стек можно сделать на основе статического массива, на основе динамического массива(`malloc`/`free`) или на основе связного списка. Внутренняя реализация не важна, важно только наличие операций `push` и `pop`.
Не нужно путать абстракстный тип данных стек с сегментом памяти стек.

Очередь (Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `enqueue` - добавить элемент в очередь.
- `dequeue` - извлечь из очереди первый добавленный элемент из оставшихся.



Дек (Deque = Double-ended queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью четырёх операций:

- `push_back` - добавить элемент в конец.
- `push_front` - добавить элемент в начало.
- `pop_back` - извлечь элемент с конца.
- `pop_front` - извлечь элемент с начала.

Очередь с приоритетом (Priority Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

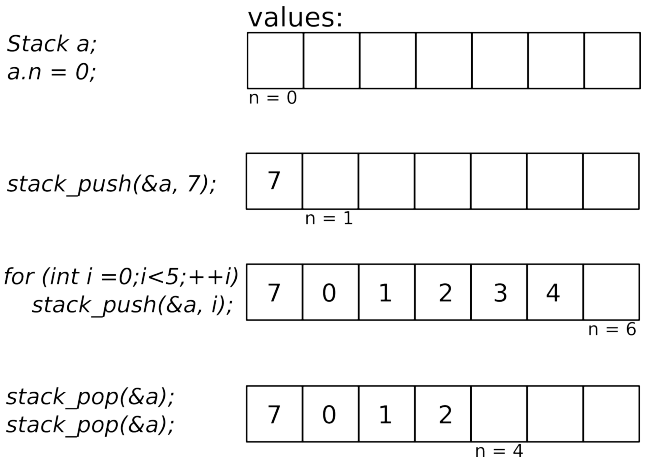
- `insert` - добавить элемент.
- `extract_best` - извлечь из очереди элемент с наибольшим приоритетом.

То, что будет являться приоритетом может различаться. Это может быть как сам элемент, часть элемента (например, одно из полей структуры) или другие данные, подаваемые на вход операции `insert` вместе с элементом. В простейшем случае, приоритетом является сам элемент (тогда очередь с приоритетом просто возвращает максимальный элемент) или сам элемент со знаком минус (тогда очередь с приоритетом возвращает минимальный элемент).

Часть 6: Реализация стека на основе статического массива

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

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



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(const Stack* s)`, которая возвращает 1 если стек пуст и 0 иначе.
3. Написать функцию `int stack_get(const Stack* s)`, которая возвращает элемент, находящийся в вершине стека, но не изменяет стек.
4. Написать функцию `void stack_print(const 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: ({<}>)

Часть 7: Реализация стека на основе динамического массива

Описание такого стека выглядит следующим образом:

```
struct stack {
    int capacity;
    int size;
    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`.