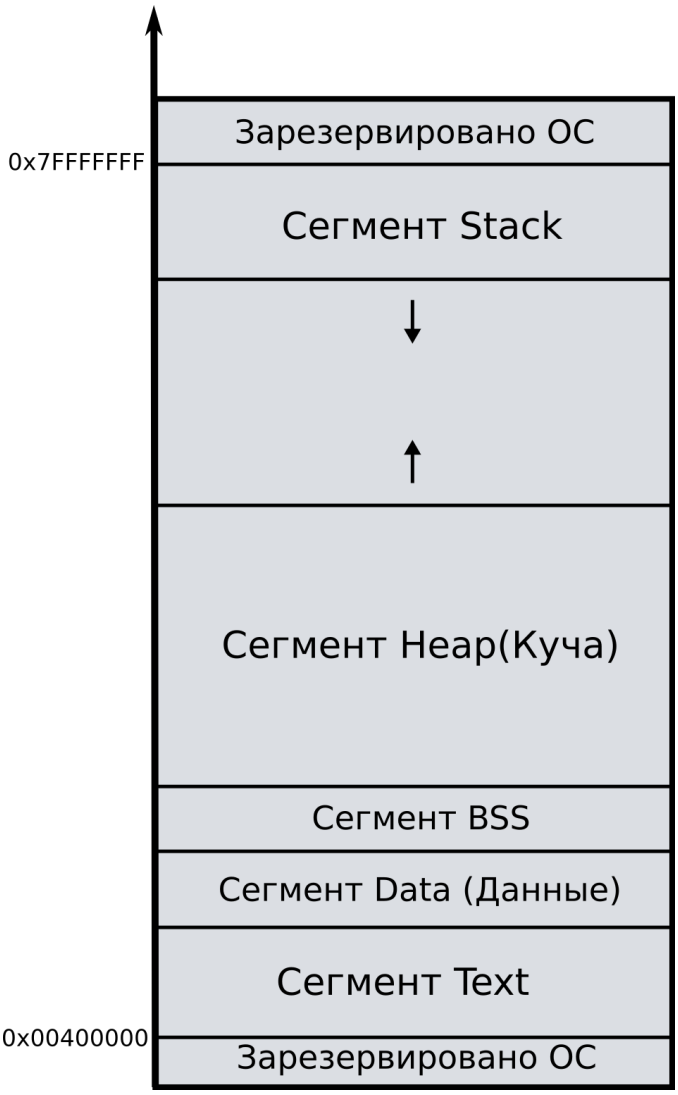
Семинар #11: Динамический массив. Стек и Очередь. Классные задания.

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.

```c
#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.

```c
#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.

```c
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

Задачи:

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`.
- `void array_erase(const Array* a, int index)` – эта функция должна удалять элемент под индексом `index`. При этом, конечно, все элементы, которые следовали после удаляемого нужно сместить влево на 1 элемент.
- `void array_insert(const Array* a, int index)` – эта функция должна вставлять элемент в массив в место между элементами с индексами `index - 1` и `index`. При этом, конечно, все элементы, которые находились правее этого места, должны сместиться вправо на один элемент.
- `void array_append(Array* a, const Array* b)` – эта функция должна добавлять всё содердимое массива `b` в конец массива `a`.
- **Проверка индекса:** Как известно, в языке `C` у статических массивов нет проверки на выход за пределы массива. Например, следующий код может сработать и не выдасть никакой ошибки.

  ```c
  int array[100] = {};
  printf("%i\n", array[101]);
  ```

  Компилятор просто преобразует строку `array[101]` в `*(array + 101)` и обратится к соответствующему участку памяти. Причина того, что в `C` нет такой проверки заключается в том, что она бы немного замедлила программу. Язык `C` ориентирован на максимальное быстродействие, поэтому и не делает такую проверку.

  Однако, за отсутствие такой проверки приходится платить тем что в программе могут появиться сложно выявляемые ошибки. Например, если вы случайно ошибётесь с индексами массива, то можете этого даже не заметить. Программа будет работать с памятью за пределами этого массива и почти всегда выдавать правильные результаты. Но иногда, выходя за пределы массива вы можете изменить другие переменные. Найти такую ошибку в большой программе может быть очень сложно.

  Добавьте в нашу реализацию массива проверку на принадлежность индекса правильному диапазону значений в функции `array_get` и `array_set`. Если индекс не входит в правильный диапазон, программа должна печатать сообщение об ошибке и завершаться. Завершить программу можно с помощью вызова `exit(1)`, функции `exit` из библиотеки `stdlib.h`.

- **Вместимость:** Текущая вместимость нашего массива – всего 100 элементов. Если размер массива превысит это значение, то тоже должна происходить ошибка. Добавьте проверку на превышение вместимости в функции `array_insert` и `array_append`. Чтобы не писать везде магическое число `100`, введите константу, которая будет задавать вместимость.

Часть 3: Динамический массив

У статического массива есть 2 очень больших недостатка:

- Во-первых, он выделяется на стеке и его размер сильно ограничен.
- Во-вторых, его вместимость является постоянным числом и не может меняться в процессе выполнения программы. Это всё ведёт к неэффективному использованию памяти, так как мы обязаны задавать размер статического массива таким, чтобы в него всё влезло при любых входных данных.

Эти проблемы исправляются в динамическом массиве. Структура для динамического массива выглядит следующим образом:

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

В этой структуре `size` означает размер массива (как и прежде). `capacity` означает текущую вместимость массива, в отличии от статического массива эта величина может меняться. `values` – это указатель на элементы массива, которые выделяются в куче. Количество памяти, выделенной в куче, всегда равно `capacity * sizeof(int)`.

Задачи:

Напишите следующие функции для работы с динамическим массивом:

- `Dynarray dynarray_create(size_t initial_capacity)` – эта функция должна создавать динамический массив с размером равным нулю и вместимостью равной `initial_capacity`. Память должна выделяться в куче с помощью `malloc`.
- `void dynarray_destroy(Dynarray* d)` – эта функция должна освобождать выделенную память (`free`).
- Перепишите все функции из прошлой части: `dynarray_push_back`, `dynarray_print` и другие.
- **Расширение массива:** Измените код так, чтобы происходило перевыделение памяти тогда, когда размер массива начинает превышать вместимость в функциях `dynarray_push_back` и `dynarray_insert`. Вместимость динамического массива должна увеличиваться в 2 раза. Это можно сделать двумя способами:
  - Выделить новый участок памяти в 2 раза больше прежнего, используя `malloc`. Переписать все элементы в новую память. Освободить старую память с помощью `free`.
  - Использовать функцию `realloc`, которая будет делать то же самое, но более эффективно.
- **Проверка на корректность:** Функции `malloc` и `realloc` не всегда могут выделить необходимую память. Например, если вы запросите больше чем вся оперативная память, то они ничего не смогут сделать. В этом случае эти функции возвращают нулевой указатель (т.е. указатель, равный `NULL`). В случае возникновения такой ошибки `realloc` не освобождает старую память. Добавьте в программу проверки на возникновения таких ошибок. Если память выделить нельзя, то программа должна печатать сообщение о нехватке памяти и завершаться.
- **Размер и вместимость:** Напишите программу, которая будет создавать стек вместимости `1` и добавлять в него последовательно `200` элементов. При каждом добавлении элемента печатайте размер и вместимость.
- **Другие типы элементов:** Предположим, что вы однажды захотите использовать динамический массив не для целочисленных чисел типа `int`, а для какого-нибудь другого типа (например `char`). Введите синоним для типа элементов динамического массива:

  ```
  typedef int Data;
  ```

  Измените тип элемента динамического массива во всех функциях с `int` на `Data`. Теперь вы в любой момент сможете изменить тип элементов стека, изменив лишь одну строчку.

Часть 4: Заголовочные файлы

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.

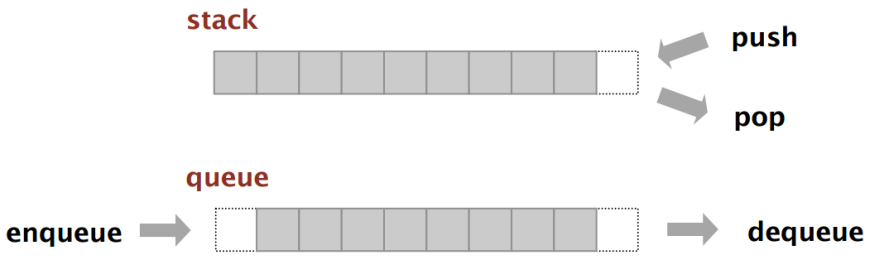
Стек (Stack, не путайте с сегментом памяти под таким же названием!) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `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 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:

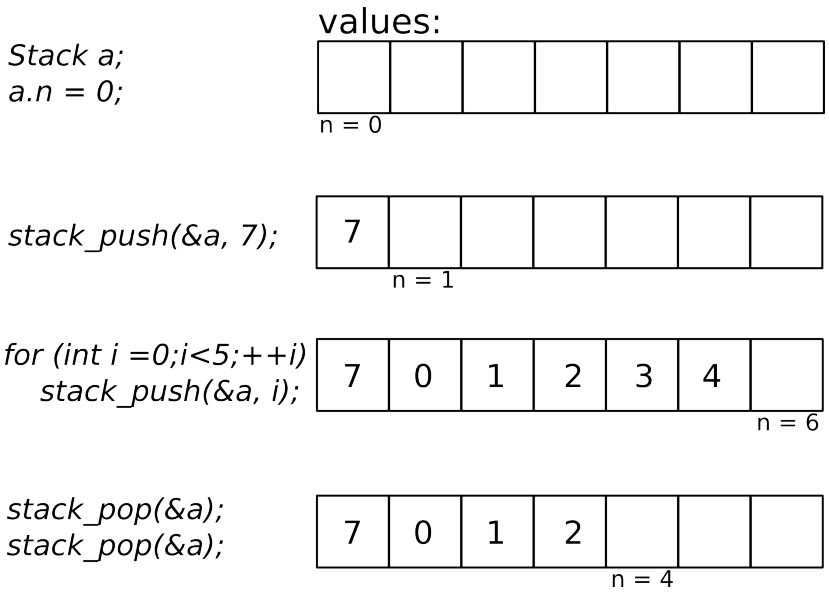
- `push_back` - добавить элемент в конец.
- `push_front` - добавить элемент в начало.
- `pop_back` - извлечь элемент с конца.
- `pop_front` - извлечь элемент с начала.

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 динамического массива



Задачи:

- Напишите следующие функции:
  1. `void stack_push(Stack* s, Data x)` – добавляет элемент в стек.
  2. `Data stack_get(const Stack* s)` – возвращает элемент, находящийся в вершине стека, но не изменяет стек.
  3. `void stack_pop(Stack* s)` – удаляет элемент, находящийся в вершине стека.
  4. `int stack_is_empty(const Stack* s)` – возвращает 1 если стек пуст и 0 иначе.
  5. `void stack_print(const Stack* s)` – распечатывает все элементы стека.
- **Скобочки**: Написать программу которая будет считывать последовательность скобочек и печатать `Yes` или `No` в зависимости от того является ли эта последовательность допустимой. Для считывания строки: `scanf("%s", str);`

вход	выход
`()`	Yes
`{[()]}`	Yes
`)))))`	No
`([)]`	No
`[{}()]`	Yes

вход	выход
`)(`	No
`(((((((())))))))`	Yes
`{{{{`	No
`{[([]()[{}])][()]}`	Yes
`]`	No

- **Следующий больший**: На вход поступает последовательность чисел. Нужно найти, для каждого элемента, индекс первого элемента, который следует после данного и является больше данного. Если такого элемента нет, то нужно напечатать `-1`.

вход	выход
10 1 5 2 4 6 9 1 8 7 3	1 4 3 4 5 -1 7 -1 -1 -1
5 1 2 3 4 5	1 2 3 4 -1
2 2 1	-1 -1