Семинар #6: Динамический массив.

Разные варианты массивов в языке C

Статический массив

Статический массив – это массив, размер которого фиксирован. В языке C такой массив создается так:

```
int a[3] = {10, 20, 30};
```

У такого статического массива есть 2 проблемы:

- Нельзя поменять размер, то есть нельзя добавить или удалить элемент.
- Он выделяется на стеке и его максимальный размер сильно ограничен.

Массив на стеке с переменным размером, но фиксированной вместимостью

Первую проблему можно частично решить, если создать массив больше, чем нужно на данный момент:

```
int a[100] = {10, 20, 30};
size_t size = 3;
```

В этом примере мы создали статический массив, который может хранить 100 элементов, но в данный момент используем только первые 3 элемента. Чтобы помнить, сколько элементов используется в данный момент мы завели переменную `size`. Введём следующие определения:

- *Размер массива (англ. size)* – количество элементов массива, которые доступны для использования.
- *Вместимость массива (англ. capacity)* – количество элементов, под которые в массиве выделена память.

То есть, для массива из примера выше размер равен 3, а вместимость равна 100. В такой массив мы можем добавлять элементы, но только до тех пор пока размер меньше, чем вместимость. Например, добавить новый элемент в конец массива `a` можно так (но только если `size < 100`):

```
a[size] = 60;
size += 1;
```

Несмотря на то, что в такой массив можно добавлять и удалять элементы, у такого подхода также есть недостатки. Вместимость не может меняться во время выполнения программы. Её нужно указать заранее. Если указать слишком маленькую вместимость, то этого может не хватить, а если указать слишком большую, то будет напрасно потрачено слишком много памяти. К тому же этот массив всё так же создаётся на стеке, поэтому его размер ограничен.

Выделение динамического массива в куче

Динамический массив – это массив, размер и вместимость которого может меняться во время выполнения программы. Такой массив, можно создать, выделив память в куче:

```
int* p = (int*)malloc(sizeof(int) * 3);
```

Указатель `p` указывает на массив из пяти элементов, созданный в куче. После того, как такой массив был создан, можно изменить его размер. Для этого нужно сделать следующее:

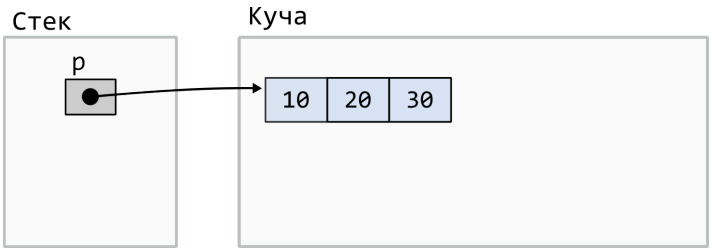
1. Выделить в куче ещё один участок памяти под новый массив большего размера.
2. Скопировать данные из старого массива в новый.
3. Освободить память старого массива.

Процесс увеличения размера массива, созданного в куче

Пусть изначально у нас есть массив из трёх элементов, созданный в куче. Попытаемся добавить ещё один элемент в такой массив.

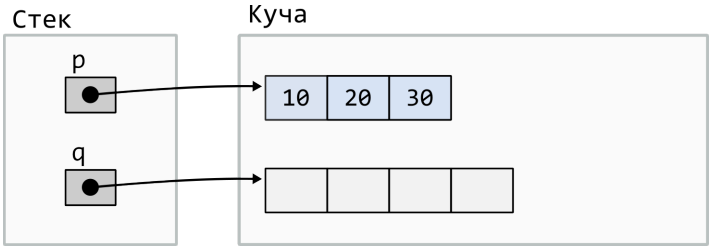
1) Исходное положение

```
int* p = (int*)malloc(sizeof(int)*3);
p[0] = 10;
p[1] = 20;
p[3] = 30;
```



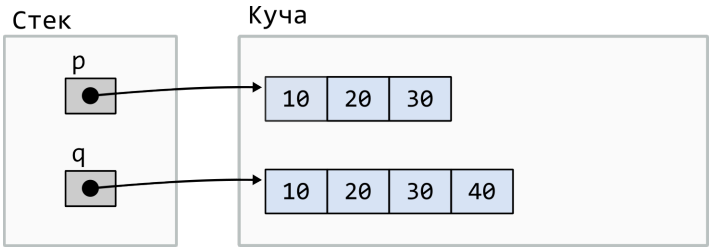
2) Выделяем новый массив большей памяти

```
int* q = (int*)malloc(sizeof(int)*4);
```



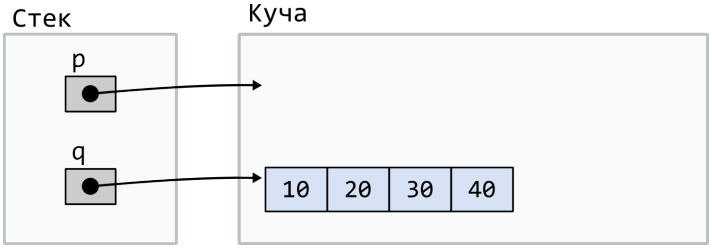
3) Копируем элементы из старого массива и добавляем новый элемент

```
for (size_t i = 0; i < 3; ++i)
    q[i] = p[i];
q[3] = 40;
```



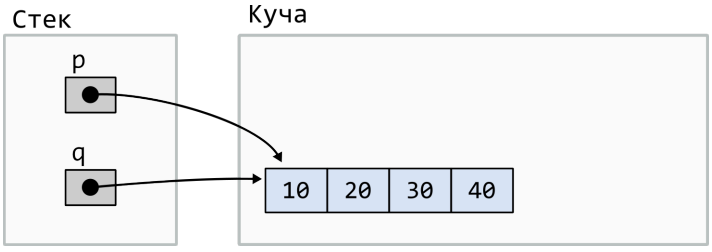
4) Освобождаем память старого массива

```
free(p);
```



5) Изменяем значение старого указателя

```
p = q;
```



Такой подход позволяет создавать массивы с изменяемым размером, не ограниченным размером стека. Однако, и этого подхода есть недостатки:

- Медленное добавление элементов. Для того, чтобы добавить один элемент в массив, нам пришлось вызвать `malloc`, так же скопировать весь массив.
- Придётся делать все операции по выделению/освобождению памяти и копированию массива каждый раз, когда нужно изменить размер. Это неудобно.

Создаём свой динамический массив

В отличии от многих других языков, в языке C нет удобного динамического массива, поэтому нам придётся написать свой. Попробуем написать наш массив так, чтобы он удовлетворял следующим требованиям:

1. В массив должно быть возможным добавление и удаление элементов. Его размер может меняться во время выполнения программы.
2. Размер массива не должен быть ограничен размером стека.
3. Массив должен быстро работать. Добавление и удаление элементов в конец массива должно работать за $O(1)$ в среднем.
4. Массив не должен занимать слишком много памяти. А именно, общее количество выделенной для массива памяти должно быть не более чем в 2 раза превышать суммарный размер всех элементов массива.
5. С нашим массивом должно быть удобно работать.
6. Тип хранимого элемента массива должен быть настраиваемым.

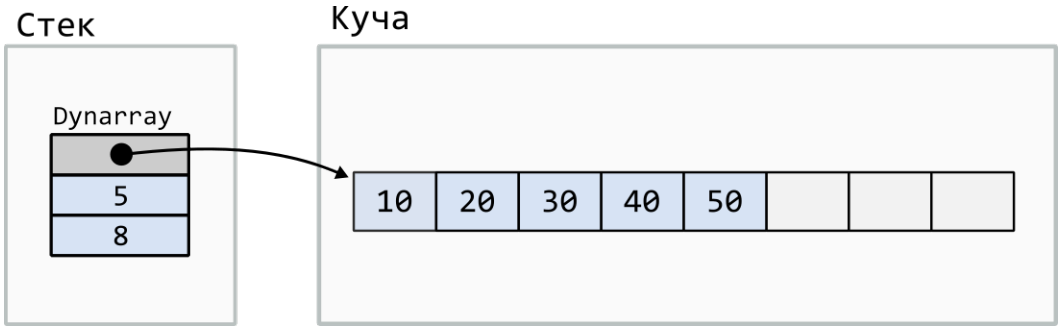
Структура для динамического массива будет выглядить следующим образом:

```
struct dynarray
{
    int* data;
    size_t size;
    size_t capacity;
};
typedef struct dynarray Dynarray;
```

Поля этой структуры:

- `data` – указатель на элементы массива, которые выделяются в куче.
- `size` – текущий размер массива. Столько элементов содержится в массиве.
- `capacity` - текущая вместимость массива. Под столько элементов в массиве выделена память. В отличии от статического массива эта величина может меняться. Количество памяти, выделенной в куче, будет равна `capacity * sizeof(int)`.

В памяти динамический массив с размером 5 и вместимостью 8 будет выглядеть следующим образом:



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.

Заголовочные файлы

Абстрактные типы данных: Стек и Очередь

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

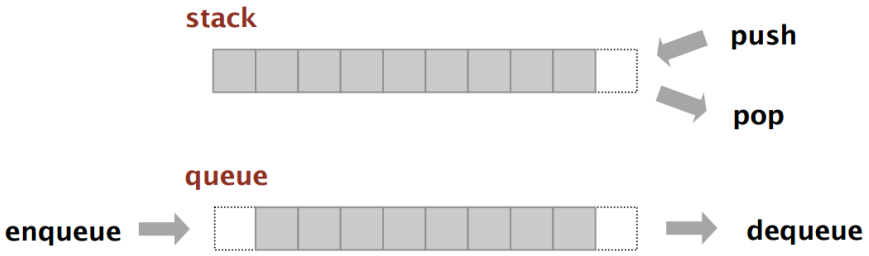
Стек (Stack, не путайте с сегментом памяти под таким же названием!) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `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:

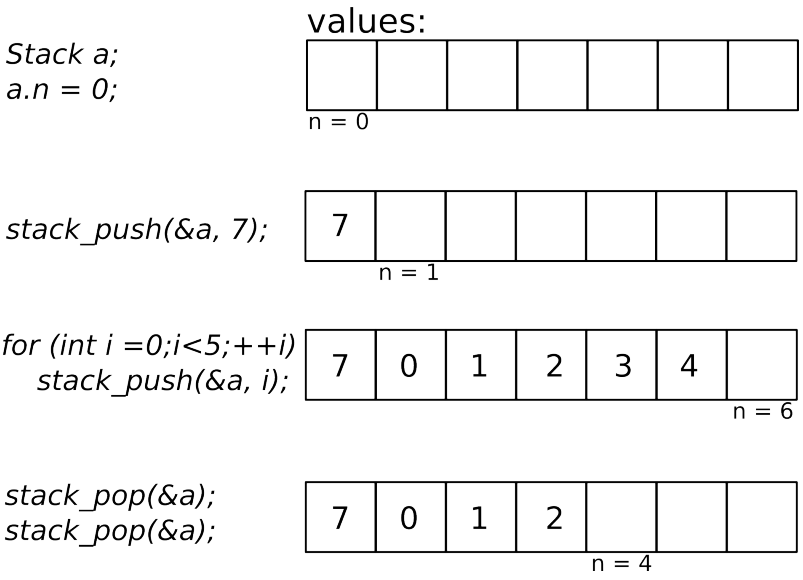
- `push_back` - добавить элемент в конец.
- `push_front` - добавить элемент в начало.
- `pop_back` - извлечь элемент с конца.
- `pop_front` - извлечь элемент с начала.

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).

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