1. Описание условия задачи.

Создать программу работы со стеком, выполняющую операции добавления, удаления элементов и вывод текущего состояния стека. Реализовать стек статическим массивом и списком. Все стандартные операции со стеком должны быть оформлены подпрограммами.

При реализации стека списком в вывод текущего состояния стека добавить просмотр адресов элементов стека и создать свой список или массив свободных областей с выводом его на экран.

При реализации стека массивом располагать два стека в одном массиве. Один стек располагается в начале массива и растет к концу, а другой располагается в конце массива и растет к началу. Заполнять и освобождать стеки произвольным образом с экрана. Элементами стека являются вещественные числа.

2. Техническое задание.

Исходные данные и правила их ввода.

- Номер пункта меню: целое число от 1 до 12.
- Число для добавления в стек: целое число от -2^{31} до $2^{31} - 1$.
- Число-результат удаления из стека: целое число от -2^{31} до $2^{31} - 1$.

Способ обращения к программе:

Запуск исполняемого файла. В рабочей директории выполнить команду ./app.exe.

Аварийные ситуации:

• Ввод некорректных данных.

• Ошибка выделения памяти.

В случае аварийной ситуации выводится сообщение об ошибке.

3. Описание внутренних структур данных.

Структура для хранения двух стеков в двух массивов:

```
typedef struct bi_stack {
      double data[BI_STACK_CAP];
      size_t size;
      ssize_t l_pointer;
      ssize_t r_pointer;
} bi_stack_t;
```

Структура содержит 4 поля:

- `data` - данные стека - статический массив типа `double`.
- `size` - размер доступной области массива - беззнаковое целое числа типа `size_t`.
- `l_pointer` - индекс головы левого стека - целое числа типа ssize_t.
- `r_pointer` - индекс головы правого стека - целое числа типа ssize_t.

Структура узла списка:

```
struct node {
      double data;
      node_t *next;
};
```

Структура содержит 2 поля:

- `double` - число, хранящееся в узле списка - вещественное число типа `double`.
- `node_t` - указатель на следующий узел.

`typedef node_t* list_t;` - Список - синоним типа указатель на узел спика.

Стек на списке реализован как АТД.

Структура стека, реализованного при помощи списка:

```
struct stack {
      list_t *data;
};
```

Структура содержит 1 поле - data - список.

Структура данных исследования:

```
typedef struct {
      size_t pushes_pops_count;
      double average_time_bi_stack_l_push;
      double average_time_bi_stack_l_pop;
      double average_time_bi_stack_r_push;
      double average_time_bi_stack_r_pop;
      size_t average_size_bi_stack;
      double average_time_list_stack_push;
      double average_time_list_stack_pop;
      size_t average_size_list_stack;
} research_t;
```

Структура содержит 9 полей:

- `pushes_pops_count` - количество произведеённых вставок и удалений из стека - беззнаковое целое числа типа `size_t`.
- `average_time_bi_stack_l_push` - среднее время добавления в левый стек на статическом массиве - вещественное число типа `double`.
- `average_time_bi_stack_l_pop` - среднее время удаления из левого стека на статическом массиве - вещественное число типа `double`.
- `average_time_bi_stack_r_push` - среднее время добавления в правый стек на статическом массиве - вещественное число типа `double`.
- `average_time_bi_stack_r_pop` - среднее время удаления из правого стека на статическом массиве - вещественное число типа `double`.
- `average_size_bi_stack` - объем памяти, занимаемый стеком на статическом массиве - беззнаковое целое числа типа `size_t`.
- `average_time_list_stack_push` - среднее время добавления в стек на списке - вещественное число типа `double`.
- `average_time_list_stack_pop` - среднее время удаления из стека на списке - вещественное число типа `double`.
- `average_size_list_stack` - средний объём памяти, занимаемый стеком на списке - беззнаковое целое числа типа `size_t`.

Для работы со структурами ииспользуются следующие функции:

Стек, реализованный на статическом массиве:
`void bi_stack_init(bi_stack_t *stack, size_t size);` - инициализация.

`int bi_stack_l_push(bi_stack_t *stack, double value);` - добавление в левый стек.

`int bi_stack_l_pop(int *dst, bi_stack_t *stack);` - удаление из левого стека.

`int bi_stack_r_push(bi_stack_t *stack, double value);` - добавление в правый стек.

`int bi_stack_r_pop(int *dst, bi_stack_t *stack);` - удаление из правого стека.

`void bi_stack_print(FILE *file, bi_stack_t *stack);` - вывести состояние стека.

Стек, реализованный на списке:

`list_t *list_new();` - создание стека.

`int list_push_back(list_t *list, double value);` - добавление в стек.

`int list_pop_back(int *value, list_t *list, int write_to_adresses_array);` - удаление из стека.

`void list_print(FILE *file,list_t *list);` - вывести состояние стека.

`void list_print_adresses(FILE *file, list_t *list);` - вывести адреса узлов стека.

`void list_delete(list_t *list);` - очистить память, занимаемую стеком.

Массив свободных адресов:

`int free_adresses_array_init(void);` - создать массив.

`int free_adresses_array_add(void *freed);` - добавить адрес в массив.

`void free_adresses_array_print(FILE *file);` - вывести массив.

`void free_adresses_array_delete();` - освободить память, занимаемую стеком.

4. Описание алгоритма.

Выводится меню программы. Пользователь вводит номер команды, после чего выполняется действие.

При добавлении в стек на статическом массиве производится проверка на переполнение. Если в стеке есть свободное место, указатель сдвигается на 1, после чего происходит запись. Коллизия указателей левого и правого стека не проверяется.

При удалении из стека на статическом массиве производится проверка на пустоту стека. Если стек не пуст, значение в ячейке запоминается, указатель сдвигается на 1, после чего значение возвращается. При данном удалении из стека, сами данные не удаляются из статического массива.

Так как в одном массиве расположены сразу два стека и отсутствует проверка коллизий, возможно изменение данных одного стека другим.

При добавлении в стек на списке, выделяется память под новый узел, который помещается в начало списка, становясь его головой. При невозможности выделить память под новый узел, возвращается код ошибки.

При удалении из стека на списке производится проверка на пустоту стека. Если стек не пуст, значение в голове списка запоминается, после чего память, выделенная под голову, очищается. Новой головой становится следующий после старой узел.

5. Оценка эффективности.

При проведении исследования по 10 раз производилось от 100’000 до 1’000’000 добавлений и удалений в стек, после вего бралось среднее значение времени добавления и удаления в каждый из стеков.

Время добавления:

	Время добавления, тики		
Количество операций	Левый стек	Правый стек	Список
10000	9.6e+01	8.4e+01	6.0e+02
15000	9.7e+01	8.6e+01	5.5e+02
20000	1.0e+02	8.9e+01	5.7e+02
25000	1.1e+02	9.7e+01	5.8e+02
30000	9.1e+01	8.8e+01	4.7e+02
35000	8.5e+01	8.2e+01	3.5e+02
40000	8.3e+01	8.2e+01	3.7e+02
45000	8.2e+01	8.2e+01	3.6e+02
50000	8.0e+01	8.0e+01	3.0e+02
55000	8.8e+01	8.6e+01	4.1e+02
60000	9.2e+01	9.1e+01	4.8e+02
65000	7.8e+01	7.8e+01	2.7e+02
70000	7.8e+01	7.9e+01	2.7e+02
75000	8.0e+01	8.0e+01	3.0e+02
80000	7.8e+01	7.8e+01	2.7e+02
85000	7.9e+01	7.9e+01	3.0e+02
90000	7.9e+01	7.8e+01	3.3e+02
95000	7.8e+01	7.8e+01	2.7e+02
100000	8.8e+01	9.1e+01	4.9e+02

Время удаления:

	Время удаления, тики		
Количество операций	Левый стек	Правый стек	Список
10000	7.7e+01	7.7e+01	4.1e+02
15000	7.9e+01	7.8e+01	4.1e+02
20000	8.5e+01	8.4e+01	4.5e+02
25000	8.9e+01	8.8e+01	4.4e+02
30000	8.3e+01	8.4e+01	3.6e+02
35000	7.9e+01	8.0e+01	2.4e+02
40000	7.9e+01	7.9e+01	3.0e+02
45000	7.9e+01	8.0e+01	2.8e+02
50000	7.8e+01	7.9e+01	2.1e+02
55000	8.2e+01	8.3e+01	3.1e+02
60000	8.4e+01	8.4e+01	3.8e+02
65000	7.7e+01	7.8e+01	2.0e+02
70000	7.8e+01	8.5e+01	2.5e+02
75000	7.7e+01	7.9e+01	2.1e+02
80000	7.7e+01	7.8e+01	2.0e+02
85000	7.6e+01	7.7e+01	2.5e+02
90000	7.6e+01	7.7e+01	2.4e+02
95000	7.8e+01	7.9e+01	2.2e+02
100000	8.2e+01	8.2e+01	3.5e+02

Занимаемая память:

	Размер, байты	
Количество операций	Массив	Список
10000	8,0E+05	1,6E+05
15000	8,0E+05	2,4E+05
20000	8,0E+05	3,2E+05
25000	8,0E+05	4,0E+05
30000	8,0E+05	4,8E+05
35000	8,0E+05	5,6E+05
40000	8,0E+05	6,4E+05
45000	8,0E+05	7,2E+05
50000	8,0E+05	8,0E+05
55000	8,0E+05	8,8E+05
60000	8,0E+05	9,6E+05
65000	8,0E+05	1,0E+06
70000	8,0E+05	1,1E+06
75000	8,0E+05	1,2E+06
80000	8,0E+05	1,3E+06
85000	8,0E+05	1,4E+06
90000	8,0E+05	1,4E+06
95000	8,0E+05	1,5E+06
100000	8,0E+05	1,6E+06

Вставка и удаление из стека на статическом массиве всегда работает быстрее, чем в стеке на списке. В стеке на списке удаление происходит быстрее, чем добавление, так как при добавлении под новый узел выделяется память.

Во время исследования создавался статический массив одинакового размера, поэтому занимаемая стеком на статическом массиве память не менялась. Память, выделяемая под стек на списке, растёт с увеличением количества узлов, поэтому до определённого момента стек на списке выигрывает по памяти, после чего начинает проигрывать.

6. Вывод.

Операции над стеком, реализованном на списке, работают на порядок медленнее, чем операции над стеком, реализованном на статическом массиве. Такая разница во времени обусловлена тем, что во время работы со списком происходят операции над динамической памятью, из-за чего затрачивается дополнительное время.

Однако стек, реализованный на статическом массиве, ограничен в размере.

Таким образом, реализовывать стек при помощи статического массива имеет смысл, если заранее известно максимальное количество элементов, которые могут быть добавлены в стек. Если это количество точно неизвестно, имеет смысл использовать список, жертвуя скоростью работы.

7. Контрольные вопросы.

1) Что такое стек?

Стек — это абстрактная структура данных, которая работает по принципу "последним вошел, первым вышел" (Last-In, First-Out, LIFO).

2) Каким образом и сколько памяти выделяется под хранение стека при различной его реализации?

При реализации статическим массивом, память выделяется при создании переменной стека и не меняется.

При реализации при помощи списка, память выделяется под каждый узел списка при каждом добавлении. Узел занимает больший объём памяти, так как в нем, помимо данных, хранится указатель на следующий узел.

3) Каким образом освобождается память при удалении элемента стека при различной реализации стека?

При реализации статическим массивом память не освобождается.

При реализации списком память освобождается при удалении элемента.

4) Что происходит с элементами стека при его просмотре?

Для просмотра верхнего элемента стека нужно произвести операцию удаления.

5) Каким образом эффективнее реализовывать стек? От чего это зависит?

При большом количестве элементов, которые нужно добавить в стек, эффективнее реализовывать стек на статическом массиве, так как такой стек будет занимать меньше памяти, а также операции над ним будут выполнятся быстрее.

Если заранее не известно количество элементов, выгоднее использовать стек, реализованный на списке, так как такой стек будет занимать меньше памяти.