Прежде чем мы начнём урок про составные типы данных, научимся выделять память ещё одним способом.

Куча (Heap)

Можно считать, что память под данные в C выделяется тремя способами:

- в стеке (почти все локальные переменные);
- в области глобальных данных (в основном глобальные переменные);
- в куче.

Куча это специальная область памяти, в которой можно резервировать блоки под текущие нужды. В отличие от стека, в куче можно попытаться выделить много памяти, и даже если это не получится, программа не сломается.

Чтобы запросить резервирование блока в куче мы пользуемся функцией `malloc` из стандартной библиотеки языка C. Она вернёт адрес начала выделенного участка памяти. Например:

```
/* Необходим заголовочный файл malloc.h */

#include <malloc.h>


void g() {


    // этот массив выделен в стеке

    int64_t marray[2] = { 10, 20 };


    // память под него освободится автоматически

}


void f() {


    // этот массив выделен в куче

    int64_t* marray = malloc( sizeof( int64_t ) * 2 );
```

```
    marray[0] = 10;

    marray[1] = 20;


    // память под указатель освободится автоматически

    // память под массив остаётся зарезервированной

}
```

Сигнатура функции `malloc` такая:

```
void* malloc( size_t size );
```

Функция принимает запрос на размер массива в байтах и возвращает указатель на начало выделенного массива. Если выделить память не получилось, то будет возвращён `NULL`.

В массиве, выделенном `malloc`, хранится мусор. Мы ничего не можем предполагать о значении его байтов, и он уж точно не будет заполнен нулями.

Если мы зарезервировали память с помощью `malloc`, она будет оставаться зарезервированной, пока программа не завершит свою работу. Если бесконтрольно выделять всё больше памяти с помощью `malloc`, то потребление программой памяти будет расти, пока вся доступная для кучи память не станет зарезервированой. Тогда:

- вырастет потребление памяти программой
- `malloc` не сможет больше выделять память

Нужно освобождать эту память вручную с помощью функции `free`.

```
void free( void* ptr );
```

Один из следующих вызовов `malloc` переиспользует память, освобождённую с помощью `free`.

Основные проблемы ручного управления памятью в куче такие:

- Не забыть вызвать `free` для каждого выделенного блока;
- Не вызвать `free` дважды для одного и того же блока.
- Не вызвать `free` до того, как данные действительно перестанут быть нужны (иначе мы затем обратимся к уже освобождённому блоку).

По возможности стоит избегать выделения памяти в куче если без него можно обойтись. Например, место под небольшие данные можно выделять в стеке, а затем передавать указатель на них в функции, которые с ними будут работать. Иногда можно использовать глобальные буферы, но функции не должны обращаться к ним напрямую: всегда лучше передавать указатель на буфер в функцию как аргумент.

В чём отличия выделения памяти с помощью `malloc` и выделения памяти в области локальных переменных?

1. Локальные переменные убиваются когда их функция завершает работу. Если мы этого не хотим, придётся использовать `malloc`. В следующем коде **распространённая ошибка**:

```
2. char* f() {
3.     char x[100];
4.
5.     ...
6.     // возвращаем адрес локальных данных
7.     // данные уничтожаются, при попытке использовать их адрес
8.     // программа может аварийно завершиться
9.     return x;
   }
```

   С другой стороны, память, выделенная в стеке, освобождается сама, за ней не нужно следить. А если не освободить память, выделенную в куче, после её использования, то она повиснет на программе мёртвым грузом.

10. Выделить большой объём данных в стеке нельзя – стек переполнится и программа аварийно завершит работу.
    Выделить большой кусок данных в куче и получить на него указатель можно всегда. Если памяти не хватит, то `malloc` вернёт `NULL`, можно это проверить и сообщить пользователю, что памяти не хватает.
11. В общем случае выделить память в стеке можно только зная заранее, сколько её нужно. Нельзя считать число `n` от пользователя и потом выделить в стеке ровно `n` байт.

Когда не стоит использовать кучу?

Если можно обойтись без использования кучи, нужно это делать. Помимо того, что память в куче надо освобождать вручную, куча работает значительно медленнее, чем стек. Более того, выделение памяти в куче может занимать очень разное количество времени в зависимости от её текущего состояния.

Не используйте кучу если нужно выделить буфер небольшого размера (скажем, пару килобайт). Мы можем даже не знать точно размер необходимого буфера, но если он не превышает фиксированного числа, можно выделить памяти в стеке с запасом.

В этой серии заданий вы реализуете считывание массива заранее неизвестной длины, и поиск в нём минимума.

Начнём со считывания массива.

- Первое число на входе – длина массива, затем идут его элементы.
  Например, пользователь ввёл числа 3 2 0 43. Это значит, что мы выделим массив на 3 элемента, в который запишем 2 0 43.
- Числа в массиве находятся в диапазоне [-2^{63}; 2^{63}-1][-2_{63};$2_{63}-1$]

Напоминаем, что:

- спецификаторы ввода начинаются на `SCN`, например, `SCNd64` для чисел `int64_t`;
- спецификаторы вывода начинаются на `PRI`, например, `PRId64` для чисел `int64_t`;
- для `size_t` спецификаторы ввода и вывода `%zu`.

1task – program

Теперь реализуйте функцию для поиска минимума в массиве. Аргументы функции --- адрес начала массива и количество элементов в нём.

Функция ничего не считывает со стандартного ввода.

Если минимального элемента в массиве нет, функция возвращает NULL.

Иначе она возвращает адрес любого из минимальных элементов в массиве.

В тестирующей системе ввод начинается с количества элементов в массиве, затем идёт сам массив, как и на предыдущем шаге;

Sample Input:

```
4 8 9 2 3
```

Sample Output:

```
2
```

2task – program

Наконец, реализуйте функцию `perform` которая считывает массив со входа, и выводит его минимальный элемент. Если массив пустой, выведите `None`.

Не забудьте освободить выделенную динамически память.

3task – program

Массив массивов

Теперь перейдём к другому практически важному случаю – массиву, содержащему указатели на другие массивы. Это разновидность *двумерного массива,* своеобразной таблицы, в которой каждая строчка может иметь разный размер.

Научимся считывать массив массивов чисел `int64_t`. Формат входа такой:

```
<число строк>

<количество в строке 1> <элемент11> <элемент12> ...

<количество в строке 2> <элемент21> <элемент22> ...
```

Для примера, пусть мы вводим следующие числа:

```
3

4 1 2 5 3

0

2 1 2
```

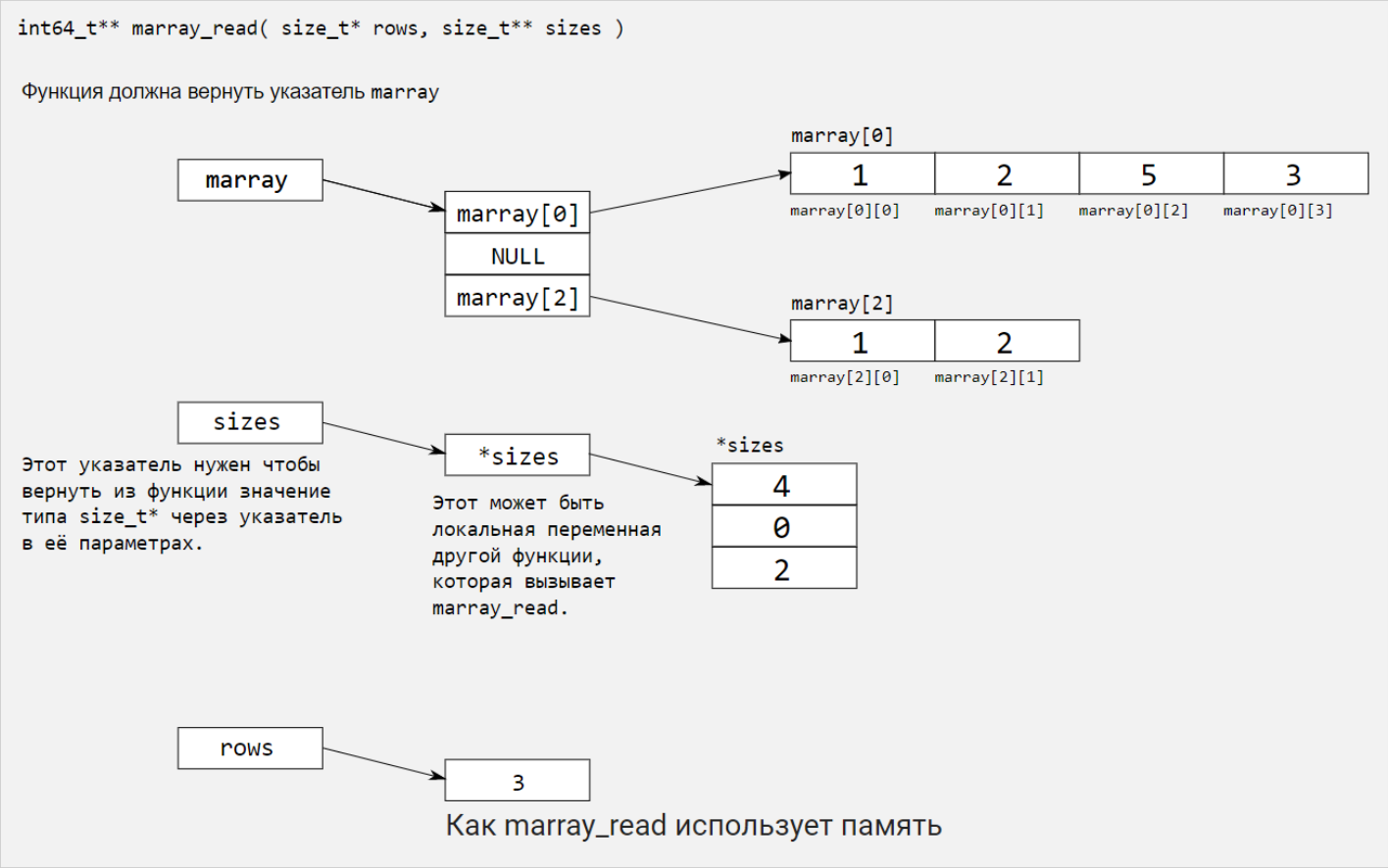
Тогда массив массивов можно изобразить следующим образом. Обратите внимание на пустую строчку, соответствующую пустому массиву.

```
1 2 5 3


1 2
```

Чтобы работать с массивом массивов придётся создать в памяти достаточно сложную структуру данных:

- выделить массив, хранящий адреса строчек;
- выделить память на каждую строчку;
- выделить память на отдельный массив `sizes`, который хранит длины строчек. Первый элемент массива `sizes` это длина первой строчки, второй – длина второй и т.д.



Нам пригодится функция из предыдущего задания, вы можете её вызывать.

```
int64_t* array_int_read( size_t* size );
```

Надо реализовать функцию marray_read(), которая:
- создает массив массивов и заполняет его
- создает массив размеров подмассивов и заполняет его
- возвращает ссылку на массив массивов
- возвращает ссылки на массив размеров и количество строк

Вывод массива уже реализован и использовать в задании функцию marray_print() не надо.

Sample Input:

```
3
9 3 2 4 54 9 2 1 872 123
8 123 12354 23 232 43412 534 8237 -99292
3 45 2 245
```

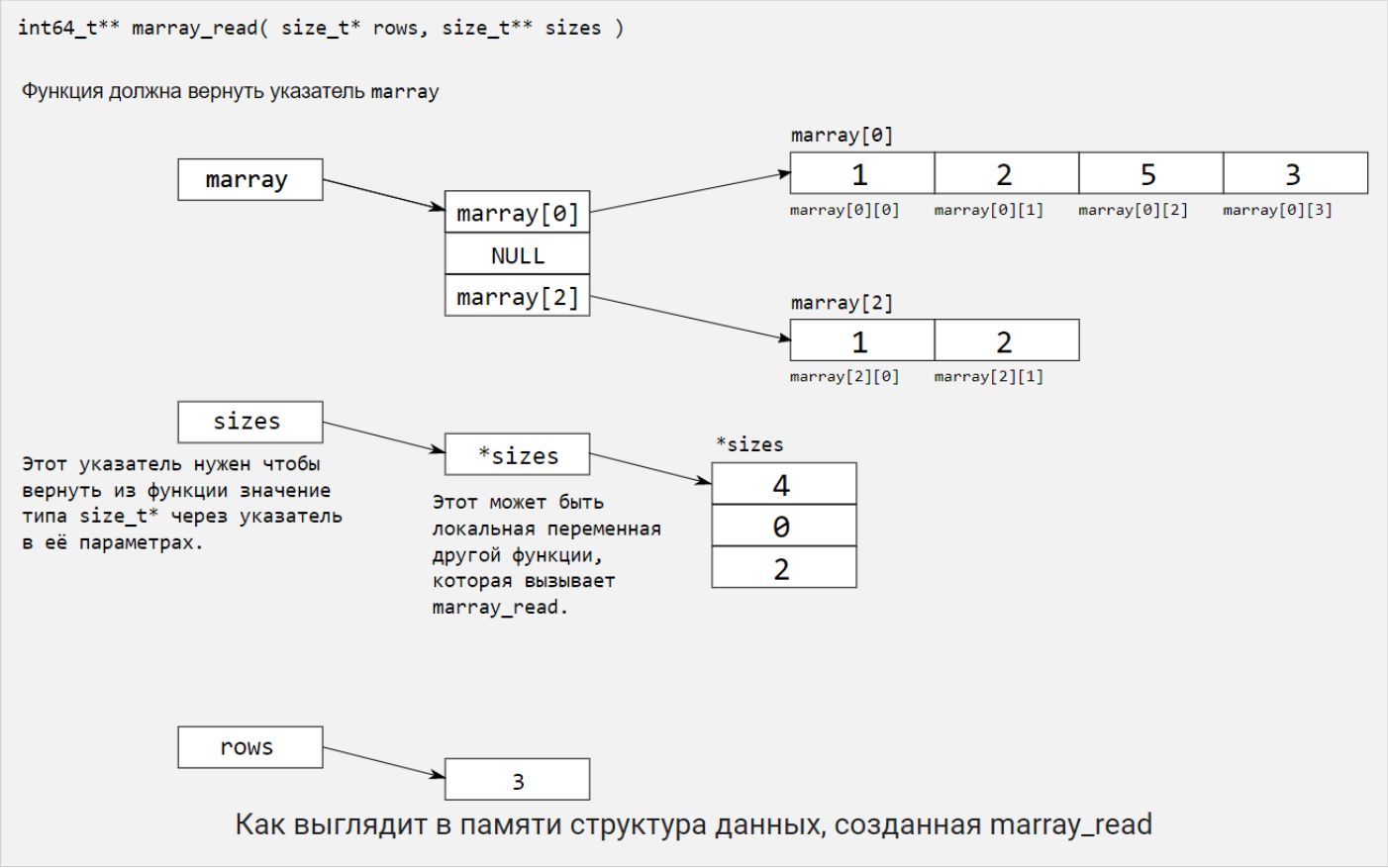
Sample Output:

```
3 2 4 54 9 2 1 872 123
123 12354 23 232 43412 534 8237 -99292
45 2 245
```

4task – program

Массив массивов нужно уметь освобождать. Недостаточно вызвать для него `free`, потому что каждая строка выделялась отдельно.

Реализуйте функцию для освобождения памяти под динамический массив массивов, такой, какой был показан на предыдущем слайде.



Как выглядит в памяти структура данных, созданная marray_read

Sample Input:

```
3
9 3 2 4 54 9 2 1 872 123
8 123 12354 23 232 43412 534 8237 -99292
3 45 2 245
```

Sample Output:

```
3 2 4 54 9 2 1 872 123
123 12354 23 232 43412 534 8237 -99292
45 2 245
```

5task – program

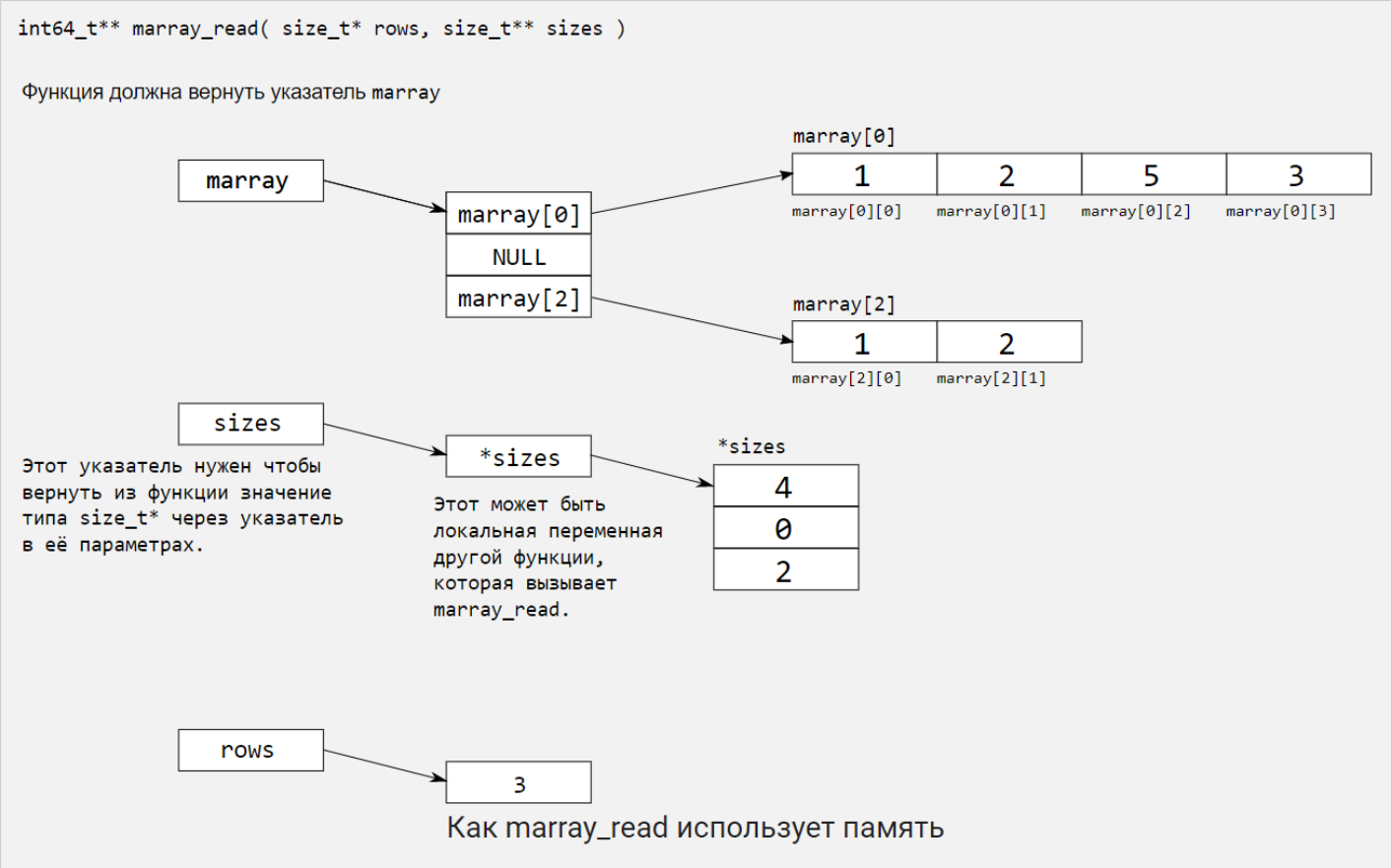
Наконец, сделаем что-нибудь интересное с массивом массивов.

1. Считаем массив массивов;
2. Найдем минимальный элемент *M* (один на весь массив массивов);
3. Вычтем *M* изо всех элементов массива массивов;
4. Выведем результат.

Вы можете использовать функции из предыдущих заданий.

Возможна ситуация, когда массив массивов состоит только из пустых строчек.

Для напоминания: массив массивов представляется в памяти для функции `marray_read` вот так:



6task.c – program

Начиная с C99 стандарт допускает возможность создавать в стеке массивы такой длины, которая заранее неизвестна:

```
void f() {

   size_t n = 0;


   // считываем число size_t с помощью спецификатора zu
```

```
    scanf("%zu", &n);

    int64_t array[n];

    ...
}
```

Эта возможность опциональна, то есть компилятор может её не реализовывать. Пожалуйста, не делайте так. Если массив окажется слишком большой, стек переполнится, и программа аварийно завершится. Нельзя попробовать выделить в стеке массив и проверить, получилось ли[1]: тут или успех, или программа аварийно завершается.

[1] Впрочем, и для массивов фиксированной длины нельзя это сделать. Однако когда длина массива известна заранее, мы по крайней мере точно можем оценить, сколько памяти нам потребуется. Мы точно можем предотвратить ситуации, когда на стеке выделяется массив размером в мегабайты, который не уместится в стеке.

Существуют также функции `calloc` и `realloc`

- `realloc` пытается выделить блок другого размера, скопировать туда содержимое старого блока (сколько получится) и деаллоцировать старый. Если выделить новый блок не получилось, старый не будет деаллоцирован.

```
void *realloc(void *ptr, size_t size);
```

- `calloc` выделяет память как `malloc`, но также записывает в неё нули. Размер выделенной памяти – произведение размера элемента `size` и количества элементов `nmemb`.

```
void *calloc(size_t nmemb, size_t size);
```

Имейте в виду, что запись нулей в память не всегда означает запись туда нулевых значений. К примеру, указатель в никуда `NULL` на некоторых архитектурах может кодироваться числом, в котором, наоборот, все биты равны единицам. Или число с плавающей точкой 0.0 типа `double` может представляться иначе, чем 8-байтовым числом, в котором все биты равны нулям. Поэтому обнуление памяти с помощью `calloc` (или, например, функции `memset`) в общем случае *не означает корректной инициализации данных нулевыми значениями*.