Массивы

Массив — это объединение некоторого количества переменных под одним идентификатором.

Отдельные переменные внутри массива различаются по уникальным номерам — целочисленным индексам.

Элементы массива расположены в памяти подряд, одним непрерывным участком.

Элементом массива может быть любой тип данных, в том числе — другой массив. Массив из массивов — это аналог двухмерной матрицы.

Массивы в языке C

Массив объявляется так же, как переменная, но после названия переменной добавляется количество элементов в квадратных скобках. Количество элементов должно быть константой, известной на момент компиляции программы.

```
int var;     // Одиночная переменная
int ar[100]; // Массив из 100 чисел
```

Индексы в массиве всегда начинаются с 0. В массиве из 100 элементов существуют индексы с 0 по 99 включительно.

Для обращения к элементу массива используется оператор квадратных скобок.

```
ar[0] = 0;
// ar[0] является самостоятельной переменной
printf("%d", ar[0]);
```

Статические и динамические массивы

Рассмотренные массивы являются **статическими**, так как их размер известен на этапе компиляции. Также применяются **динамические массивы**, создаваемые на этапе выполнения программы. Для использования динамических массивов необходимо объявить указатель, хранящий адрес первого по счету элемента массива.

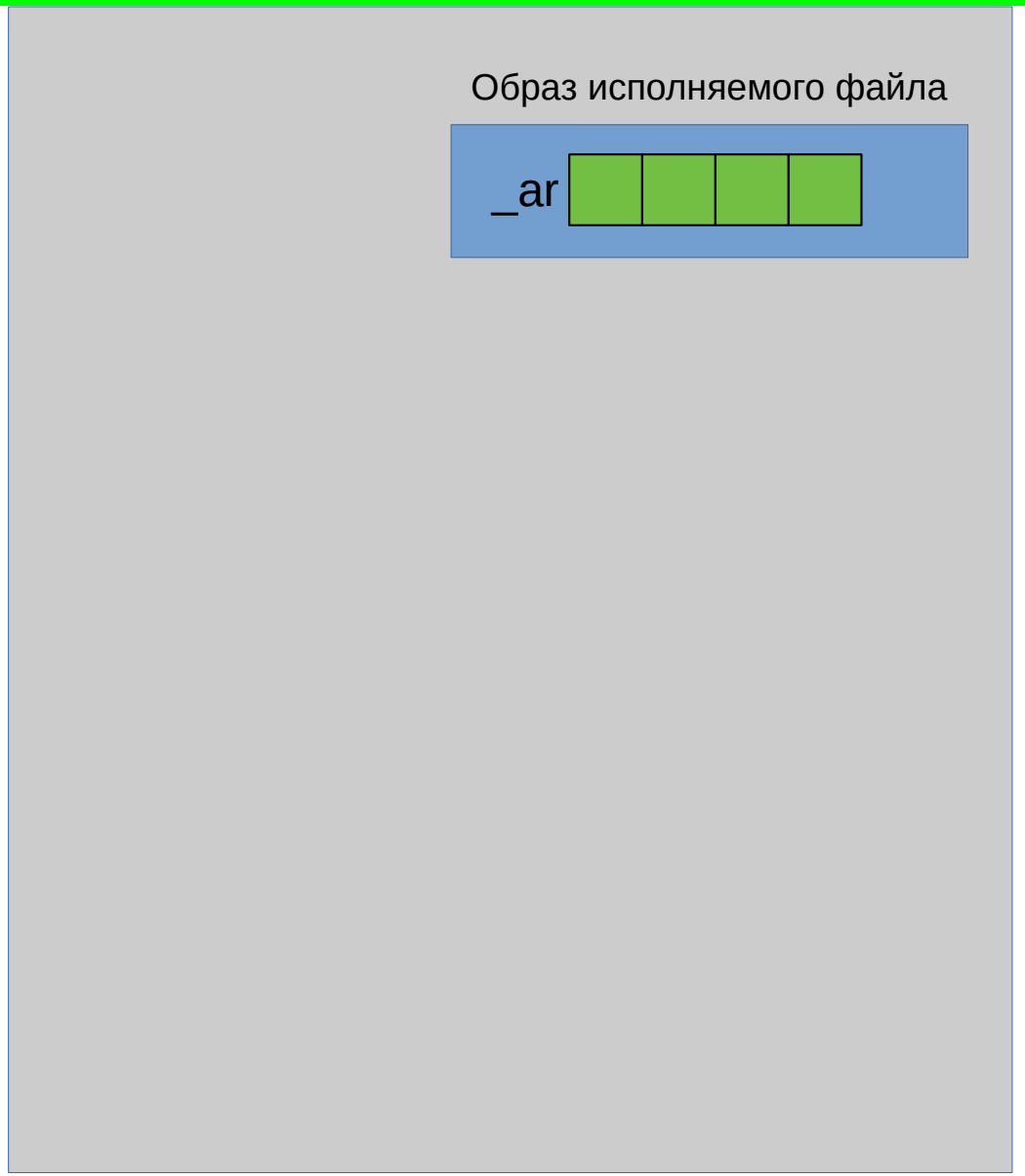
```
int* ar;
```

Для создания динамического массива в C необходимо использовать заголовочный файл **stdlib.h**. Функция **malloc** выделяет участок памяти, который используется для размещения массива, а функция **free** освобождает его. Работа с элементами динамического массива также идет при помощи квадратных скобок.

```
int* ar = (int*)malloc(100 * sizeof(int));
ar[0] = 0;
free(ar);
```

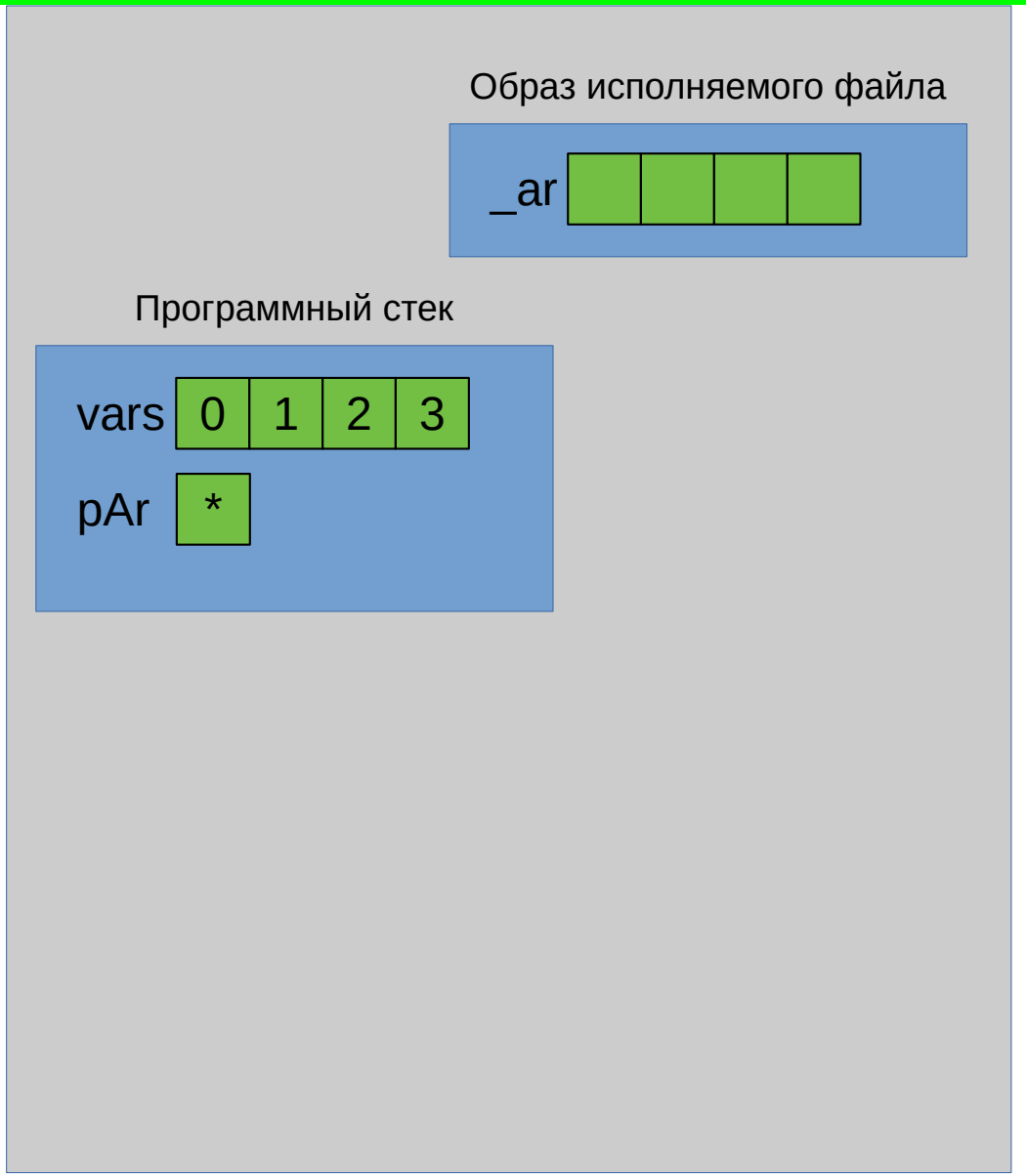
Статические и динамические массивы

```
#include <stdlib.h>
int _ar[4];
int main()
{
  // int vars[4];
  // int* pAr;
}
```



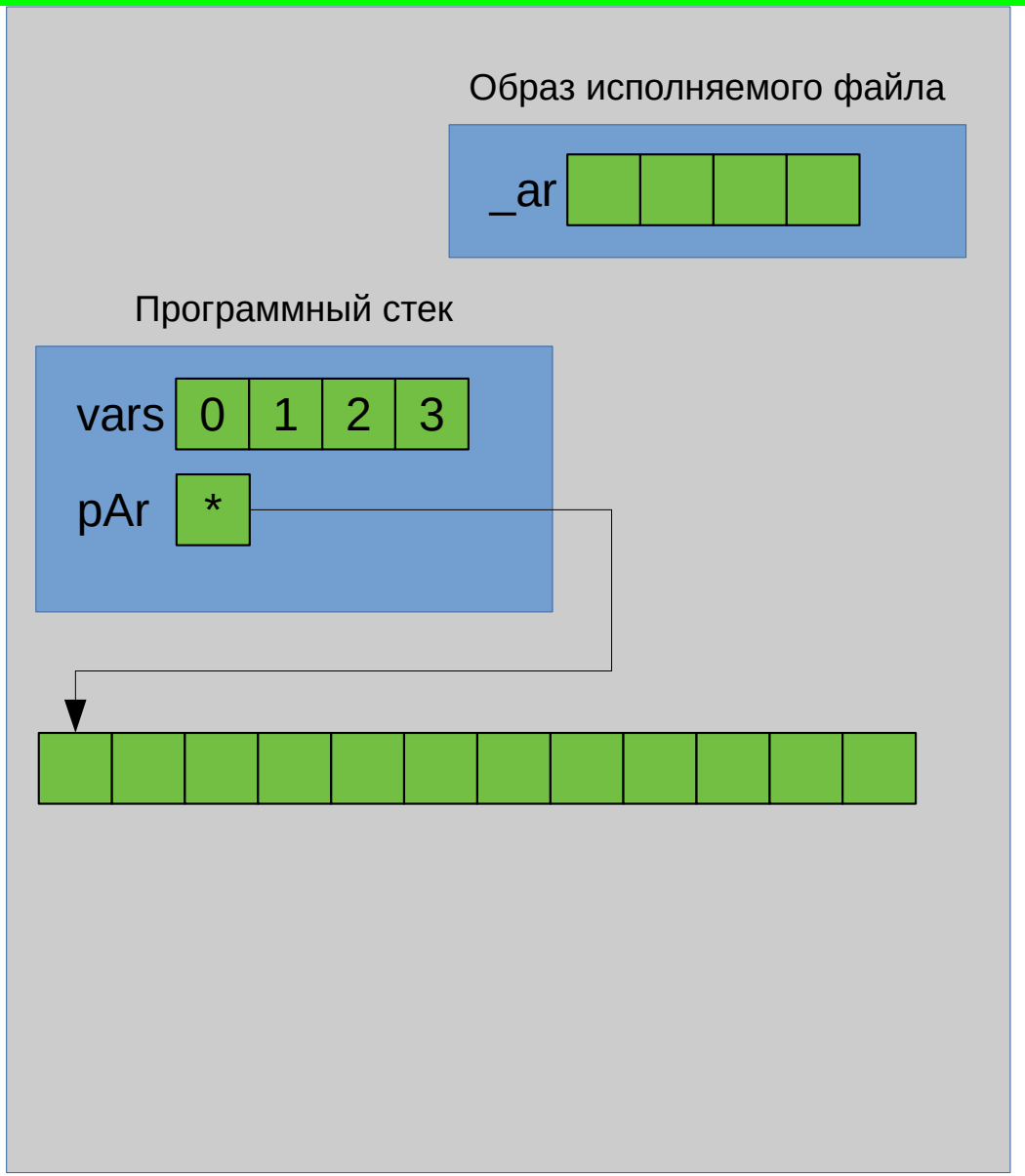
Статические и динамические массивы

```
#include <stdlib.h>
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
}
```



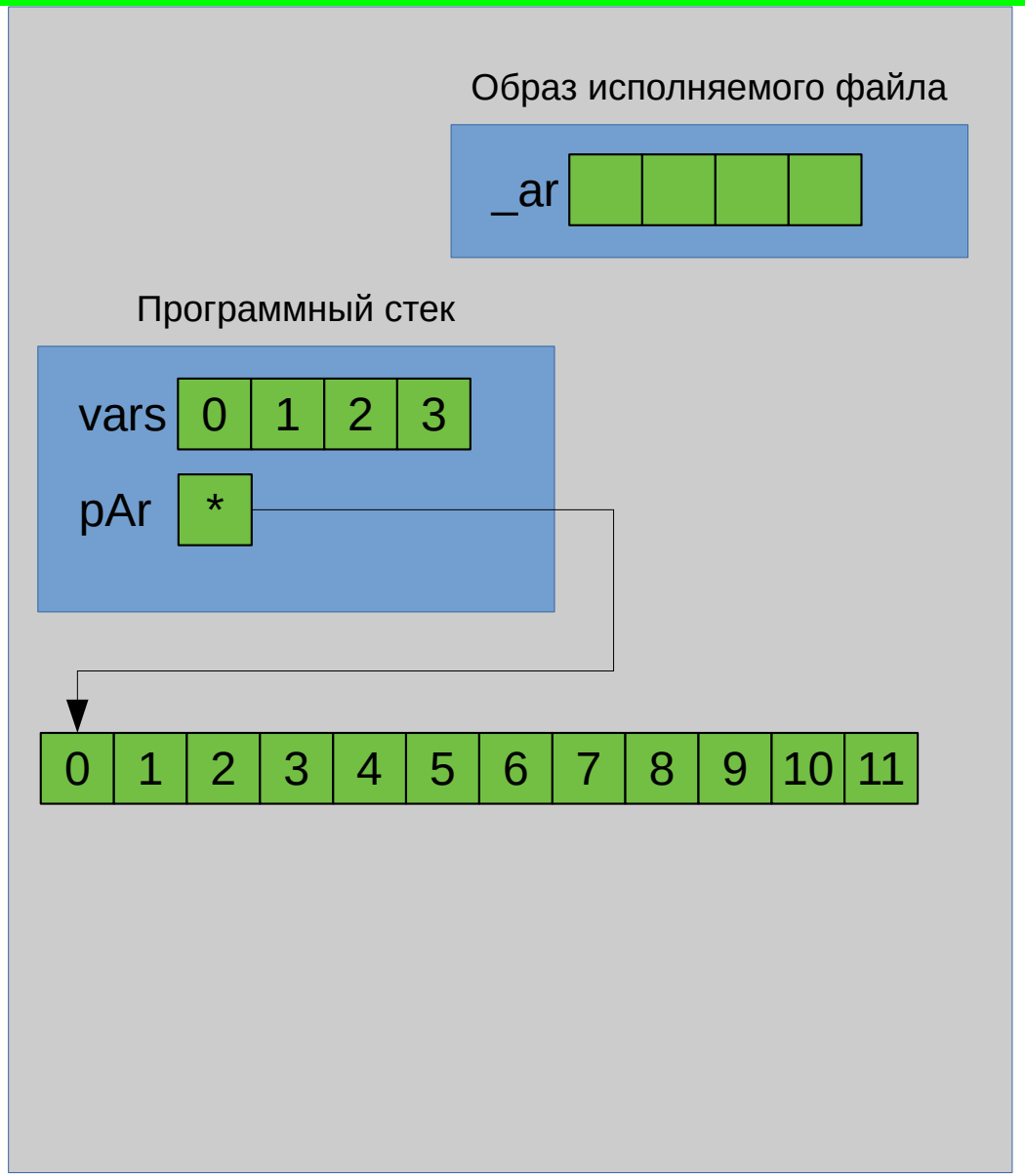
Статические и динамические массивы

```
#include <stdlib.h>
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
  pAr = (int*)malloc(
    12 * sizeof(int)
  );
}
```



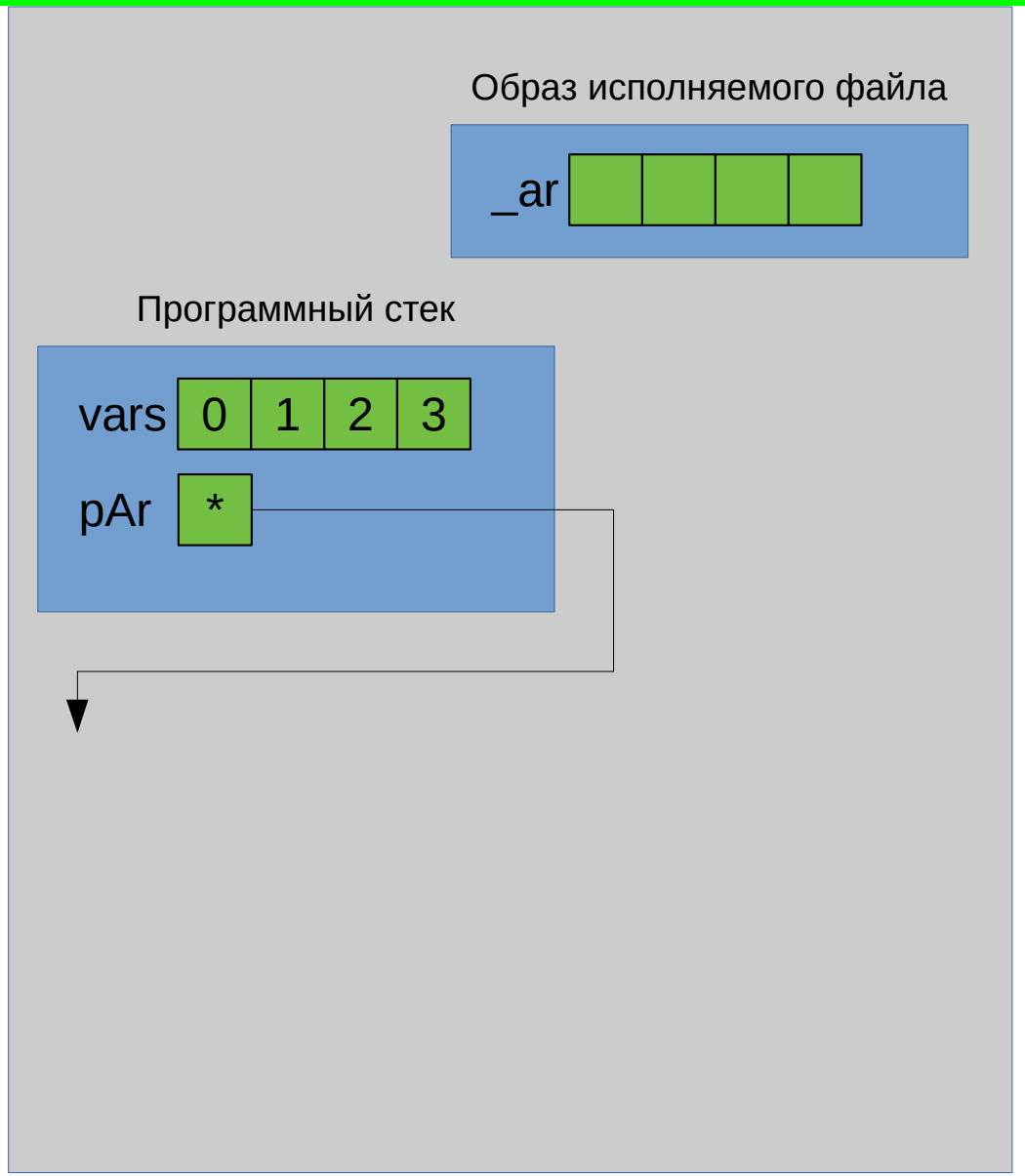
Статические и динамические массивы

```
#include <stdlib.h>
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
  pAr = (int*)malloc(
    12 * sizeof(int)
  );
  for (int a=0; a<12; a++)
    pAr[a] = a;
}
```



Статические и динамические массивы

```
#include <stdlib.h>
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
  pAr = (int*)malloc(
    12 * sizeof(int)
  );
  for (int a=0; a<12; a++)
    pAr[a] = a;
  free(pAr);
  // pAr[0] = 0; // !!!
}
```



Массивы в языке C++

Статические массивы в C++ полностью аналогичны массивам C. Для динамических массивов точно так же используются указатели. Однако для работы с динамической памятью используются операторы **new** и **delete**.

```
int* pVal = new int;
int* ar = new int[256];
*pVal = 4;                    // pVal[0] = 4;
for (int a=0; a<256; a++)
  ar[a] = a;
delete [] ar;
delete pVal;
```

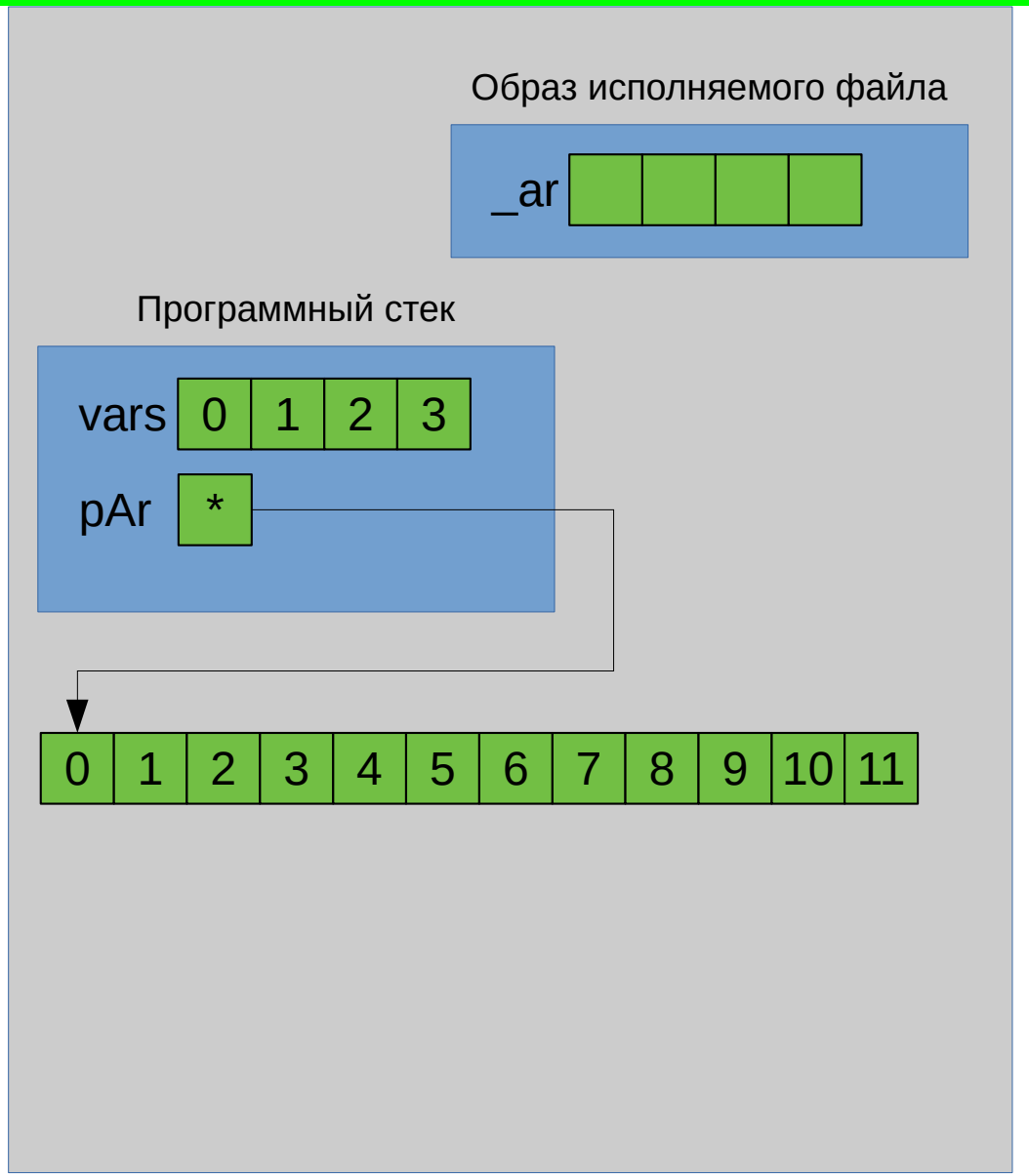
Оператор **new** вызывает конструктор для каждого создаваемого элемента. Оператор **delete** вызывает деструктор для каждого элемента. Если new был с **квадратными скобками**, delete также должен быть с квадратными скобками.

Массивы в языке C++

```
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
}
```

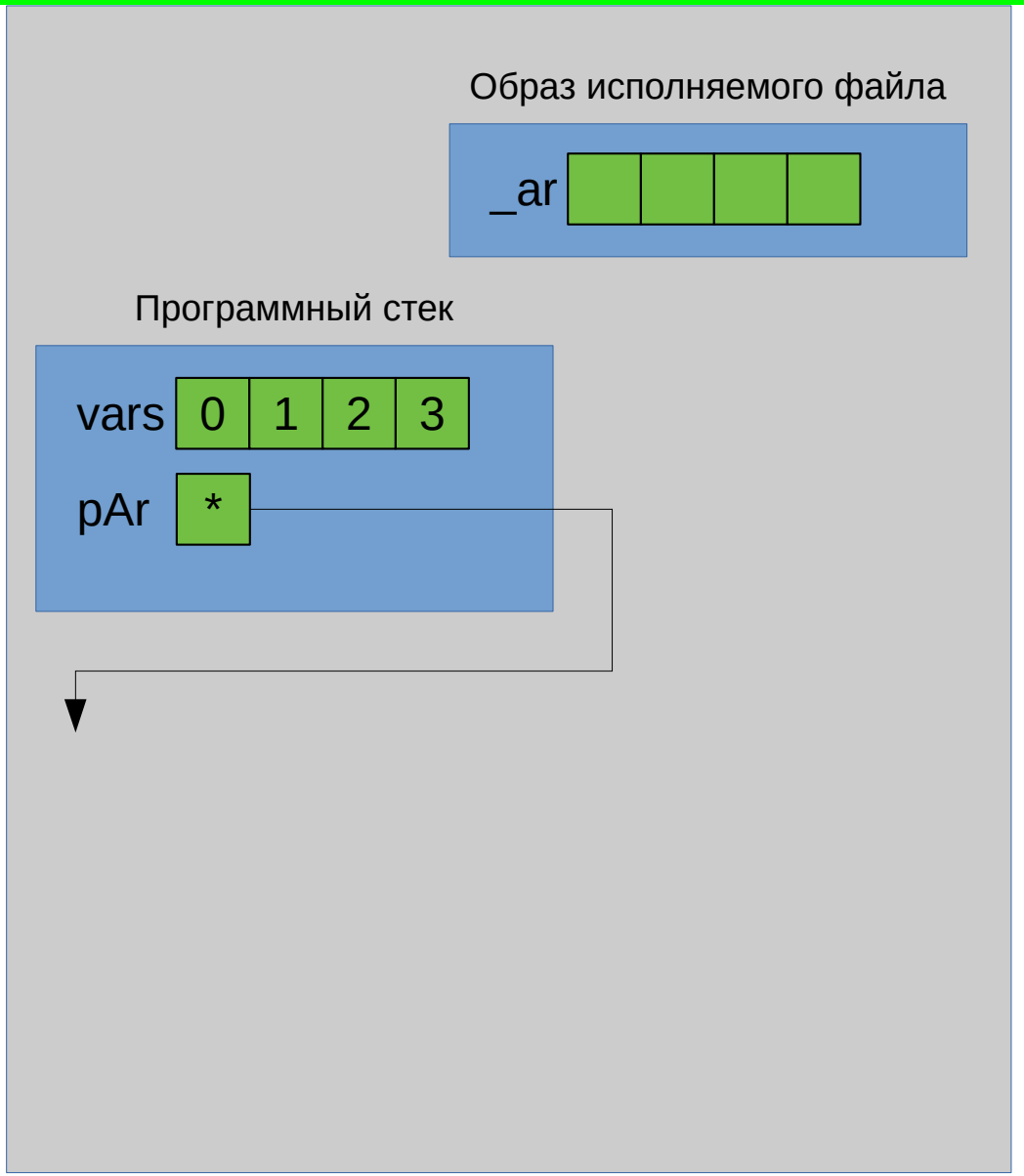
Массивы в языке C++

```
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
  pAr = new int[12];
  for (int a=0; a<12; a++)
    pAr[a] = a;
}
```



Массивы в языке C++

```
int _ar[4];
int main()
{
  int vars[4] =
    {0, 1, 2, 3};
  int* pAr;
  pAr = new int[12];
  for (int a=0; a<12; a++)
    pAr[a] = a;
  delete [] pAr;
  // pAr[0] = 0; // !!!
}
```



Алгоритм работы с массивом

Процессор не может работать с массивом целиком. Любая работа с массивом — это цикл, в котором последовательно, по отдельности обрабатываются все его элементы.

Пример иллюстрирует расчет скалярного произведения двух четырехмерных векторов, координаты которых записаны в массивах A и B.

```
double A[4] = {1, 0, 4, 2};
double B[4] = {0, 2, -5, 1};
double S = 0;
for (int a=0; a<4; a++)
  S = S + A[a] * B[a];
```

Двумерные массивы

Для создания статического двумерного массива необходимо создать массив из массивов. Для этого используются две пары квадратных скобок.

```
double R[3][4];
```

Доступ к элементам такого массива также осуществляется через пару квадратных скобок.

```
R[0][0] = 1;
```

Для обхода всех элементов такого массива необходимо записать **двойной** или **вложенный цикл**:

```
for (int a=0; a<3; a++)
  for (int b=0; b<4; b++)
    R[a][b] = 0;
```

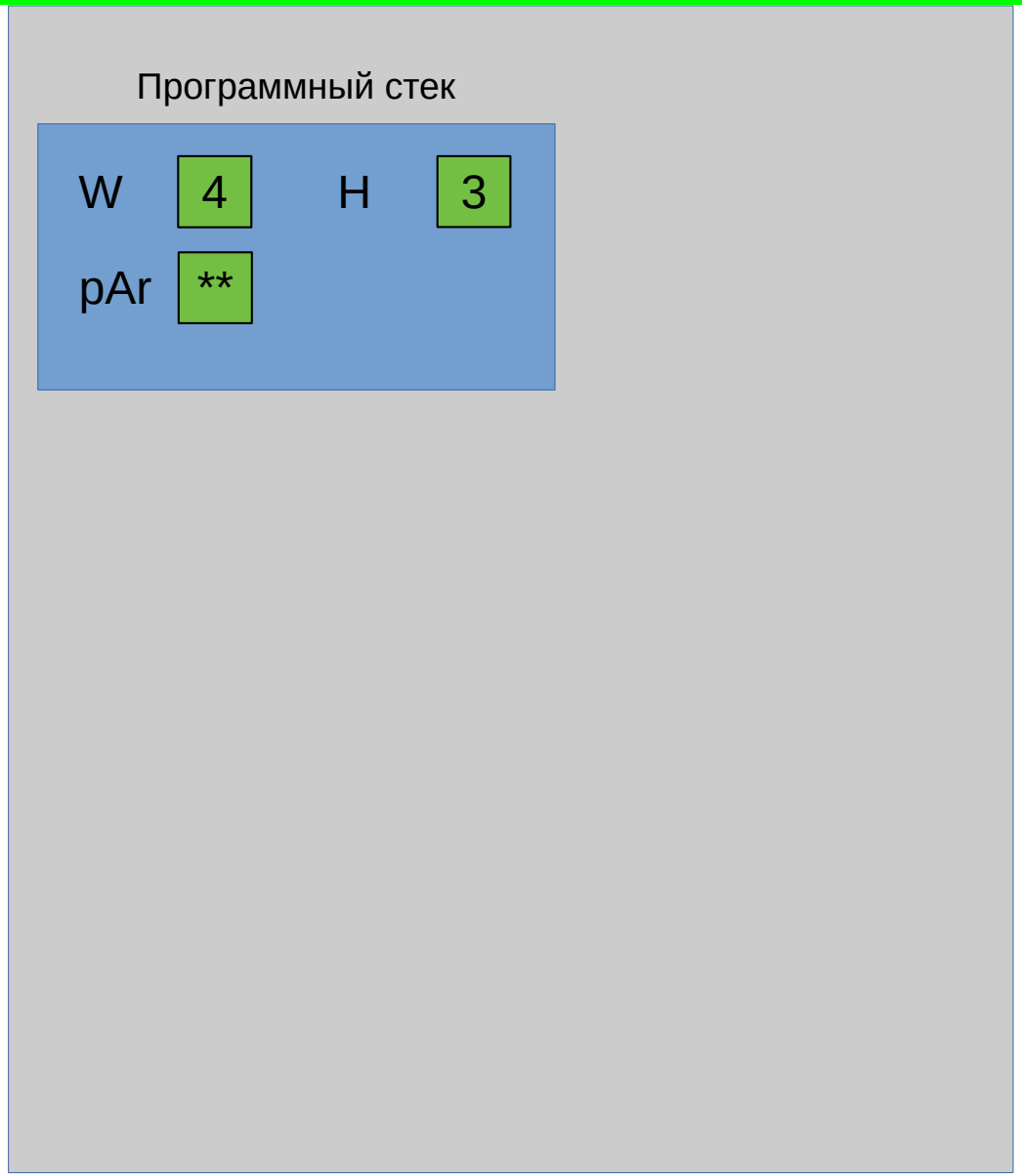
Двумерные массивы

Динамический двумерный массив можно сделать двумя способами:

- Массив из массивов: массив-столбец с указателями на массивы-строки. Удобен синтаксически, но в памяти расположен по частям.
- Непрерывный блок памяти с двумерной адресацией внутри. Используется для хранения растровых данных (изображений). Требует дополнительного пересчета координат в индекс массива.

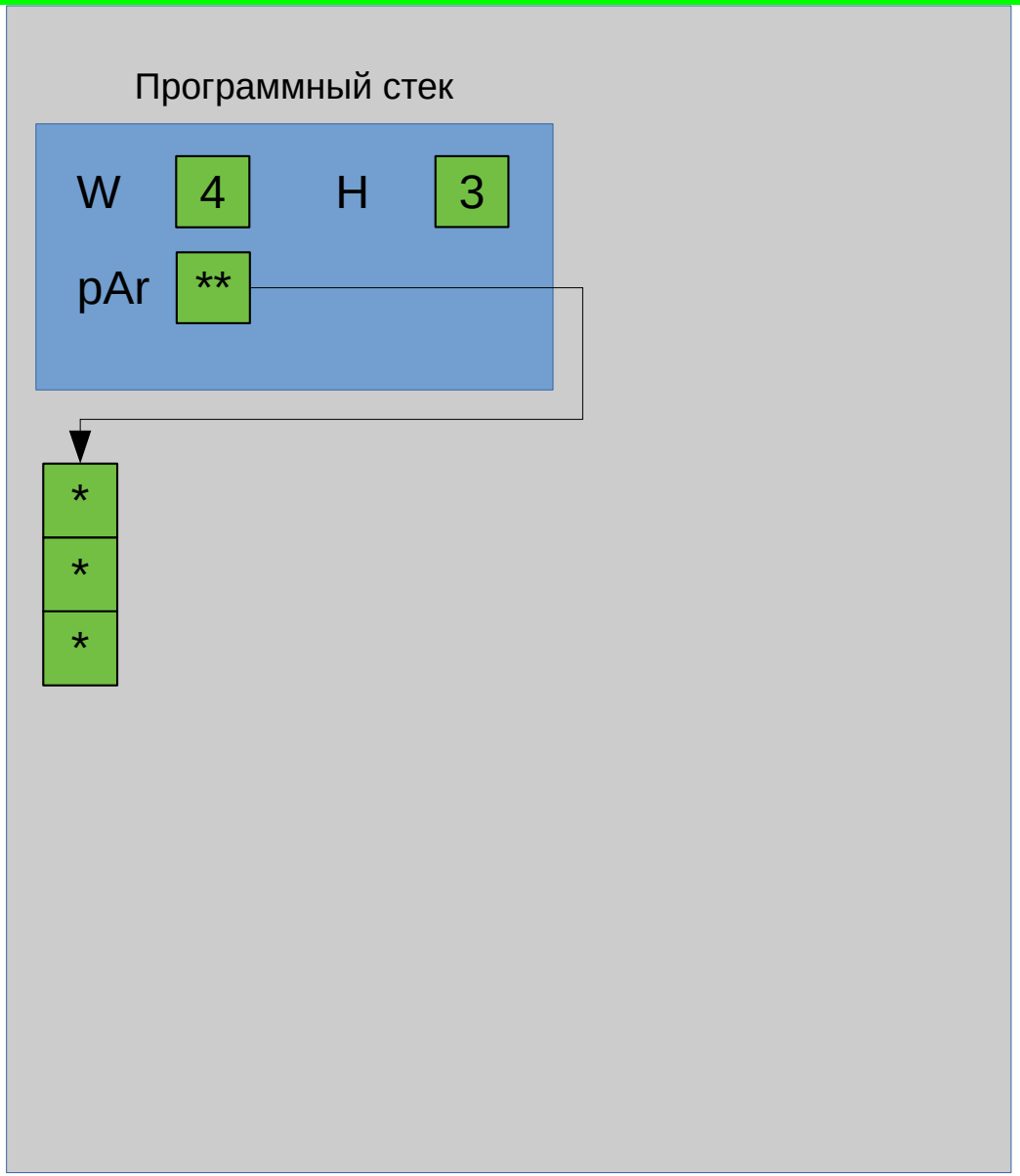
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** pAr;
}
```



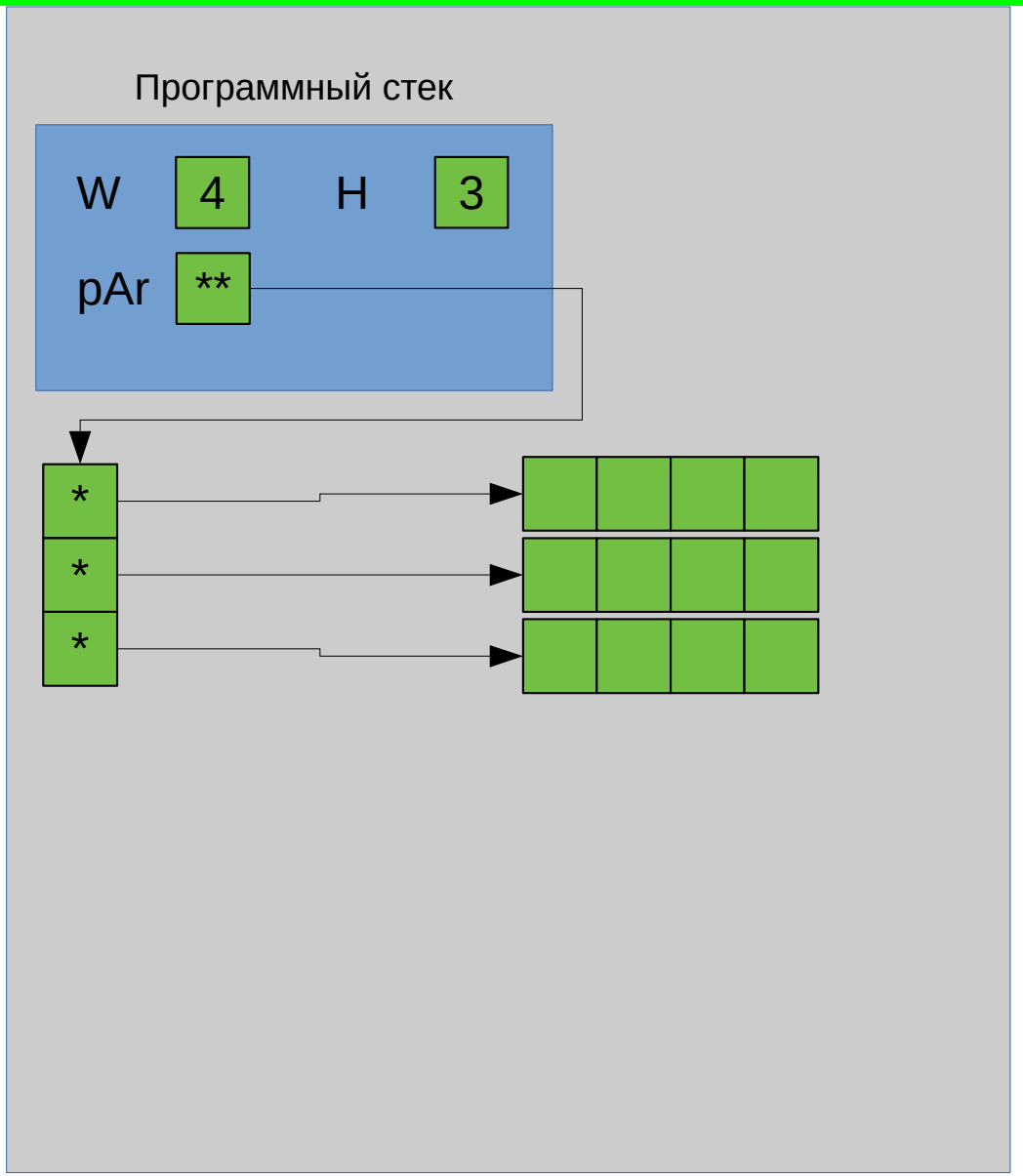
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** pAr;
  ar = new double*[H];
}
```



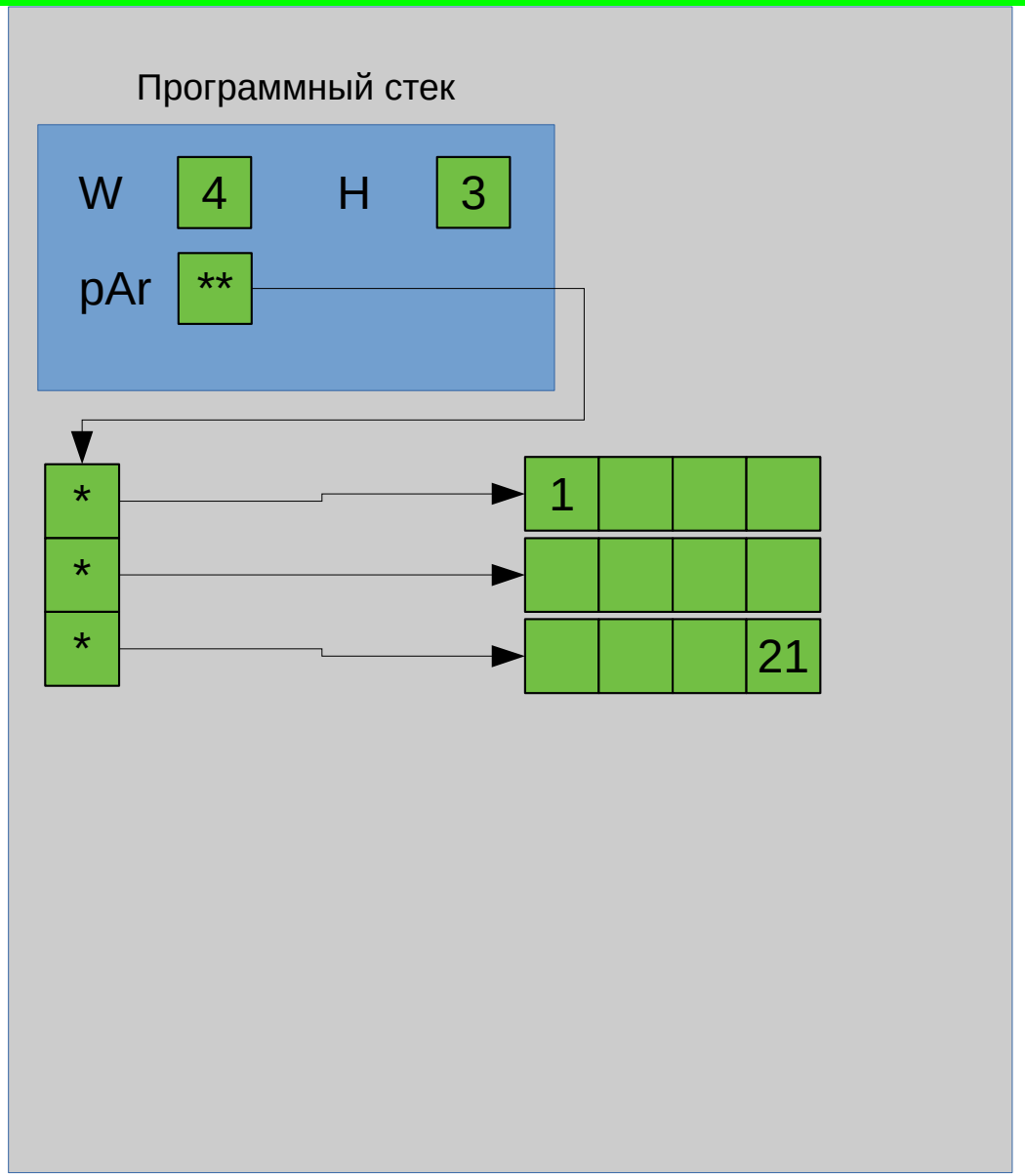
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** pAr;
  ar = new double*[H];
  for (int a=0; a<H; a++)
    pAr[a]=new double[W];
}
```



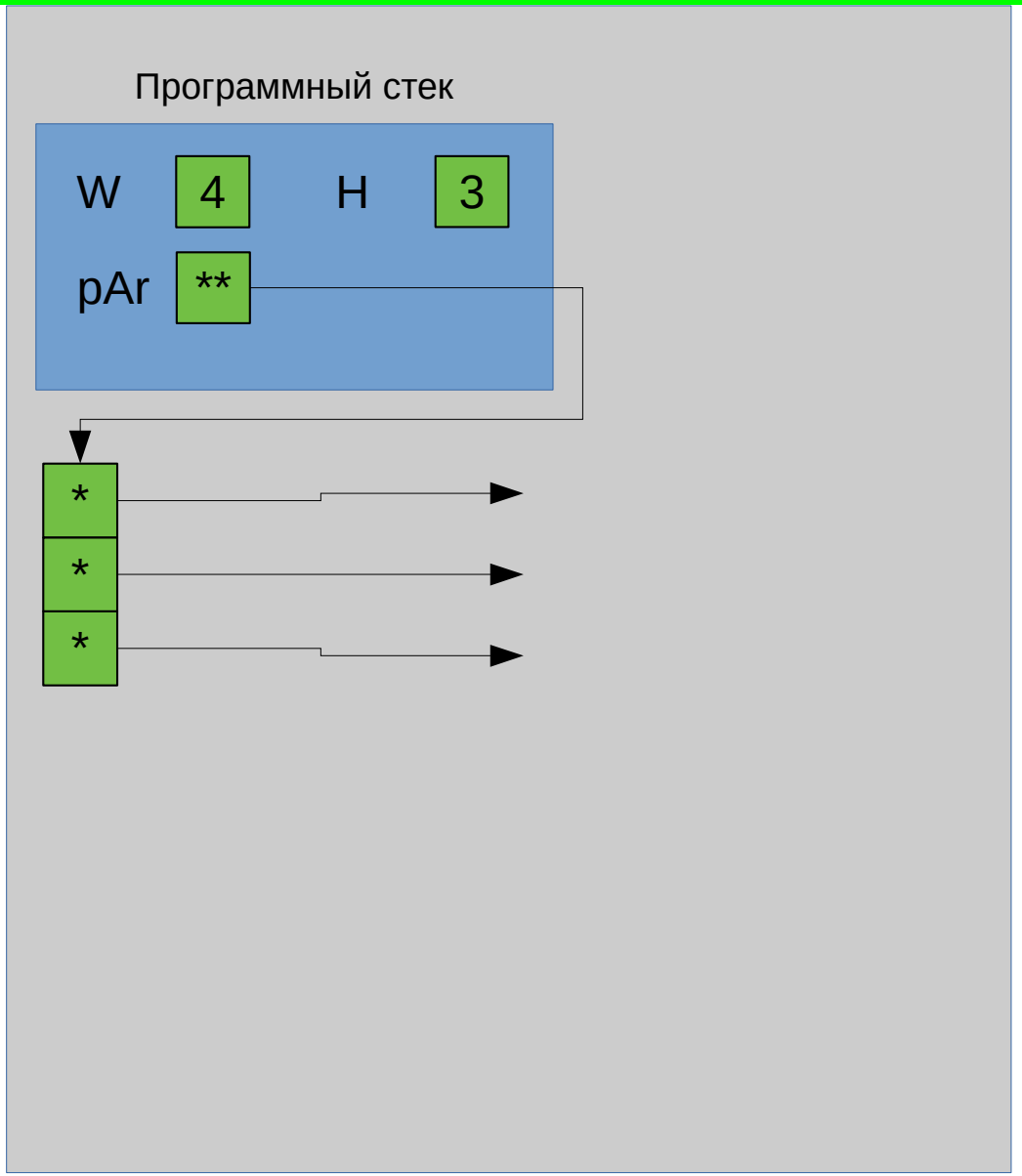
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** pAr;
  ar = new double*[H];
  for (int a=0; a<H; a++)
    pAr[a]=new double[W];

  pAr[0][0] = 1;
  pAr[2][3] = 21;
}
```



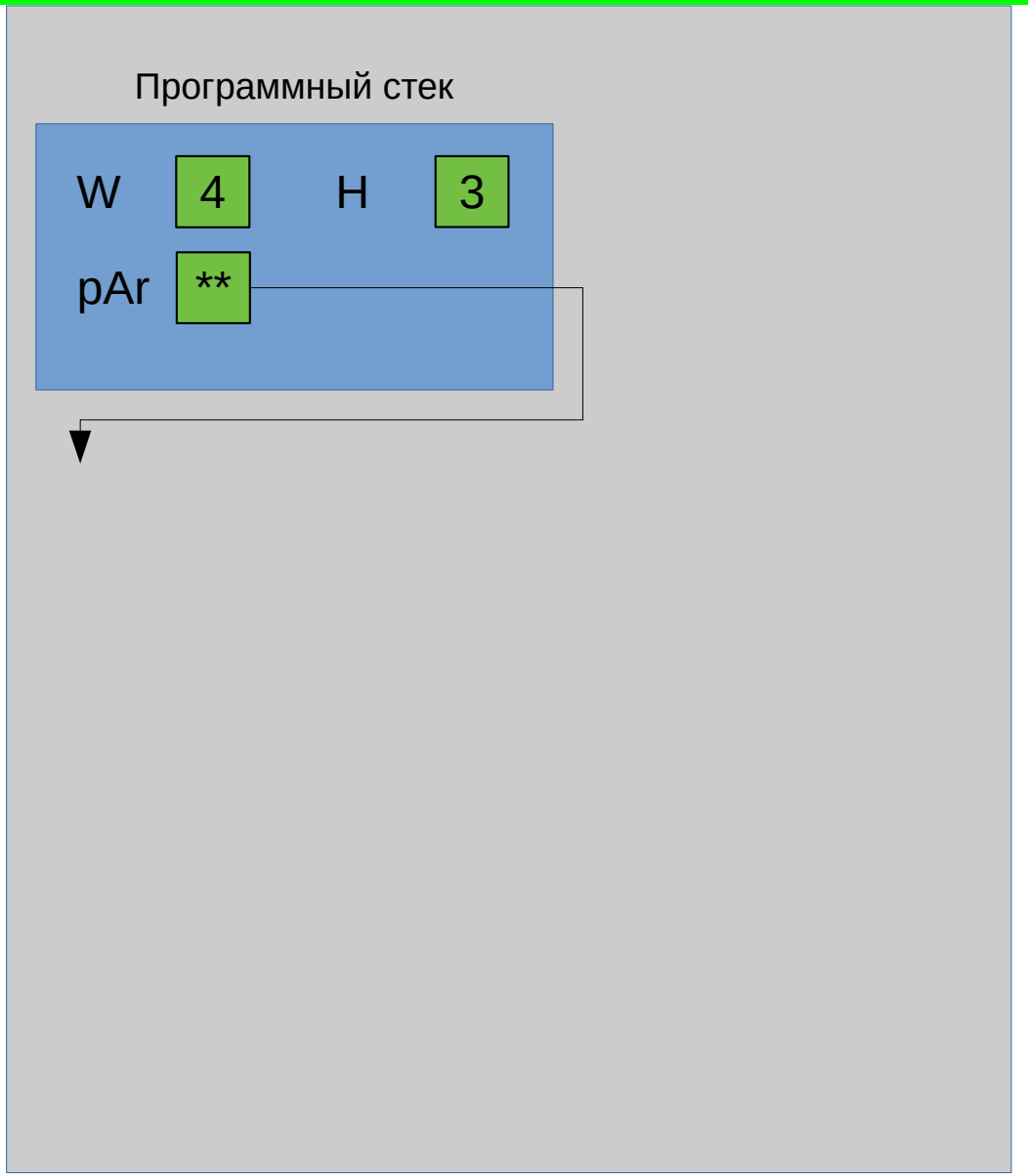
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** ar;
  ar = new double*[H];
  for (int a=0; a<H; a++)
    pAr[a]=new double[W];

  pAr[0][0] = 1;
  pAr[2][3] = 21;

  for (int a=0; a<H; a++)
    delete [] pAr[a];
}
```



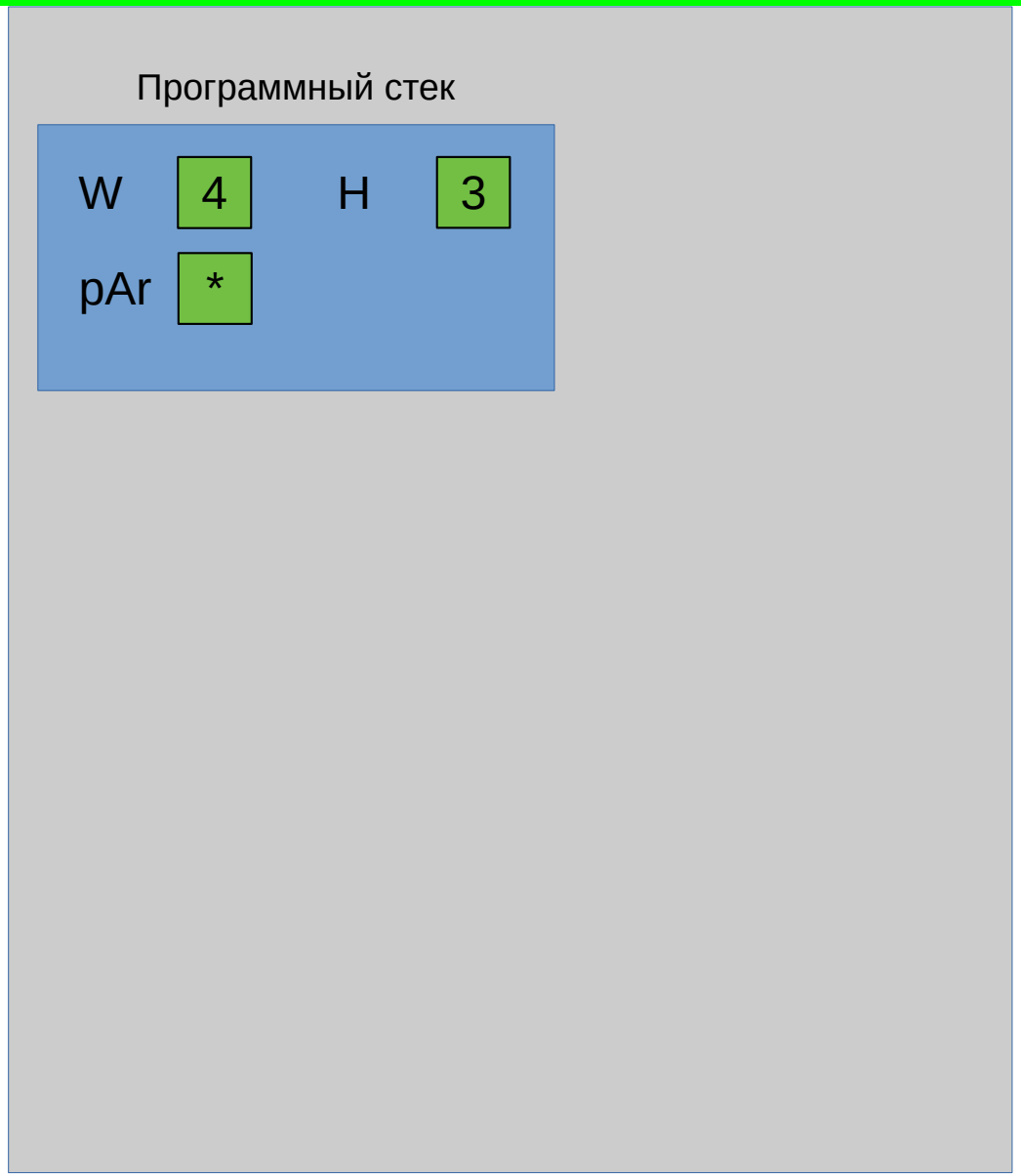
Массив из массивов

```
int main()
{
  int W = 4, H = 3;
  double** ar;
  ar = new double*[H];
  for (int a=0; a<H; a++)
    pAr[a]=new double[W];

  pAr[0][0] = 1;
  pAr[2][3] = 21;

  for (int a=0; a<H; a++)
    delete [] pAr[a];
  delete [] pAr;
}
```



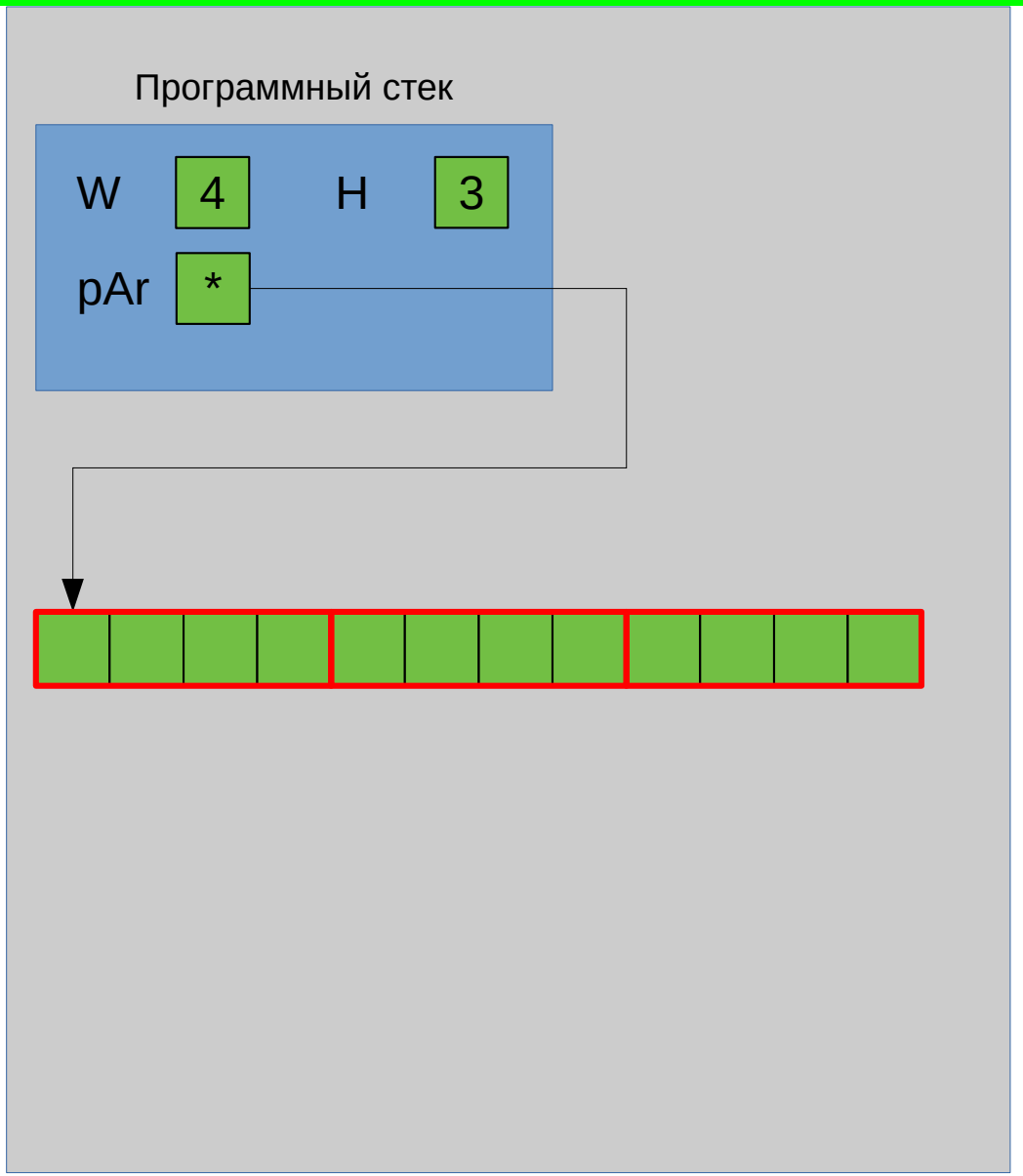
Массив с двумерной адресацией

```
int main()
{
  int W = 4, H = 3;
  double* pAr;
}
```



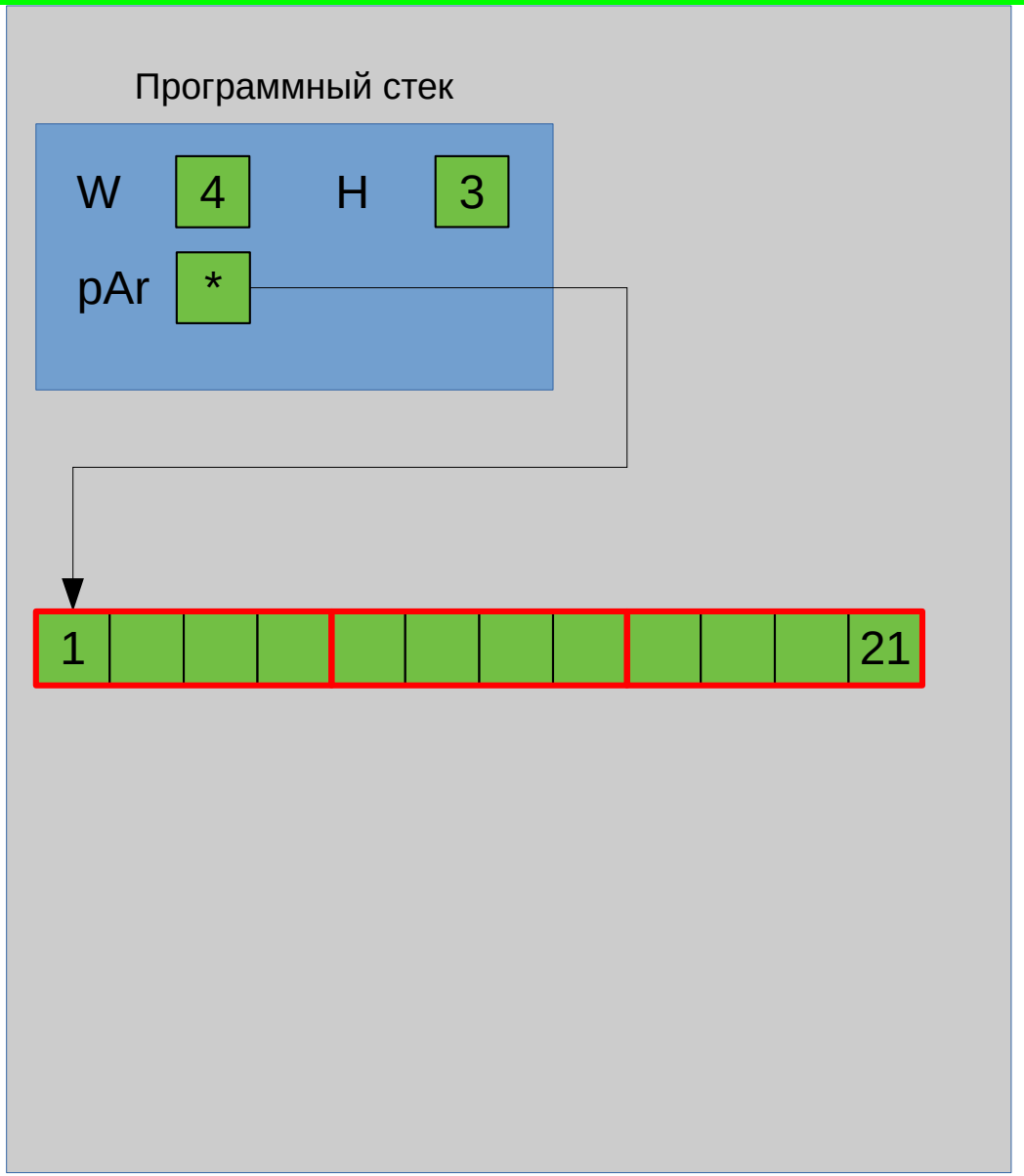
Массив с двумерной адресацией

```
int main()
{
  int W = 4, H = 3;
  double* pAr;
  pAr = new double[W*H];
}
```



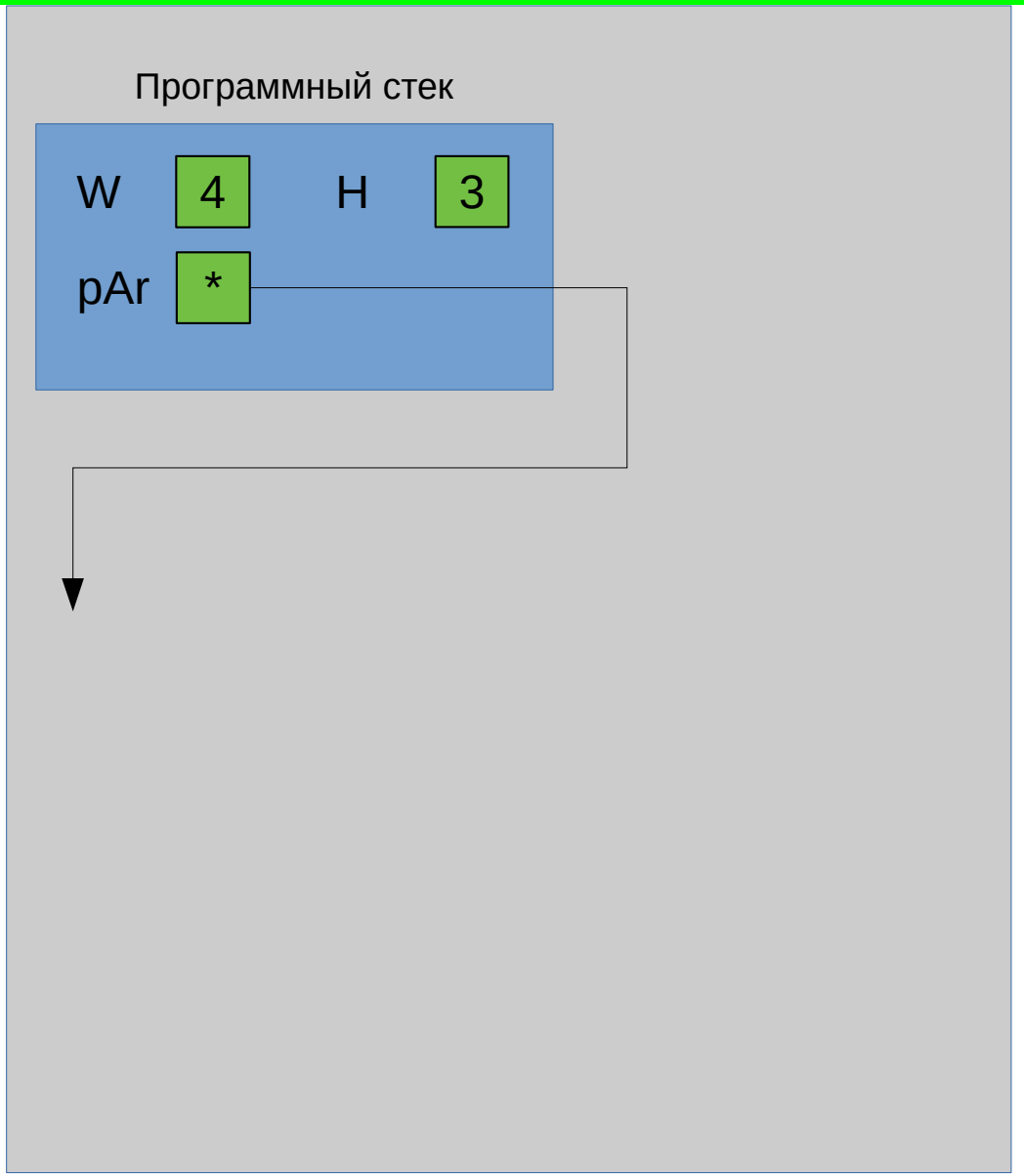
Массив с двумерной адресацией

```
int main()
{
  int W = 4, H = 3;
  double* pAr;
  pAr = new double[W*H];

  pAr[0*W + 0] = 1;
  pAr[2*W + 3] = 21;
}
```



Массив с двумерной адресацией

```
int main()
{
  int W = 4, H = 3;
  double* pAr;
  pAr = new double[W*H];

  pAr[0*W + 0] = 1;
  pAr[2*W + 3] = 21;

  delete [] pAr;
}
```



Недостатки динамических массивов

- Требуют использования указателей и, как следствие, аккуратной работы с памятью.
- Указатель не содержит информации о количестве элементов, на которое он указывает, и не позволяет проверить, принадлежит ли участок памяти, на который он указывает, приложению, то есть можно ли обратиться к этим данным.
- Для изменения размера массива необходимо создать новый массив нового размера и перенести данные из старого в новый, после чего удалить старый массив.
- Если не вызвать free / delete после использования, а переменная-указатель перестанет существовать, произойдет утечка памяти: блок памяти не будет доступен никому до завершения работы приложения.

Контейнеры

Решение проблем динамических массивов в C++ — набор классов-контейнеров в стандартной библиотеке шаблонов (STL).

Контейнер — это класс, хранящий произвольное количество однотипных элементов. Контейнеры — шаблонные классы, поэтому могут хранить любые элементы.

Контейнеры берут на себя всю работу с динамической памятью. Разные контейнеры отличаются по внутренней структуре и оптимизированы для разных операций.

Контейнер в любой момент времени знает количество хранимых в нем элементов.

Некоторые контейнеры позволяют использовать квадратные скобки, аналогично массивам.

Контейнер можно использовать как аргументы или возвращаемые значения функций.

Контейнеры

Все контейнеры STL поддерживают следующие методы:

- **empty()** возвращает true, если контейнер пустой, или false, если в нем есть хотя бы один элемент.
- **size()** возвращает количество элементов, хранимых в контейнере на момент вызова. На момент создания контейнер всегда хранит 0 элементов.
- **clear()** удаляет из контейнера все элементы. После вызова clear(), вызов size() вернет 0.
- **push_back()** добавляет элемент в конец контейнера. Аргумент метода — новое значение, размещаемое в контейнере.
- **pop_back()** удаляет последний элемент из контейнера. У метода нет аргументов или возвращаемого значения.
- Операция присвоения (=) заменяет содержимое одного контейнера содержимым другого контейнера того же типа.

std::vector

Класс **vector** хранит данные любого типа в виде массива.

```
vector<float> values;
```

Для использования класса vector необходимо подключить заголовочный файл **vector** и подключить пространство имен **std**.

```
#include <vector>
using namespace std;
```

Доступ к элементам вектора осуществляется при помощи **квадратных скобок**. Доступны индексы элементов с 0 по size()-1 включительно.

Изменить размер вектора можно при помощи метода **resize**, указав в качестве аргумента новый размер.

```
values.resize(24);
for (int a=0; a<values.size(); a++)
  values[a] = a;
```

Двумерный std::vector

Для создания двумерного массива при помощи вектора, необходимо создать вектор из векторов.

```
vector<vector<float>> matrix;
```

Необходимо задать количество строк в массиве.

```
int n_rows = 4, n_cols = 3;
matrix.resize(n_rows);
```

Каждая строка после resize будет содержать 0 элементов. Необходимо задать размер каждой строке:

```
for (int a=0; a<n_rows; a++)
  matrix[a].resize(n_cols);
```

Теперь можно заполнить матрицу числами.

```
for (int a=0; a<n_rows; a++)
  for (int b=0; b<n_cols; b++)
    matrix[a][b] = 0;
```

Итераторы STL

Не все контейнеры могут работать с целочисленными индексами. Универсальный способ работы с контейнером заключается в использовании специальных классов — итераторов.

Итератор — это класс, существующий для каждого контейнера, который позволяет получить доступ к одному элементу контейнера и осуществляет навигацию по элементам.

Итератор идеологически похож на указатель в пределах классического массива.

Каждый контейнер имеет два метода:

- **begin()** возвращает итератор, соответствующий первому элементу контейнера.
- **end()** возвращает итератор, идущий сразу после последнего элемента контейнера.

Для пустого контейнера begin() и end() возвращают одно и то же.

Итераторы STL

Итераторы делятся на категории. В зависимости от категории, они поддерживают разные операции.

- Все категории итераторов поддерживают операции:

  ++ — переводит итератор на следующий элемент контейнера

  ***** — разыменовывает итератор и возвращает значение элемента

  == — позволяет сравнить два итератора (указывают ли они на один и тот же элемент).

- Двухсторонние итераторы дополнительно поддерживают:

  -- — переводит итератор на предыдущий элемент контейнера

- Итераторы произвольного доступа поддерживают арифметические операции и операции сравнения.

Категория итератора приводится в справочном руководстве контейнера.

Итераторы std::vector

При помощи итераторов можно записать цикл обхода вектора.

```
vector<double> values;
vector<double>::iterator it;
for (it = values.begin(); it != values.end(); it++)
  cout << *it << endl;
```

Начиная со стандарта C++11, можно использовать ключевое слово auto и не записывать вручную тип итератора. Такая форма записи универсальна и подойдет для любого контейнера:

```
for (auto it = values.begin(); it != values.end(); it++)
  cout << *it << endl;
```

Также при помощи итератора можно модифицировать значение в контейнере:

```
auto it = values.begin() + 4;
*it = 2;
```

Итераторы std::vector

Итераторы позволяют вставлять значения в середину вектора, используя метод **insert**. Метод вставляет новый элемент перед тем, на который указывает итератор, и после вставки будет указывать на созданный элемент.

```
vector<double> values;
auto it = values.begin() + 2;    // Вставка в позицию 2
it = values.insert(it, 24);      // значения 24
```

Удаление элементов осуществляется методом **erase**. Он принимает либо один итератор на удаляемый элемент, либо 2 итератора на удаляемый диапазон (включая первый, не включая последний). Возвращает итератор, указывающий на элемент после последнего удаленного.

```
auto it1 = values.begin() + 4; // Удаление элементов с 4
auto it1 = values.begin() + 9; // по 8 включительно
it1 = values.erase(it1, it2);
```

std::list

Использование vector оправдано там, где часто требуется прямой доступ по индексу, однако вставка и удаление элементов вектора — трудоемкая операция, которая может потребовать перевыделения внутреннего массива.

В противоположность вектору, класс **list** хранит элементы в виде двусвязного списка. Каждый элемент снабжается указателем на следующую и предыдущую запись внутри list. Указатели крайних элементов обнуляются.

В отличие от vector, для list недоступен оператор квадратных скобок, и работа с ним возможна только через итераторы. Однако операции вставки и удаления элементов списка делаются за постоянное время, не зависящее от размера списка.

std::list

Для использования класса **list** необходимо подключить заголовочный файл list и подключить пространство имен std.

```
#include <list>
using namespace std;
```

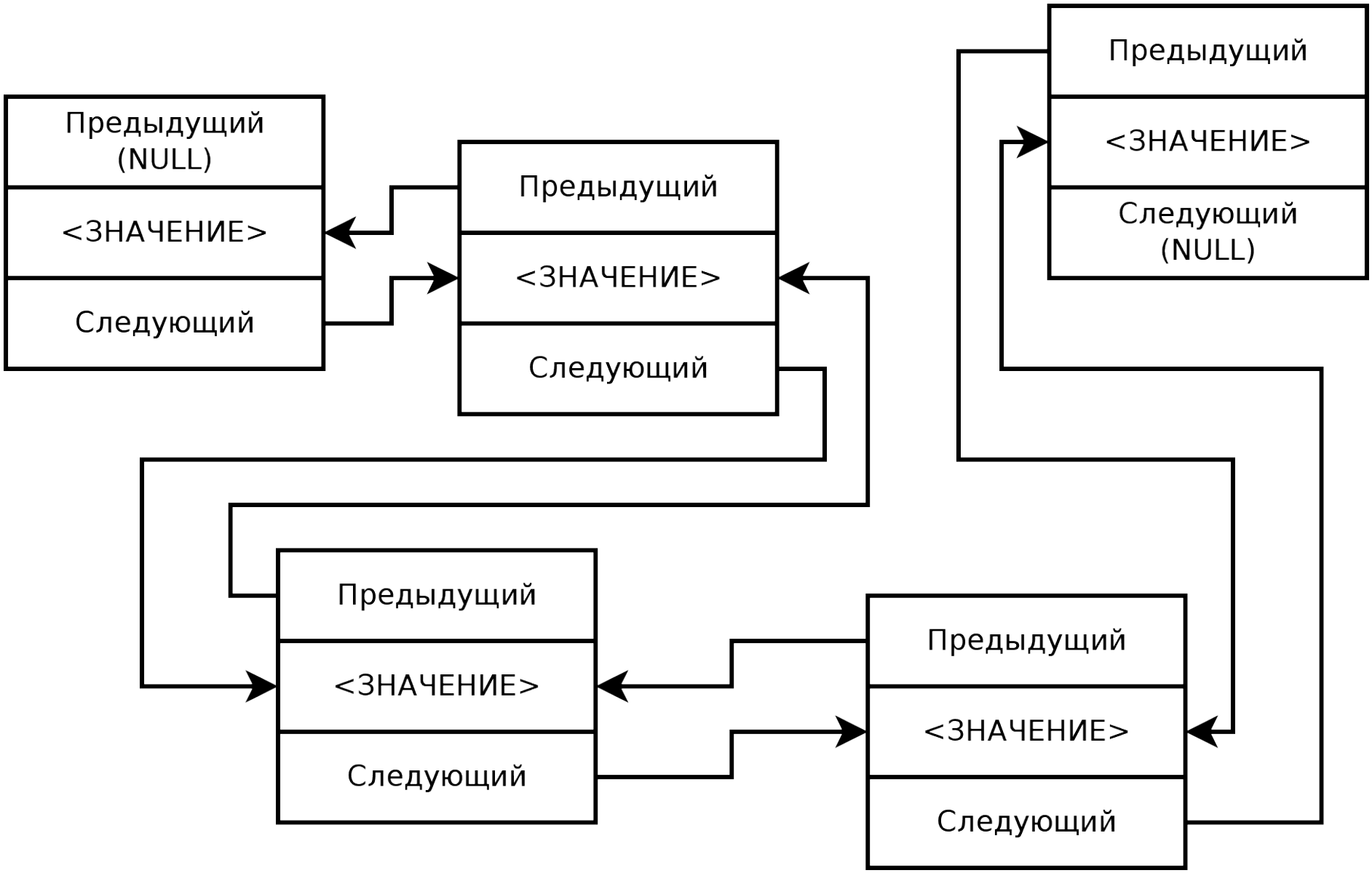
Для списка также доступен метод **resize()**, изменяющий количество хранимых элементов.

Доступ к элементам списка осуществляется при помощи итераторов.

```
list<double> values;
for (auto it = values.begin(); it != values.end(); it++)
  cout << *it << endl;
```

Добавление элементов производится методами **push_back** и **insert**. Удаление — методом **erase**. Их работа аналогична описанной для vector.

std::list


Предыдущий
(NULL)
<ЗНАЧЕНИЕ>
Следующий
Предыдущий
<ЗНАЧЕНИЕ>
Следующий
Предыдущий
<ЗНАЧЕНИЕ>
Следующий
(NULL)
Предыдущий
<ЗНАЧЕНИЕ>
Следующий
Предыдущий
<ЗНАЧЕНИЕ>
Следующий

Некоторые свойства итераторов

Для контейнера, использующего итераторы, можно организовать цикл по диапазону (range-based for), начиная с C++11. Переменная цикла будет последовательно принимать значения из контейнера. Рекомендуется использовать ссылки.

```
list<double> lst;
for (double& d : lst)  // Или: for (auto& d : lst)
  cout << d << endl;
```

Многие функции принимают диапазон итераторов в качестве аргументов. Например, функция сортировки контейнера:

```
#include <algorithm>
std::sort(lst.begin(), lst.end());
```

Некоторым функциям, таким как std::copy, может потребоваться специальный итератор для вставки элементов в конец.

```
list<double> l2;
std::copy(lst.begin, lst.end(), std::back_inserter(l2));
```

Другие контейнеры STL

Стандартная библиотека шаблонов (STL) содержит ряд контейнеров для широкого круга задач:

- **map** — это ассоциативный контейнер, хранящий пары «Ключ» — «Значение». Ключи могут идти в произвольном порядке. Поддерживает оператор квадратных скобок, в которых указывается значение ключа, однако для перебора всех значений нужно использовать итераторы.
- **deque** — двусторонняя очередь. Контейнер оптимизирован для быстрого добавления и удаления элементов в начало и конец.
- **set** — хранит набор уникальных значений. Попытка добавить уже имеющийся в set элемент не приведет к изменению контейнера.
- **stack** — колодец, добавляющий элементы в конец и убирающий их с конца. Последний добавленный элемент извлекается первым.

Массивы Python

В Python изначально отказались от поддержки массивов, так как посчитали их ограниченными по функционалу и подверженными нетривиальным ошибкам.

Вместо массивов Python использует классы-контейнеры, позволяющие хранить произвольное количество элементов любых типов. Использование данных классов почти полностью покрывает все необходимости в составных типах данных.

- **Кортеж** (**tuple**) — неизменяемое хранилище данных. Отдельные переменные внутри кортежа различаются по индексам.
- **Список** (**list**) — это хранилище данных, аналогичное кортежу, но способное менять свое содержимое.
- **Словарь** (**dict**) — это хранилище данных, в котором доступ к элементам осуществляется по ключу, а не по индексу.

Кортеж

Создается при помощи круглых скобок, в которых через запятую записаны значения входящих в кортеж переменных.

```
T = (1, 2, "24", "hello!")
```

Доступ к элементам кортежа осуществляется по индексу при помощи квадратных скобок. Нумерация может идти как с начала, так и с конца. Для кортежа из N элементов справедливы индексы [0 .. N-1] и [-N .. -1]

```
print(T[0])    # 1
print(T[-1])   # hello!
```

Элементами кортежа могут быть любые типы данных, в том числе другие кортежи:

```
T = ((1,1), (1,2), (2,2))
print(T[0])      # (1,1)
print(T[1][1])   # 2
```

Кортеж

Длину кортежа можно получить при помощи встроенной функции **len**:

```
T = (1, 2, "24", "hello!")
print(len(T))      # 4
```

Для выбора нескольких идущих подряд элементов используется двоеточие внутри квадратных скобок. При этом первый индекс всегда включается в диапазон, а второй не включается.

Если не указывать первый индекс, то выбор идет с начала. Если не указывать второй — то до конца.

```
print(T[:2])       # (1, 2)
print(T[1:3])      # (2, "24")
print(T[2:])       # ("24", "hello!")
print(T[1:-1])     # (2, "24")
```

Кортеж

Одиночная переменная, заключенная в скобки, не превращается в кортеж:

```
T = (4.25)
print(type(T))      # <class 'float'>
```

Кортеж, состоящий из одной переменной, создается так:

```
T = (1,)
```

Это необходимо, чтобы избежать случайного создания кортежей в математических выражениях.

Пустой кортеж создается так:

```
empty_T = ()                # либо empty_T = tuple()
print(len(empty_T))         # 0
```

Список

Создается при помощи квадратных скобок, в которых через запятую записаны значения входящих в список элементов.

```
L = [2, 32, 42, "nop", "Hi there!"]
```

Индексы элементов списка точно такие же, как у кортежа, и доступ к элементам осуществляется точно так же через квадратные скобки.

```
print(L[0])       # 2
print(L[-1])      # Hi there!
```

Элементами списка могут быть любые типы данных, в том числе другие списки и кортежи:

```
L = [(1,1), (1,2), (2,2), ["01", "11", "21", "31"]]
print(L[0])       # (1,1)
print(L[-1][2])   # 21
```

Список

Длину списка можно получить при помощи встроенной функции **len**:

```
L = [2, 32, 42, "nop", "Hi there!"]
print(len(L))      # 5
```

Для выбора нескольких идущих подряд элементов используется двоеточие внутри квадратных скобок. При этом первый индекс всегда включается в диапазон, а второй не включается.

Если не указывать первый индекс, то выбор идёт с начала. Если не указывать второй — то до конца.

```
print(L[:2])       # [2, 32]
print(L[1:3])      # [32, 42]
print(L[2:])       # [42, "nop", "Hi there!"]
print(L[1:-1])     # [32, 42, "nop"]
```

Список

Модификация элементов списка также идет через квадратные скобки:

```
L = [2, 32, 42, "nop", "Hi there!"]
L[1] = -4.41
print(L)    # [2, -4.41, 42, "nop", "Hi there!"]
```

Удаление элемента осуществляет оператор **del**:

```
del L[-1]
print(L)    # [2, -4.41, 42, "nop"]
```

Можно убрать элемент из списка и переложить его в другую переменную используя метод **pop**. Аргумент метода — индекс удаляемого элемента.

```
a = L.pop(3)
print(L)    # [2, -4.41, 42]
print(a)    # nop
```

Список

Добавление элементов в конец списка производится при помощи метода **append**:

```
L = [2, -4.41, 42]
L.append("End")
print(L)              # [2, -4.41, 42, "End"]
```

Вставка метода в произвольную позицию в списке осуществляется методом **insert**. Аргументы метода — позиция, перед которой нужно вставить значение, и само значение.

```
L = [2, -4.41, 42]
L.insert(0, 10)
print(L)              # [10, 2, -4.41, 42, "End"]
```

Словарь

Создается при помощи фигурных скобок. Каждый элемент словаря — это пара «Ключ»-«Значение», разделенные двоеточием.

```
D = {"one": 1, "two": 2}
```

Доступ к элементам словаря осуществляется путем указания значения ключа в квадратных скобках.

```
print(D["one"])    # 1
```

Запись значения в словарь также происходит при помощи оператора квадратных скобок. При этом, если на момент записи значение с таким ключом не существовало в словаре, оно будет создано, а если существовало — то будет перезаписано.

```
D["two"] = 22
D["three"] = 3
print(D)           # {"one": 1, "two": 22, "three": 3}
```

Цикл по массиву в Python

Оператор **in** в Python позволяет быстро проверить, входит ли элемент в контейнер. Для списков и тьюплов проверяется, присутствует ли данное значение в списке или тьюпле. Для словаря — есть ли в словаре такой ключ.

```
L = ["one", "two", "three"]
D = {"one" : 1, "two" : 2}
if ("three" in L) and not ("three" in D):
    D["three"] = 3
```

Также in позволяет записать список по элементам списка или тьюпла, либо по ключам словаря:

```
for a in L:
    print(a)
for a in D:
    print(str(a) + " : " + str(D[a]))
```

Двумерные массивы в Python

Двумерный массив в Python может быть создан как массив из одномерных массивов. Для создания массива из массивов необходимо следующее:

```
n_rows = 4
n_cols = 6
# создание списка из n_rows пустых списков
ar = [[]] * n_rows
for a in range(n_rows):
  # создание для каждой строки списка из n_cols нулей
  ar[a] = [0] * n_cols
```

После создания такого массива, элементы доступны через двойные квадратные скобки

```
for a in range(n_rows):
  for b in range(n_cols):
    ar[a][b] = 0
```

Файловый ввод-вывод

Зачастую большие объемы данных представляются в виде массивов и хранятся в текстовых или двоичных файлах.

Двоичный файл хранит копию участка памяти. При чтении или записи двоичного файла данные никак не модифицируются.

Текстовый файл хранит данные в текстовом виде, пригодном для человека. При чтении или записи текстового файла происходит преобразование данных.

Двоичный файл. 10, тип int, и 3.5, тип float.

```
0a 00 00 00
00 00 60 40
```

```
□□□□
□□`@
```

Текстовый файл. Те же данные.

```
10
3.5
```

Текстовый файл в двоичном виде.

```
31 30 0a 33
2e 35 0a
```

```
10□3
.5□
```

Файловый ввод-вывод

Единственный способ уникальной идентификации файла — его полное имя в файловой системе (полный путь от корня диска).

Для работы с файлами существует специальный программный интерфейс — **файловый дескриптор**. Это специальная переменная, идентифицирующая файл для функций ввода-вывода.

После создания файлового дескриптора необходимо **открыть** файл, то есть установить соответствие между его полным именем и файловым дескриптором.

Если файл успешно открыт, можно использовать функции чтения и записи данных, осуществляющие обмен между переменными программы и содержимым файла.

По окончании работы с файлом следует **закрыть** его.

Файловый ввод-вывод

Создать дескриптор и массив данных

Дескриптор

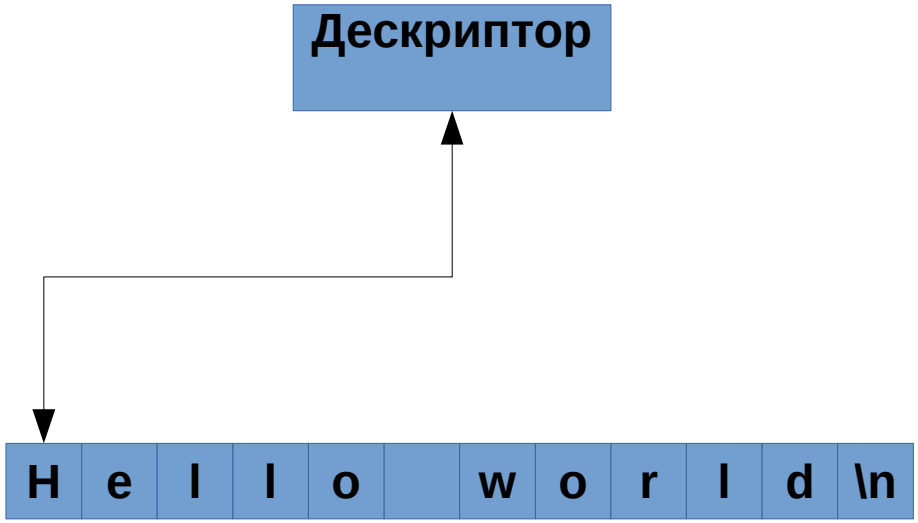
Содержимое файла

Файловый ввод-вывод

Создать дескриптор и массив данных

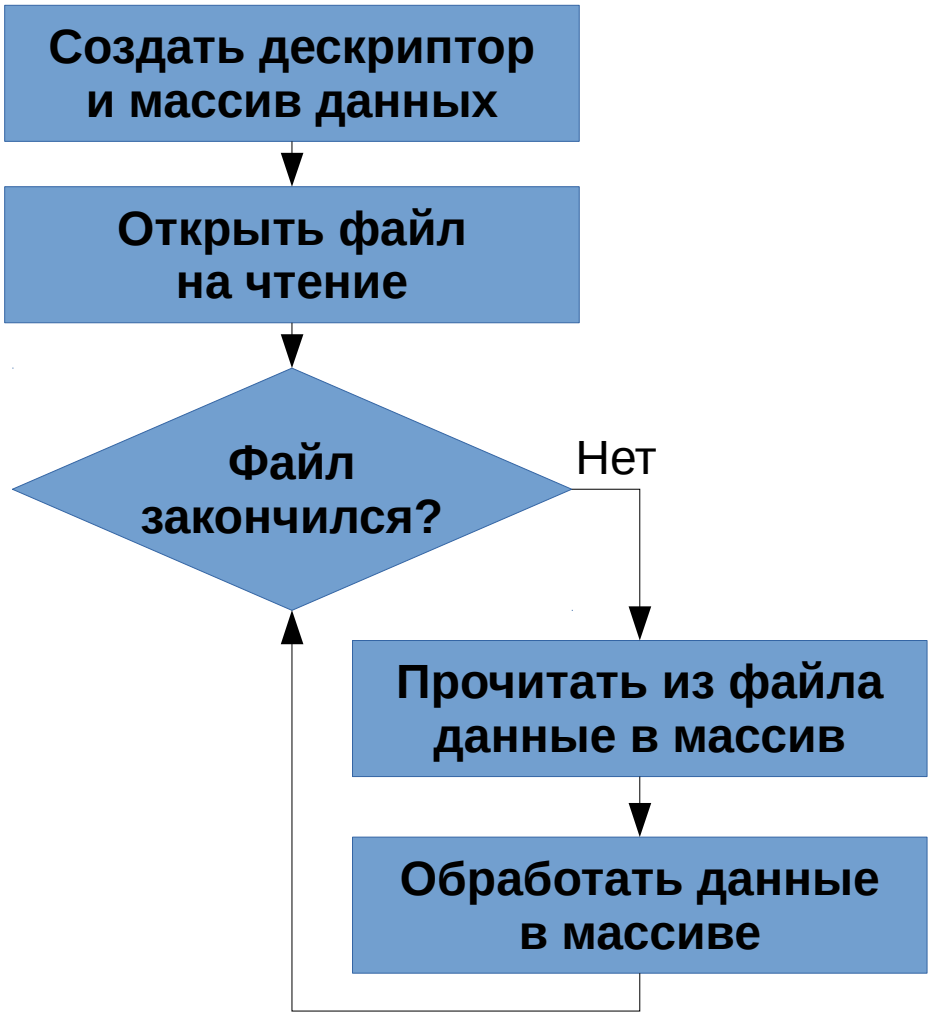
↓

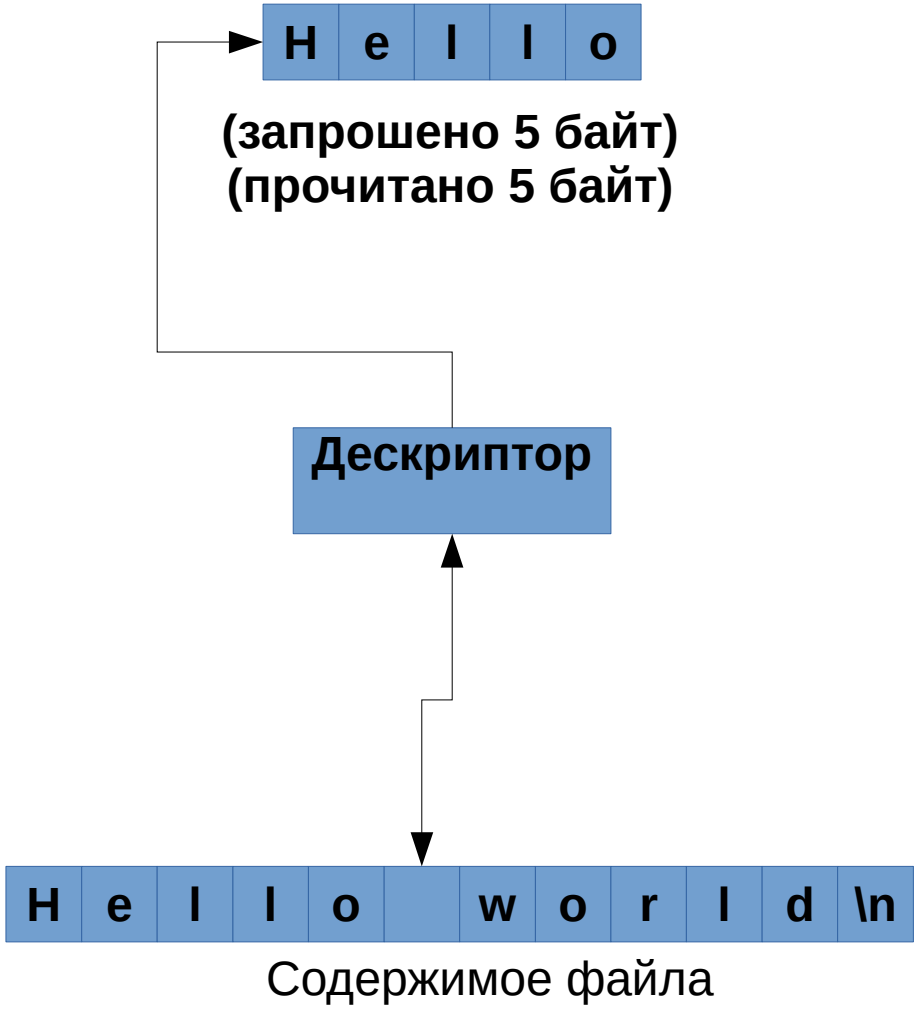
Открыть файл на чтение



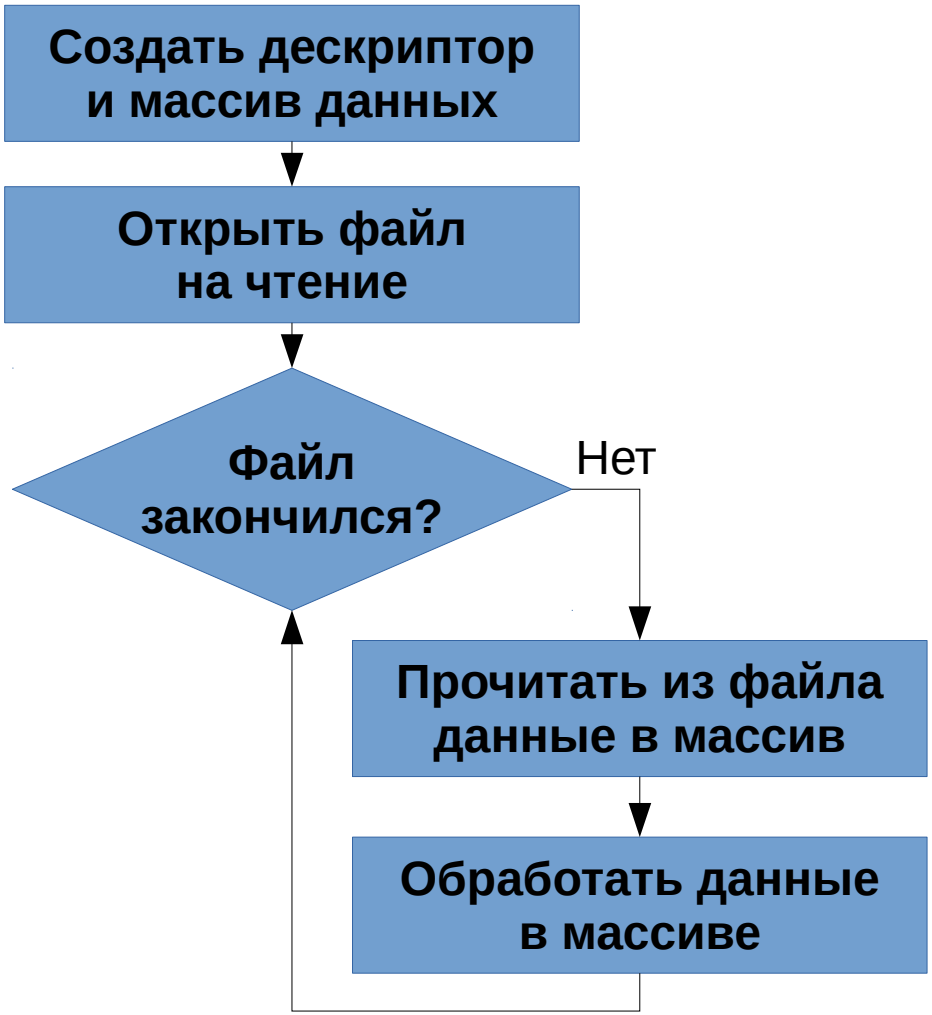
Содержимое файла

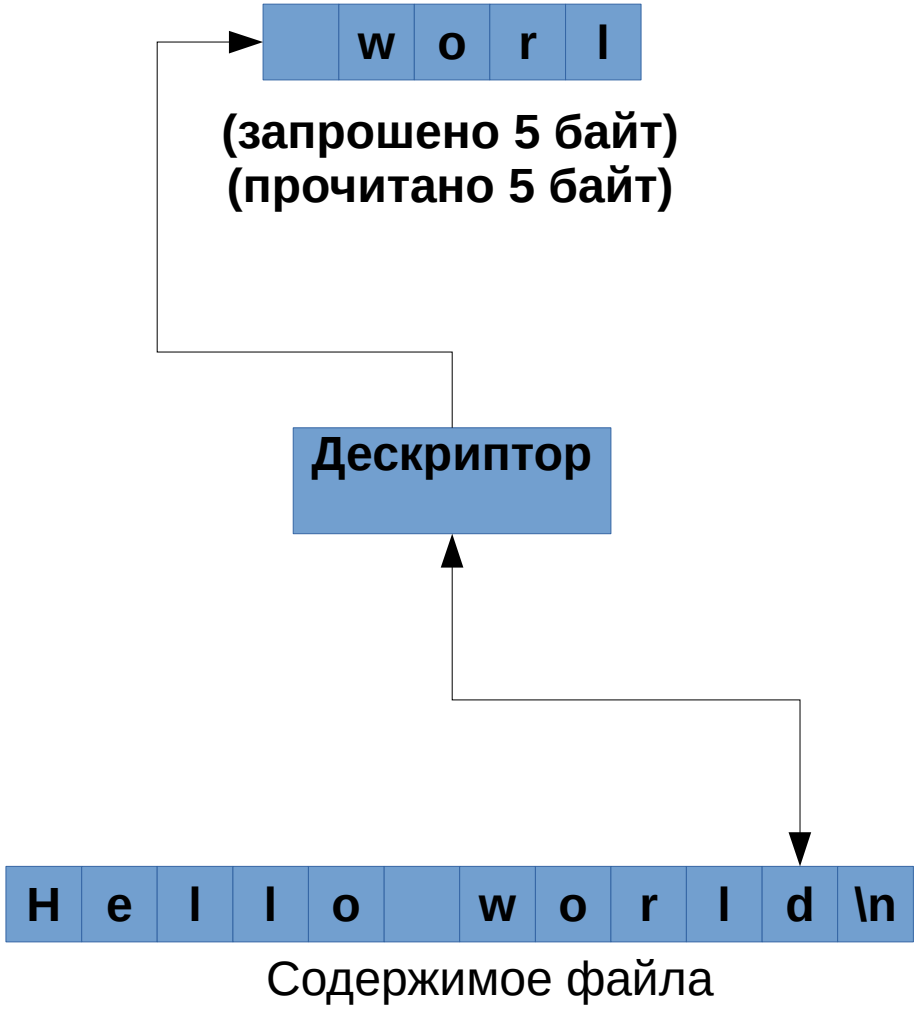
Файловый ввод-вывод



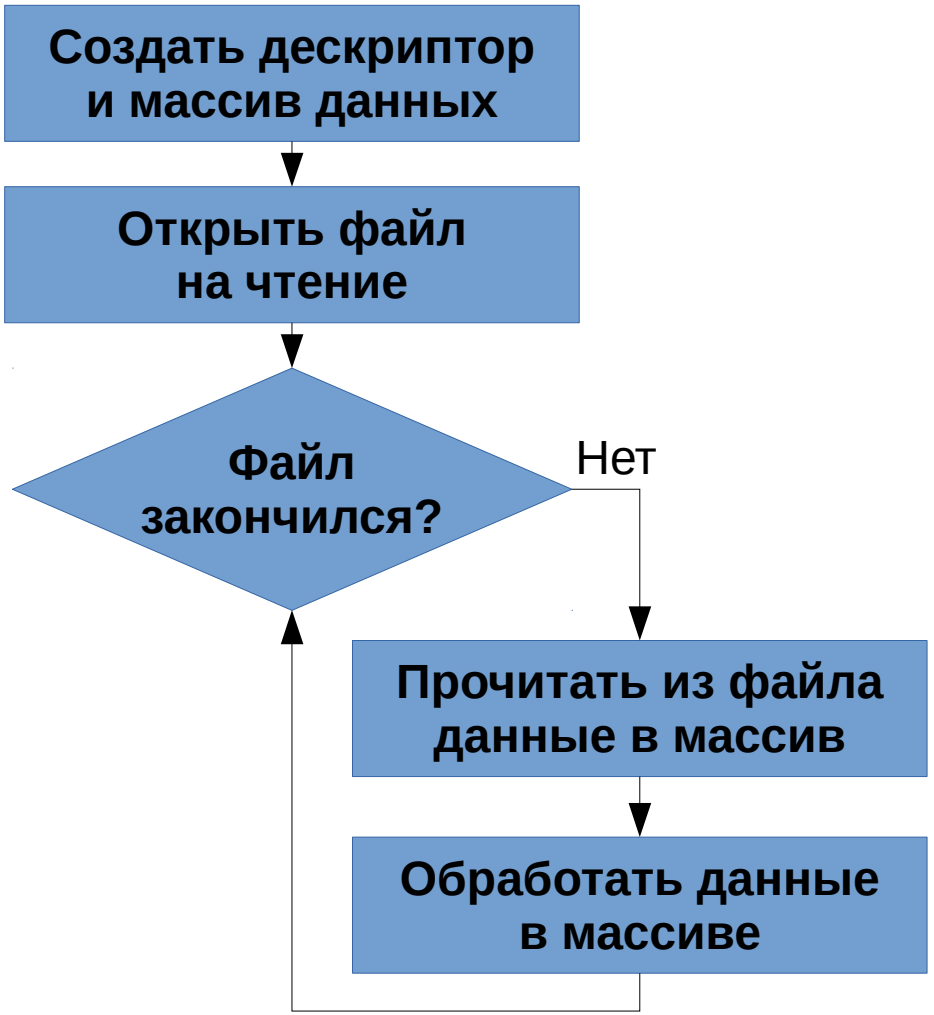


Файловый ввод-вывод

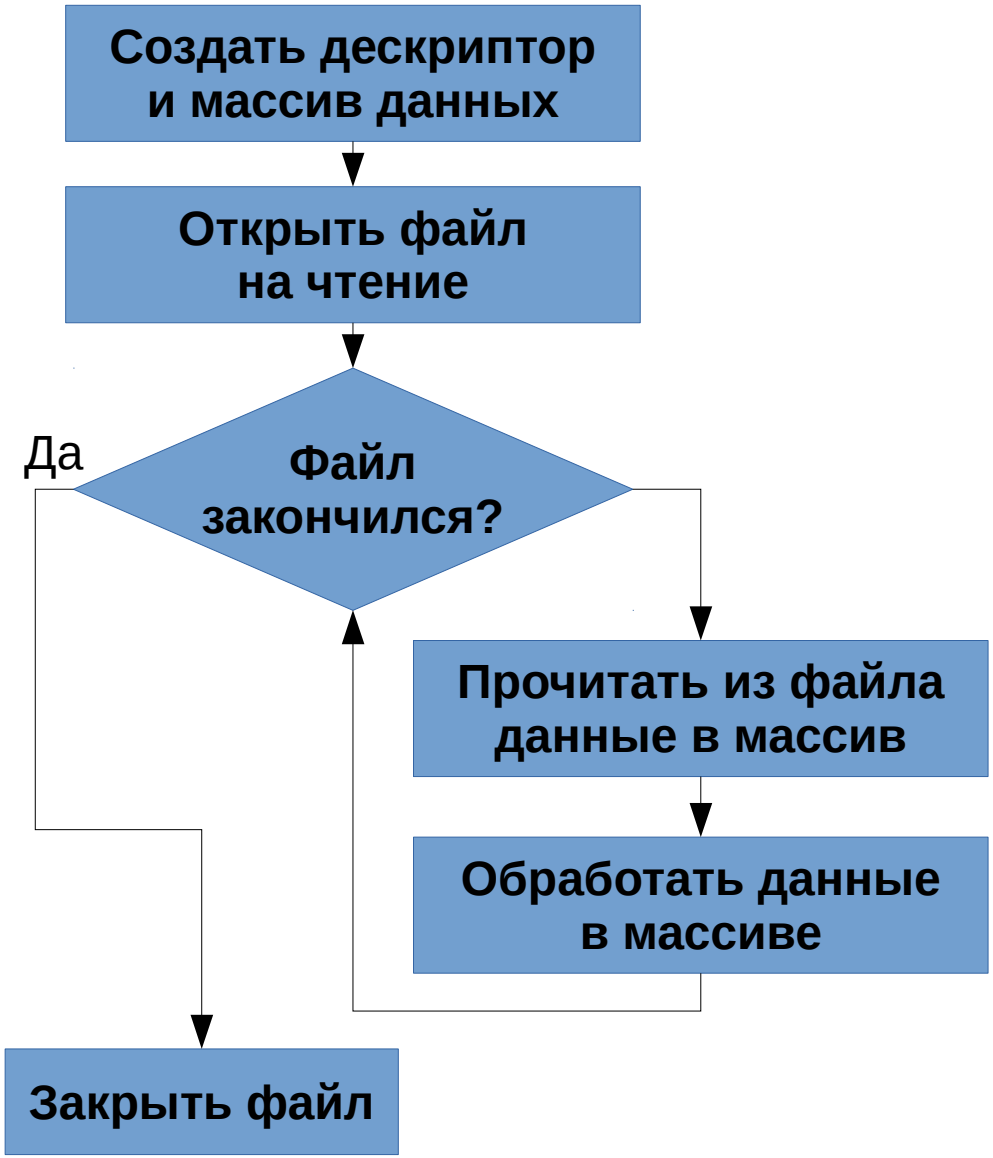


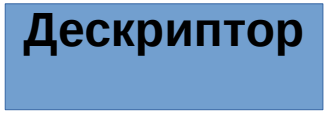


Файловый ввод-вывод



Файловый ввод-вывод





Содержимое файла

Файловый ввод-вывод в C

Стандартная бибиотека языка C содержит заголовочный файл **stdio.h**, в котором объявлен файловый дескриптор — переменная типа **FILE***. Для открытия файла необходимо вызвать функцию **fopen**, в которую нужно передать путь к файлу и режим открытия.

```
#include <stdio.h>
FILE* f = fopen("example.txt", "r");
```

- **"r"** — открыть файл на чтение. Файл должен существовать.
- **"w"** — открыть файл на запись. Файл будет создан, если не существовал. Существующий файл будет усечен до нулевого размера.
- **"a"** — открыть файл на дозапись. Файл должен существовать, и он не будет усечен. Запись будет идти с конца файла.
- Суффикс "b" открывает файл в двоичном режиме (например "rb").

Если файл нельзя открыть, функция **fopen** вернет **NULL**.

Файловый ввод-вывод в C

С открытым файлом можно работать при помощи функций **fread** и **fwrite**, осуществляющих чтение и запись данных без их преобразования (в двоичном режиме):

```
FILE* in  = fopen("source.bin", "rb");
FILE* out = fopen("dest.bin",   "wb");
int ar[32];
int sz = fread(ar, sizeof(int), 32, in);
fwrite(ar, sizeof(int), sz, out);
```

Аргументы функций:

1) Адрес участка памяти, с которым идет обмен данными.

2) Размер одного элемента.

3) Количество элементов.

4) Файловый дескриптор, с которым идет обмен данными.

Возвращаемое значение — количество обработанных элементов.

Файловый ввод-вывод в C

Для работы в текстовом режиме существуют функции **fprintf** и **fscanf**. Они работают точно так же, как printf и scanf, но вместо терминала используют текстовый файл, дескриптор которого передается первым аргументом.

```
FILE* in  = fopen("source.txt", "r");
FILE* out = fopen("dest.txt",   "w");
int value;
fscanf(F, "%d", &value);
fprintf(F, "%d", value)
```

Строковые данные не нуждаются в преобразовании, поэтому могут быть записаны в файл или прочитаны из файла как в текстовом, так и в двоичном режиме.

Файловый ввод-вывод в C

Чтобы определить в процессе чтения, когда закончился файл, необходимо считывать возвращаемое значение функции чтения, либо использовать функцию **feof** (сокращение от End Of File). Пример показывает чтение файла построчно, используя функцию **fgets**, считывающую одну строку.

```
FILE* in  = fopen("source.txt", "r");
while (!feof(in))
{
  char line[256];
  fgets(line, 256, F);
}
```

Аргументы fgets: массив для помещения результата чтения, максимальное количество символов для чтения и файловый дескриптор.

Файловый ввод-вывод в C

По окончании работы с файлом необходимо закрыть его, используя функцию **fclose**.

```
FILE* in  = fopen("source.txt", "r");
while (!feof(in))
{
  char line[256];
  fgets(line, 256, F);
}
fclose(in);
```

После закрытия файла, можно использовать тот же дескриптор для открытия другого файла.

Файловый ввод-вывод в C++

Стандартная библиотека языка C++ предлагает заголовочный файл **fstream** и набор классов для чтения и записи файлов. Все эти классы объявлены в пространстве имен std.

- **ifstream** — поток ввода данных, используется для чтения данных из файла.
- **ofstream** — используется для записи данных в файл.
- **fstream** — может использоваться как для чтения, так и для записи, в зависимости от переданных в конструктор аргументов.

```
#include <fstream>
ifstream ifs("example.txt");
```

После создания экземпляра класса, файл сразу открывается в нужном режиме и остается открытым до удаления объекта или до вызова метода **close**.

```
ifs.close();
```

Файловый ввод-вывод в C++

Второй аргумент конструктора файлового потока — это набор констант, объединенных через побитовое ИЛИ:

- **ios::in** — открыть файл на чтение. Недоступно для ofstream.
- **ios::out** — открыть файл на запись. Недоступно для ifstream.
- **ios::binary** — открыть файл в двоичном режиме.
- **ios::ate** — вывод в файл начинается с конца (At The End).
- **ios::app** — открыть файл на дозапись (append).
- **ios::trunc** — усечь файл до нулевого размера (truncate).

Метод **is_open** проверяет, открыт ли файл.

```
fstream fs("example.bin", ios::in | ios::binary);
if (!ifs.is_open())
  cout << "Unable to open file!" << endl;
```

Файловый ввод-вывод в C++

Текстовый ввод-вывод может осуществляться через операторы **<<** и **>>** так же, как через cin и cout.

```
ifstream in("input.txt");
ofstream out("output.txt");
int val;
in >> val;
out << val;
```

Метод **eof** покажет, достигнут ли в процессе чтения конец файла. Метод **getline** позволяет прочитать из текстового файла одну строку. С их помощью можно разобрать файл построчно:

```
while (!in.eof())
{
  char buf[256];
  in.getline(buf, 256);
}
```

Файловый ввод-вывод в C++

Двоичный ввод-вывод происходит при помощи методов **read** и **write**. В качестве аргументов каждый из них принимает адрес массива однобайтных элементов (char*), с которым будет происходить обмен данными, и его размер.

Для определения объема данных, прочитанного за последнюю операцию, необходимо использовать метод **gcount**.

```
ifstream in("input.bin", ios::binary);
ofstream out("output.bin", ios::binary);
while (!in.eof())
{
  char buf[1024];
  in.read(buf, 1024);
  int size = in.gcount();
  out.write(buf, size);
}
```

Файловый ввод-вывод в Python

Для открытия файла в Python используется стандартная функция **open**. Она принимает в качестве аргументов две строки: путь к файлу и режим, в котором необходимо открыть файл. Строка с режимом открытия взята из функции fopen библиотеки языка C.

- "r" — открыть файл на чтение. Файл должен существовать.
- "w" — открыть файл на запись. Файл будет создан, если не существовал. Существующий файл будет усечен до нулевого размера.
- "a" — открыть файл на дозапись. Файл должен существовать, и он не будет усечен. Запись будет идти с конца файла.
- Суффикс "b" открывает файл в двоичном режиме (например "rb").

Если файл нельзя открыть, функция **open** вернет **None**.

Файловый ввод-вывод в Python

Python не предусматривает разных функций для двоичного и текстового ввода-вывода. Все операции чтения или записи происходят при помощи методов **read** и **write**.

При работе в двоичном режиме, методы работают с данными типа **bytes** или **bytearray**. Все передаваемые данные необходимо преобразовать в байтовые массивы на записи и из байтовых массивов — на чтении. Для этого используется модуль **struct**, а также методы строк **encode** и **decode**.

В текстовом режиме методы работают со строками **str**.

Метод **read** считывает содержимое всего файла и возвращает его в виде строки. При передаче целочисленного аргумента, метод считает столько байт, сколько передано. Если **read** прочитал меньше байт, чем было запрошено, значит файл закончился.

Метод **write** записывает переданную строку в файл.

Файловый ввод-вывод в Python

Пример показывает копирование файла в двоичном режиме. Для этого необходимо открыть один файл на чтение и второй — на запись. Далее данные читаются порциями по 1024 байта из первого файла и записываются во второй файл.

Процесс заканчивается, когда из файла inf ничего нельзя прочитать.

```
inf = open("input.bin", "rb");
outf = open("output.bin", "wb");
while True:
    buf = inf.read(1024)
    if len(buf) == 0:
        break
    outf.write(buf)
inf.close()
outf.close()
```

Файловый ввод-вывод в Python

При обработке текстовых файлов можно представить текстовый файл как множество строк и пройти по нему при помощи цикла. Пример показывает построчное чтение файла и запоминание его содержимого в списке.

```
L = []
f = open("input.txt", "r");
for line in f:
    L.append(line)
f.close()
```

Для записи массива чисел из массива в файл в читаемом виде необходимо каждое число преобразовать к строке.

```
values = [ ... ]                    # Массив чисел
f = open("dump.txt", "a")
for val in values:
    f.write(str(val) + "\n")  # Одно число — одна строка
```