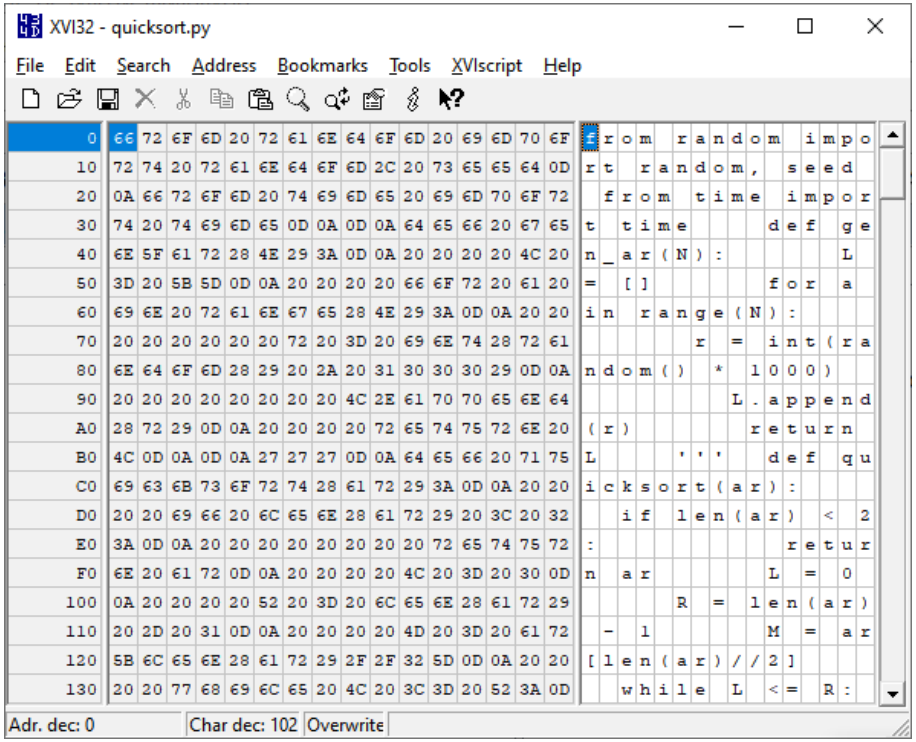
Двоичные данные

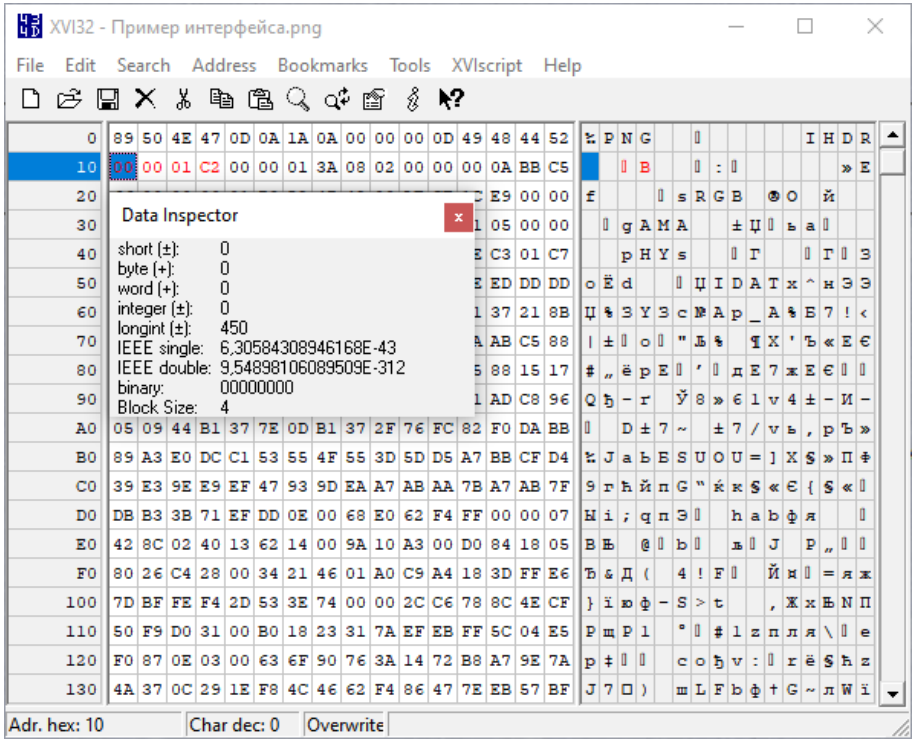
Двоичные данные — это данные в той форме, с которой непосредственно работает процессор.

- Постоянство размера независимо от хранимого значения
- Возможность быстрого чтения данных, так как отсутствует необходимость конвертации
- Необходимо точно знать типы переменных и порядок их следования
- При отсутствии знания о типах, представляет собой бессмысленный набор байт

Двоичные данные

Для работы с двоичными данными существуют специальные шестнадцатеричные редакторы. Они представляют байты в шестнадцатеричном виде, позволяют увидеть печатный текст и декодировать многобайтные числа.





Указатели

Для работы с двоичными данными в языке C++ используются указатели. Указатель хранит адрес первого по счету байта в непрерывном блоке динамической памяти. Задача по хранению размера этого блока ложится на программиста.

```
int* ar;
```

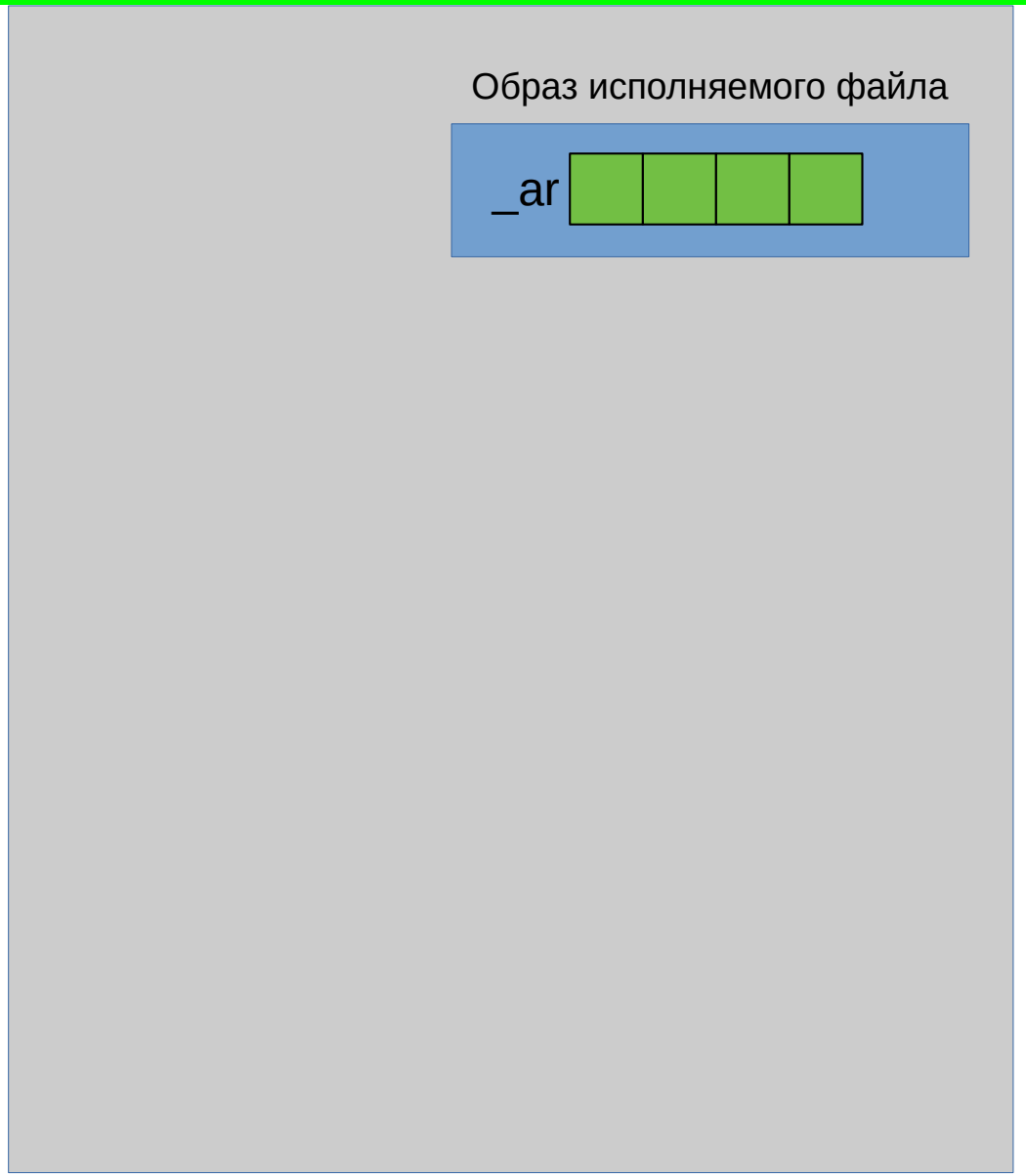
При работе используются также **нетипизированные указатели** (тип **void***). Такой указатель не содержит никакой информации о типе данных, расположенных по хранимому адресу, и должен быть приведен к необходимому типу.

```
int* ar = new int[44];
void* ptr = ar;
int a = ((int*)(ptr))[0];
```

Разрядность операционной системы определяет, сколько бит отводится на хранение адреса и, соответственно, сколько памяти потенциально может адресовать система.

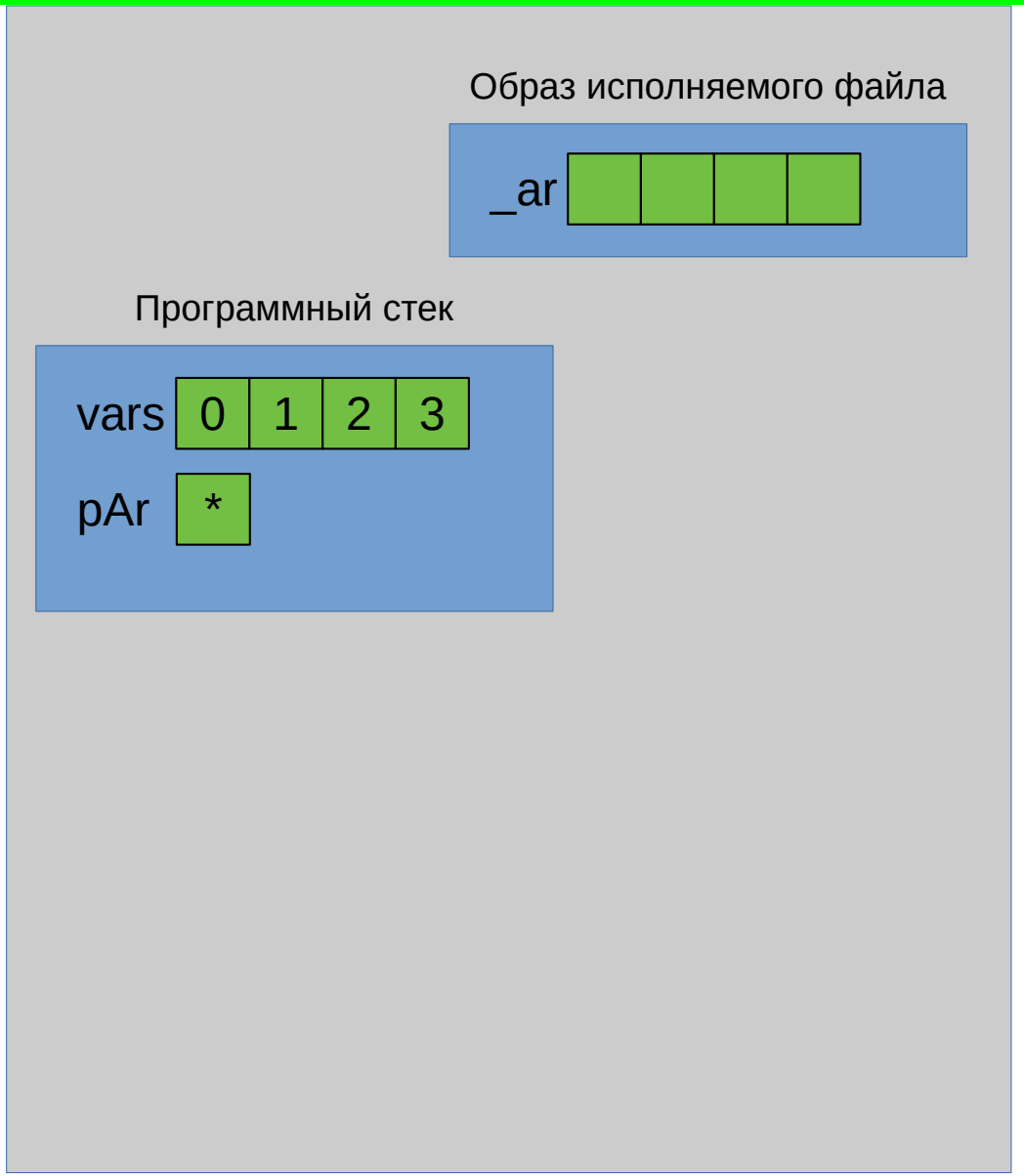
Статическая и динамическая память

```
#include <stdlib.h>
int _ar[4];
int main()
{
  return 0;
}
```



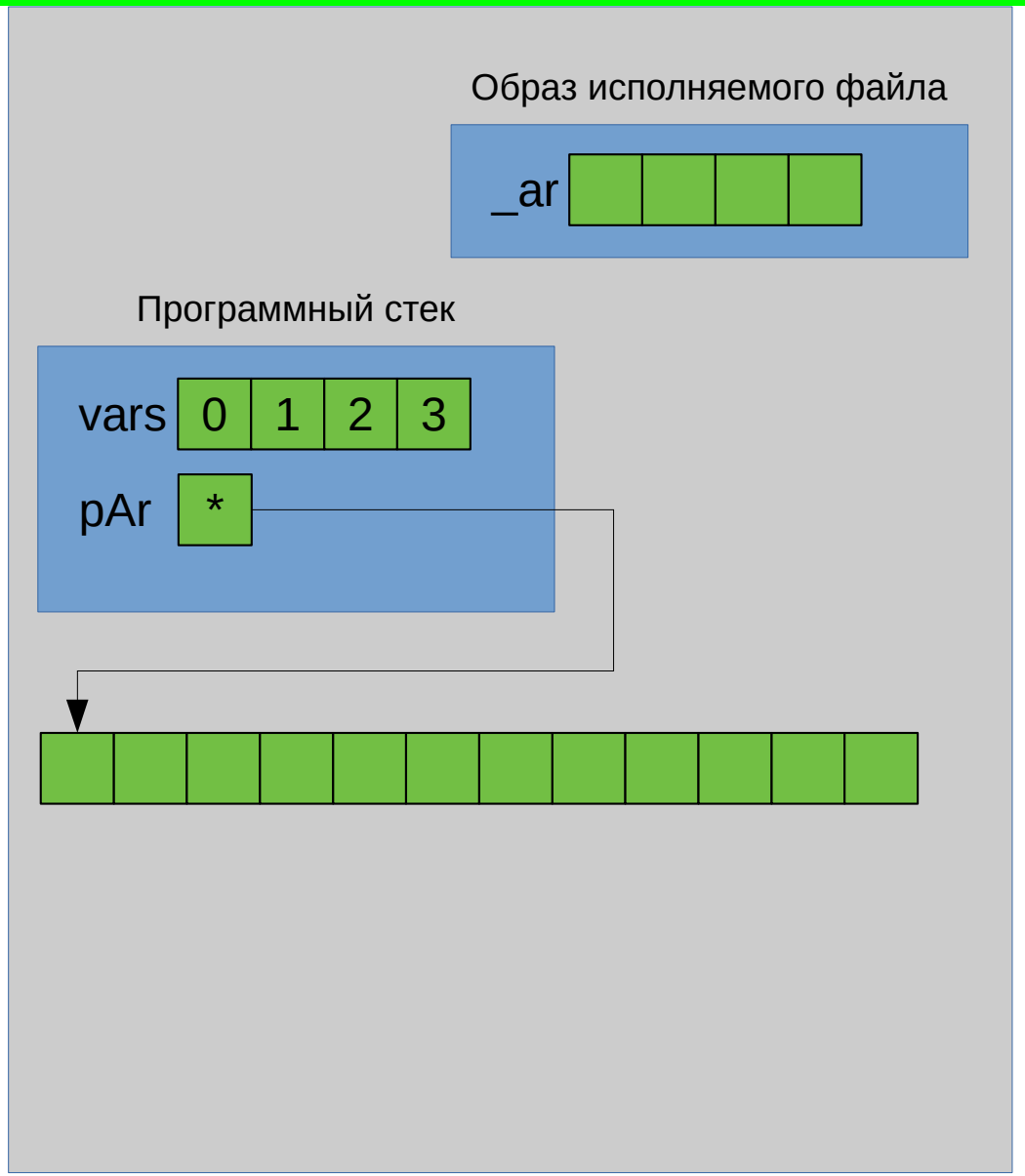
Статическая и динамическая память

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



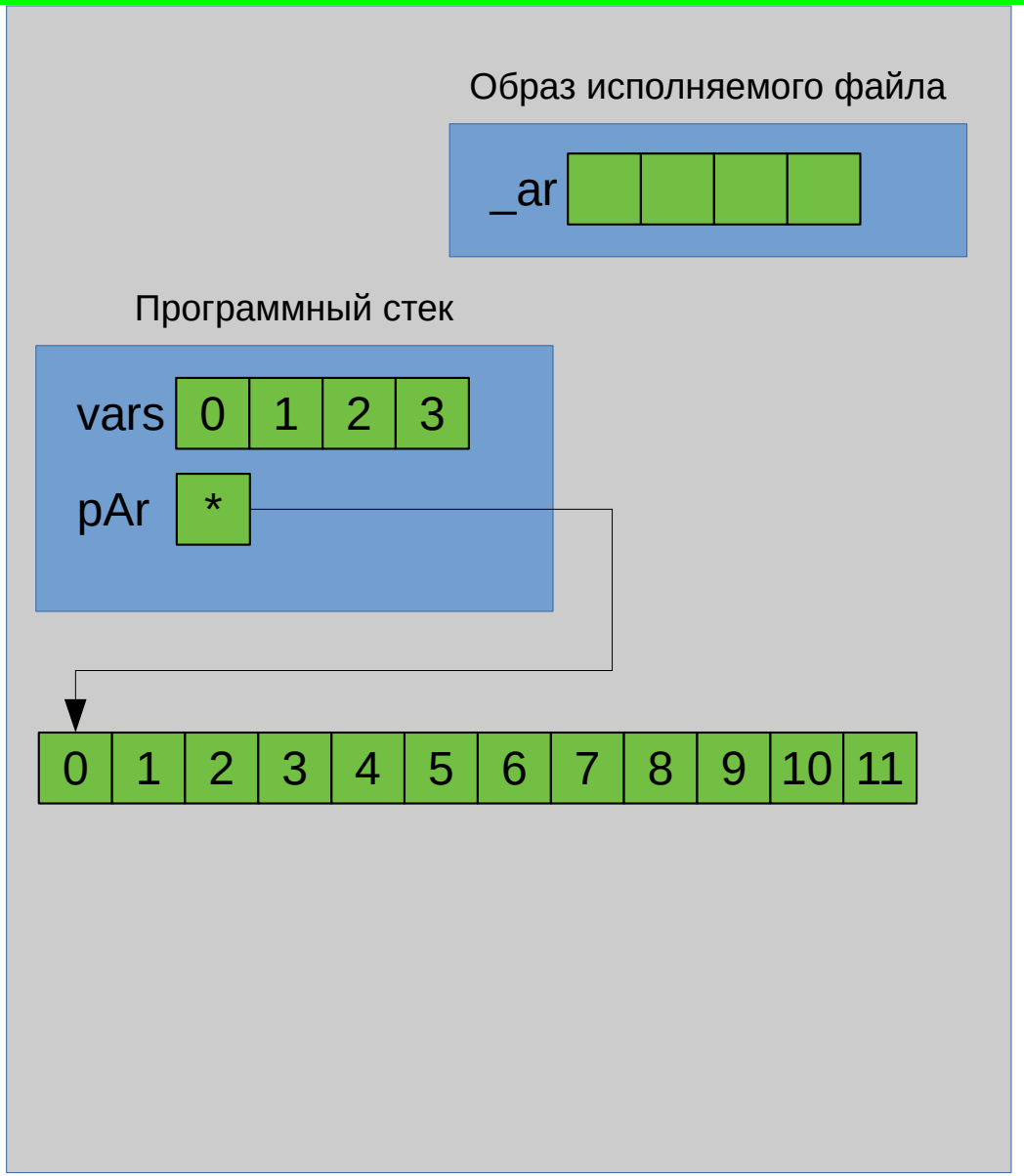
Статическая и динамическая память

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



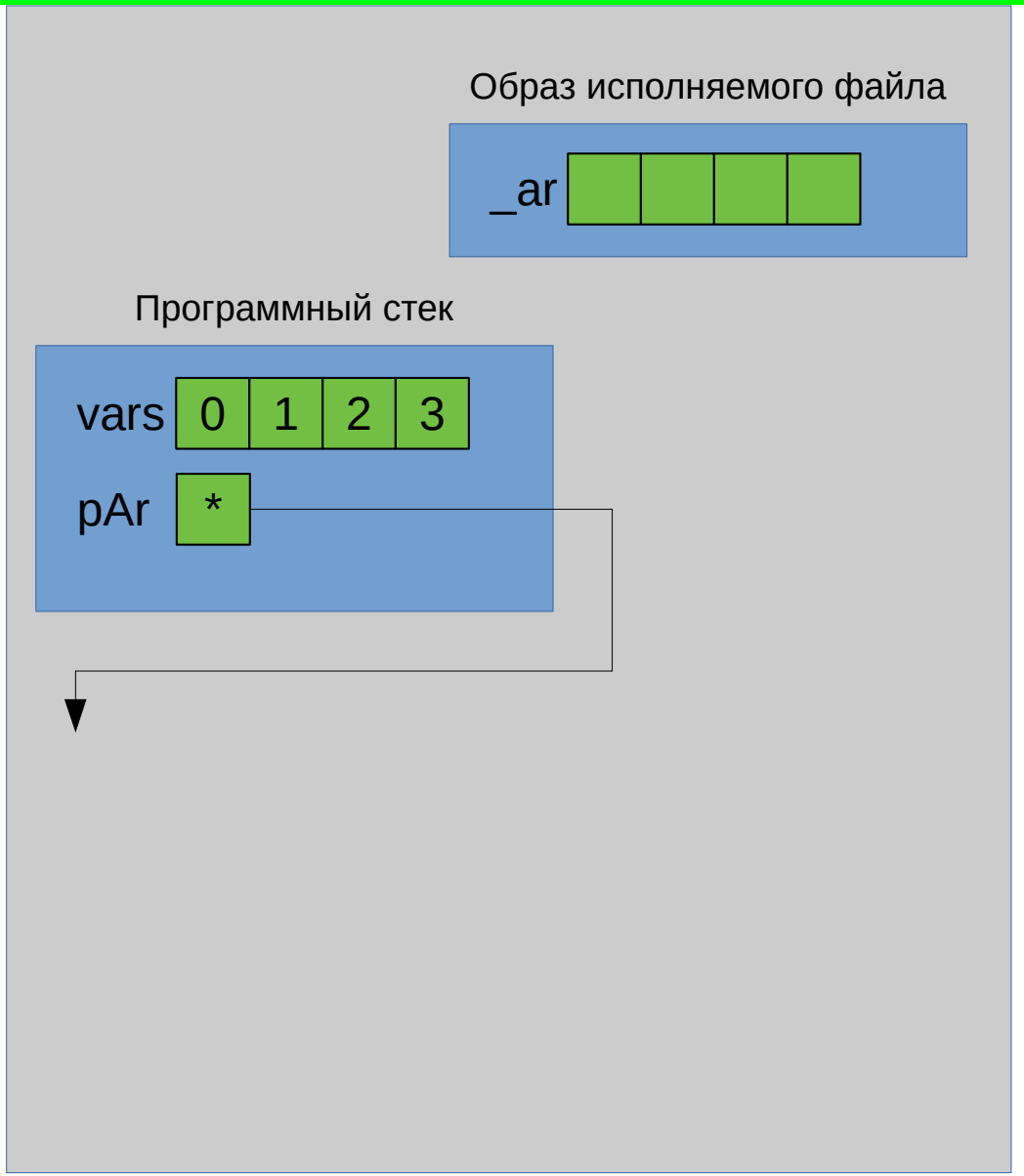
Статическая и динамическая память

```
#include <stdlib.h>
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;
  return 0;
}
```



Статическая и динамическая память

```
#include <stdlib.h>
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; // !!!
  return 0;
}
```

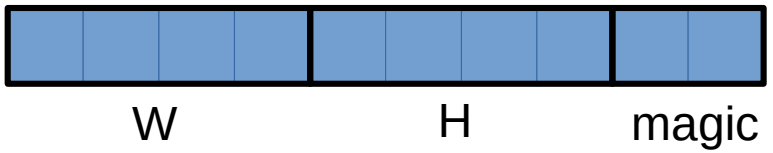


Структуры

Структура позволяет объединить в один тип данных несколько переменных произвольного типа.

Отдельные переменные внутри структуры будут расположены в памяти в порядке их записи. Адрес структуры и адрес ее первого элемента совпадают.

```
struct pxHeader
{
  int W;
  int H;
  char magic[2];
};
```



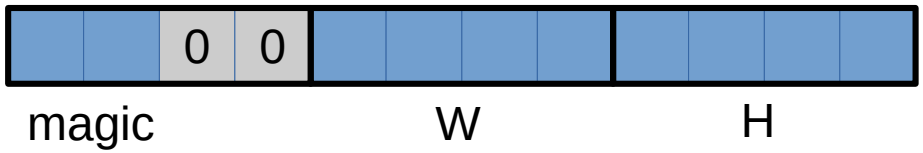
Выравнивание

Для того, чтобы переменная могла быть прочитана или записана за одну машинную инструкцию, ее абсолютный адрес должен быть кратен ее размеру.

При размещении структуры в памяти, к ней могут добавиться нулевые байты для обеспечения этого требования.

При необходимости выравнивание можно отключить.

```
struct pxHeader
{
  char magic[2];
  int W;
  int H;
};
```

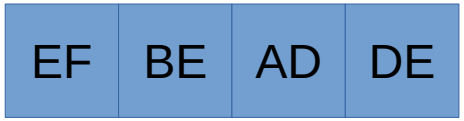


Порядок следования байт

В зависимости от архитектуры различают два варианта записи многобайтных чисел.

```
unsigned int a = 0xDEADBEEF;
```

- **Little-Endian**. Многобайтные числа записываются начиная с младшего байта. Распространено на x86, ARM.



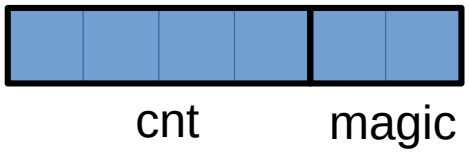
- **Big-Endian**. Многобайтные числа записываются начиная со старшего байта. Распространено в сетевом оборудовании и некоторых микроконтроллерах.

Порядок следования бит (MSB, LSB) никак не связан с порядком байт.

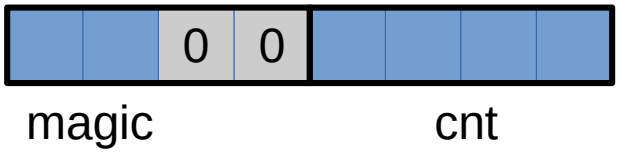
Оператор sizeof

Оператор **sizeof** вычисляет фактическое место, которое занимает в памяти переданный тип или переменная.

```
struct unpadded
{
  int cnt;
  char magic[2];
};
// sizeof(unpadded) == 6
```




```
struct padded
{
  char magic[2];
  int cnt;
};
// sizeof(padded) == 8
```



Двоичные данные в Python

В Python не конкретизируется размер базовых типов в байтах, так как он зависит от интерпретатора.

В Python 2 для представления байтовых массивов используются строки **str**.

В Python 3 строки str работают с текстом, а байтовые массивы представлены отдельными идентичными типами **bytearray** и **bytes**. Они аналогичны строкам, но позволяют изменять отдельные элементы не пересоздавая массив.

При необходимости работы с двоичными данными необходимо использовать модуль **struct**, осуществляющий упаковку и распаковку данных. При упаковке отдельные переменные преобразуются в **bytearray**. При распаковке **bytearray** преобразуется в тьюпл.

Преобразование происходит в соответствии со специальной строкой — **формат**.

Модуль struct

```
import struct
```

Модуль содержит следующие функции:

- **pack(format, args...)** — упаковывает переданный тьюпл или набор аргументов в соответствии с форматом. Возвращает bytearray.
- **unpack(format, bytes)** — распаковывает переданный байтовый массив в соответствии с форматом. Возвращает тьюпл.
- **calcsize(format)** — возвращает размер, занимаемый в памяти байтовым массивом переданного формата.

Формат struct

Формат представляет собой строку, символы которой последовательно определяют переменные, входящие в представляемые двоичные данные.

Первый символ определяет порядок байт и выравнивание.

Буквенный символ соответствует определенному типу.

Число обозначает повторение одинаковых переменных для чисел и количество символов для строк.

При упаковке типы передаваемых данных должны соответствовать типам формата. Например, нельзя упаковать str в качестве числа.

Полное описание формата доступно в референс-документации на модуль struct: https://docs.python.org/3/library/struct.html

Префикс struct

Формат в struct поддерживает следующие префиксы:

@	Порядок байт и выравнивание определяются платформой. Режим по умолчанию.
=	Порядок байт определяется платформой. Выравнивание отсутствует.
<	Порядок байт Little-Endian. Выравнивание отсутствует.
>	Порядок байт Big-Endian. Выравнивание отсутствует.
!	Сетевой порядок байт (Big-Endian). Выравнивание отсутствует.

Типы struct

Обозначение	Тип	Количество байт
c	char	1
b	signed char	1
B	unsigned char	1
h	short int	2
H	unsigned short int	2
i	int	4
I	unsigned int	4
q	long long int	8
Q	unsigned long long int	8
f	float	4
d	double	8
s	char[]	

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

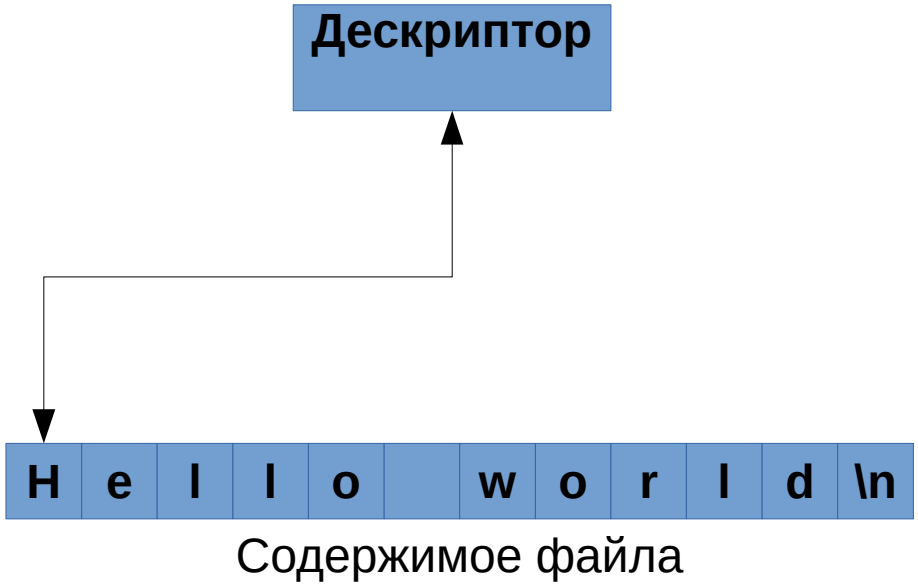
Содержимое файла

Файловый ввод-вывод

Создать дескриптор и массив данных

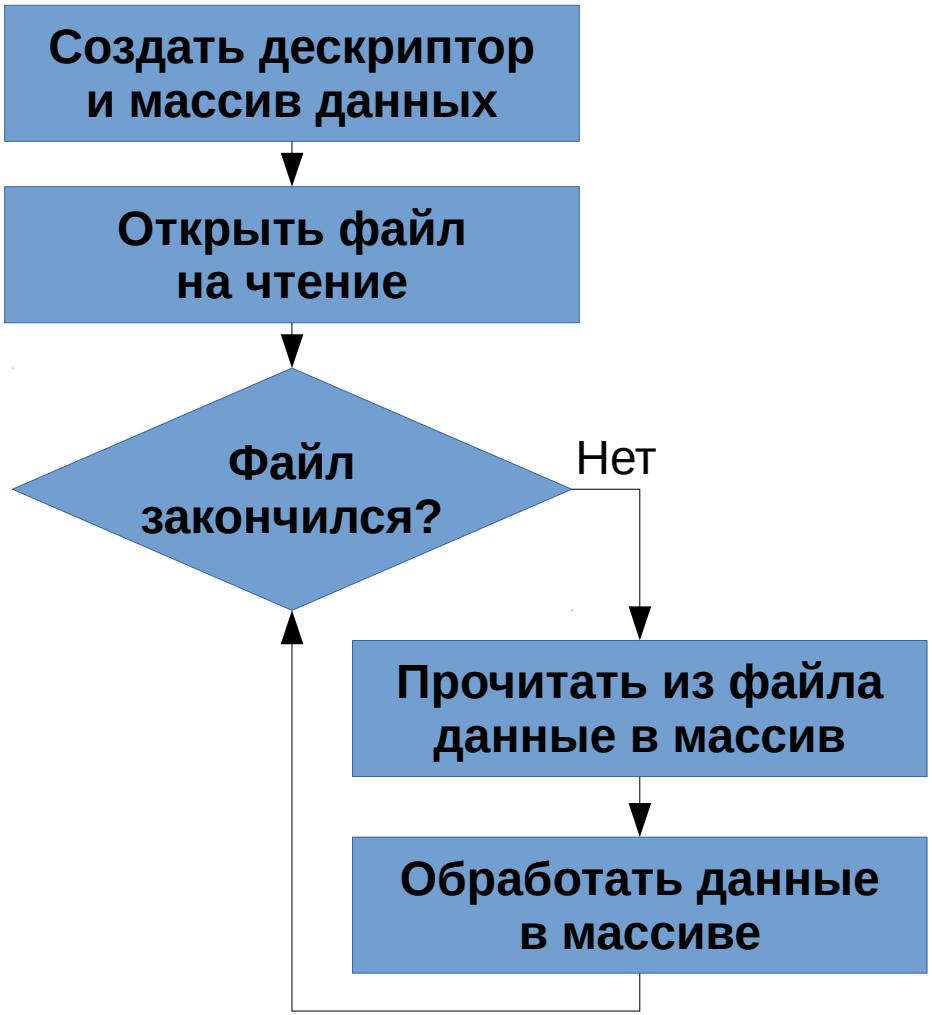
↓

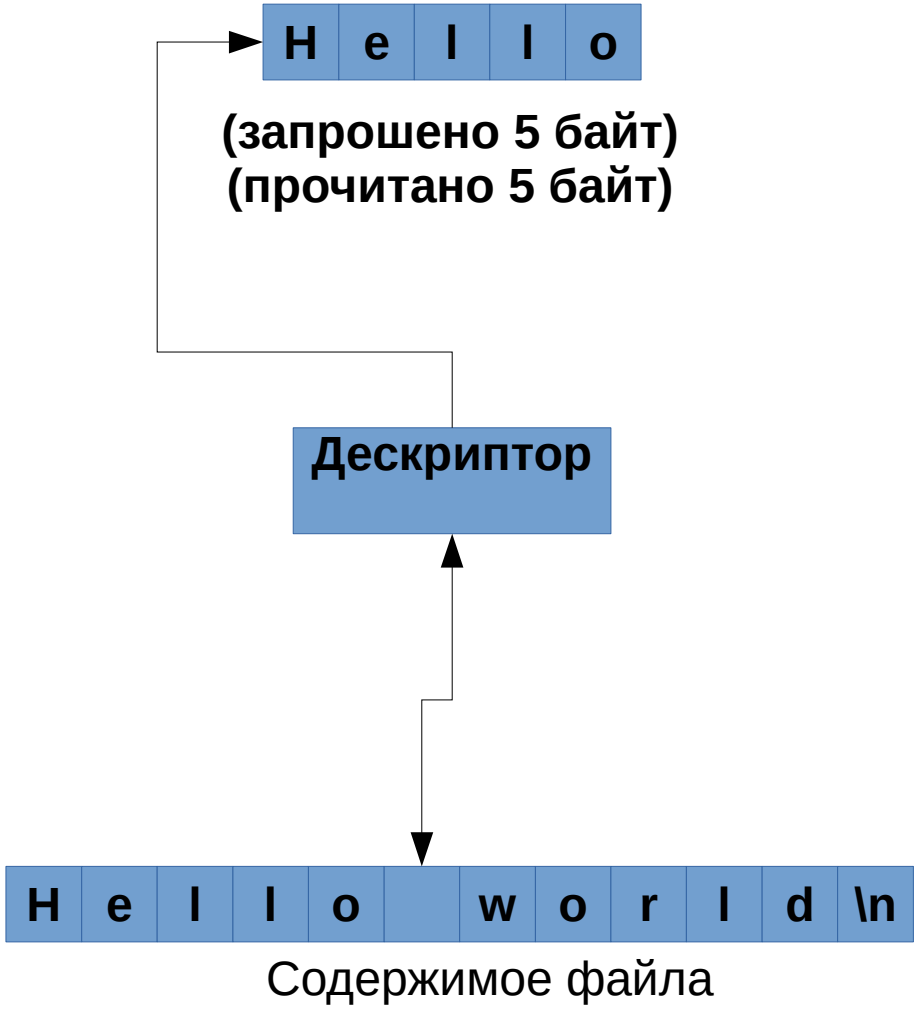
Открыть файл на чтение



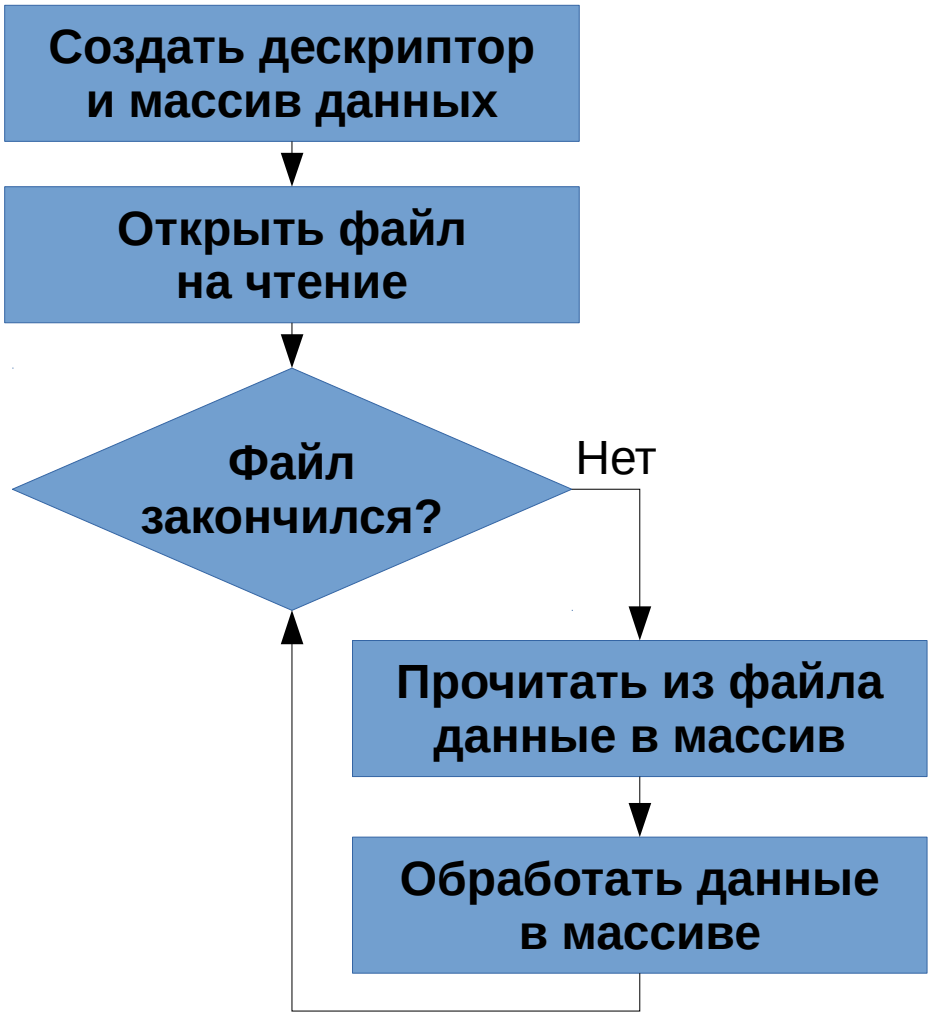
Содержимое файла

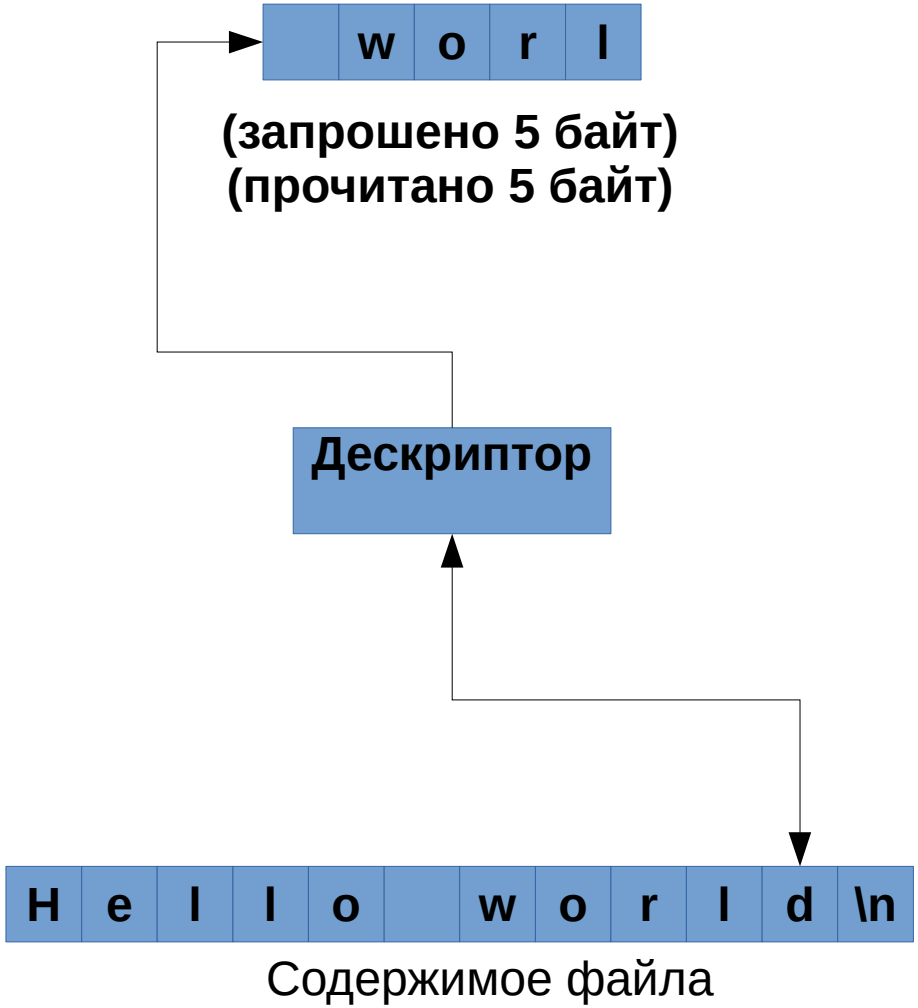
Файловый ввод-вывод



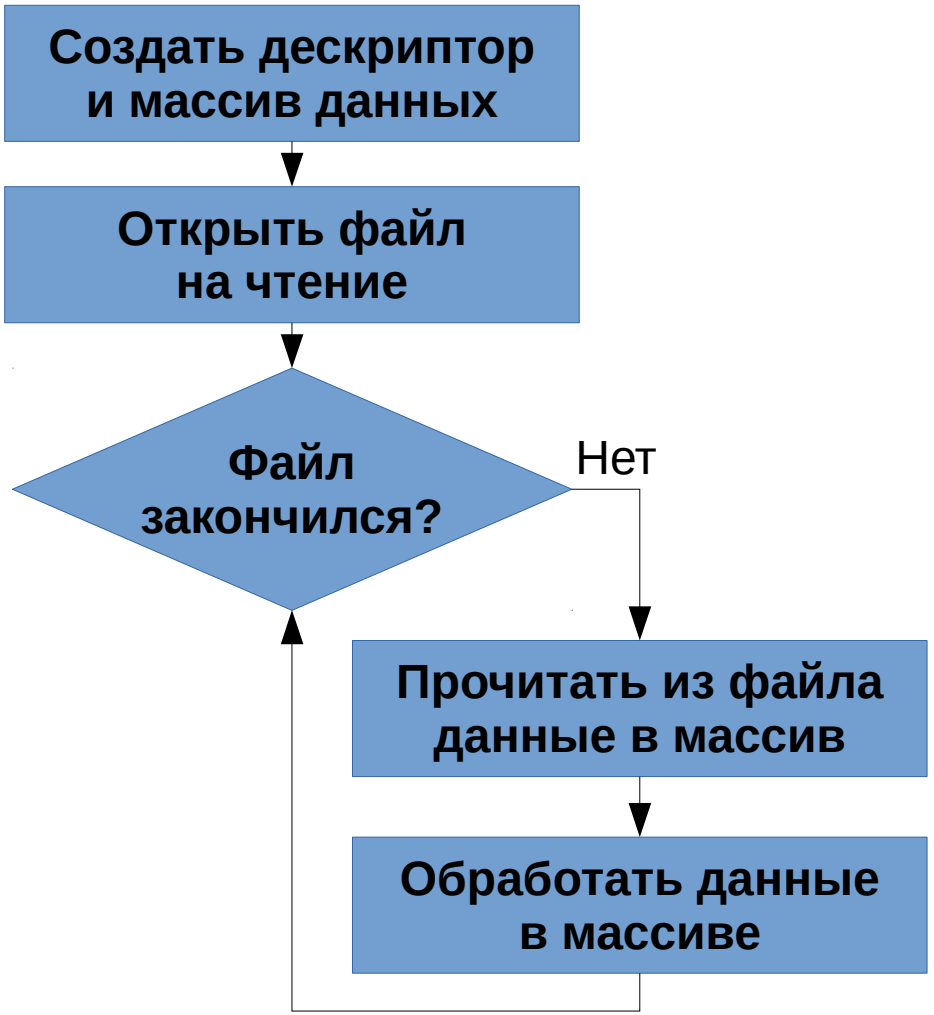


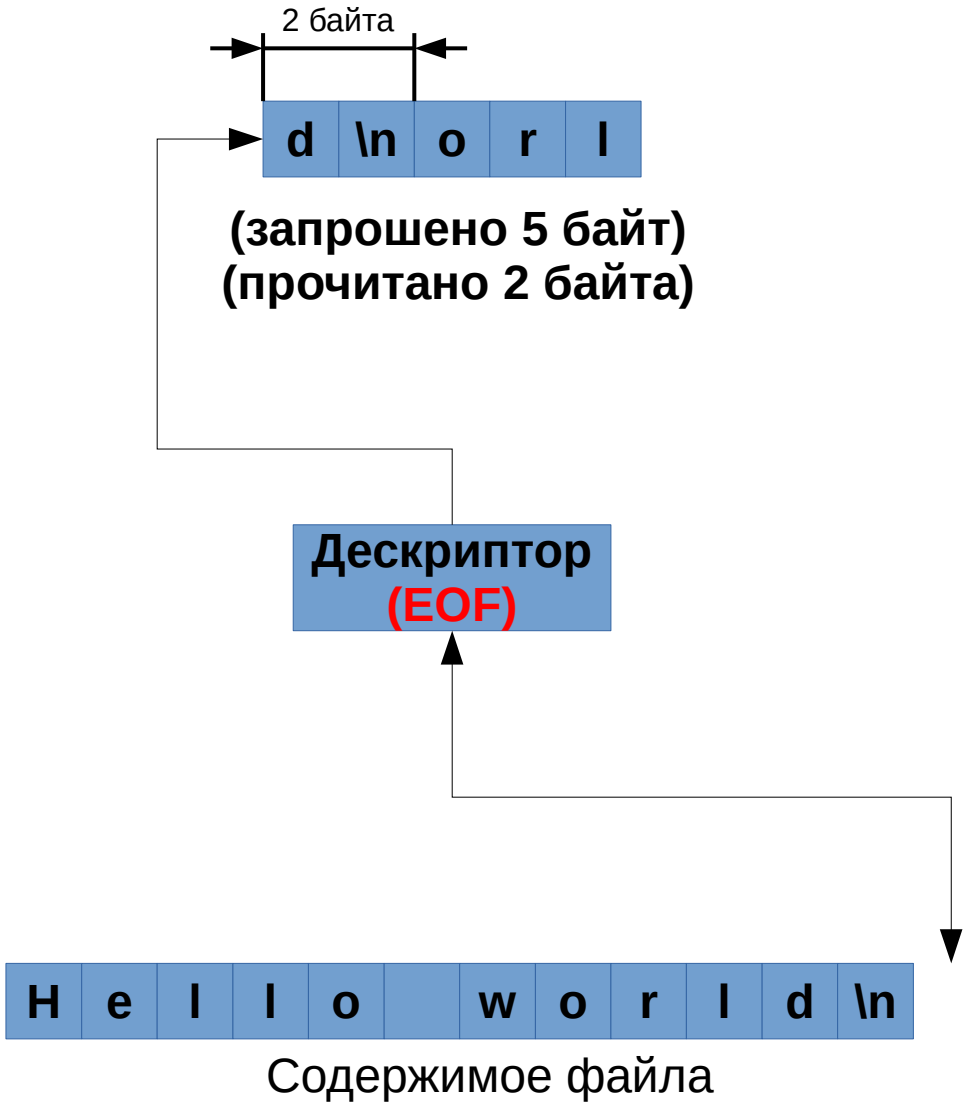
Файловый ввод-вывод



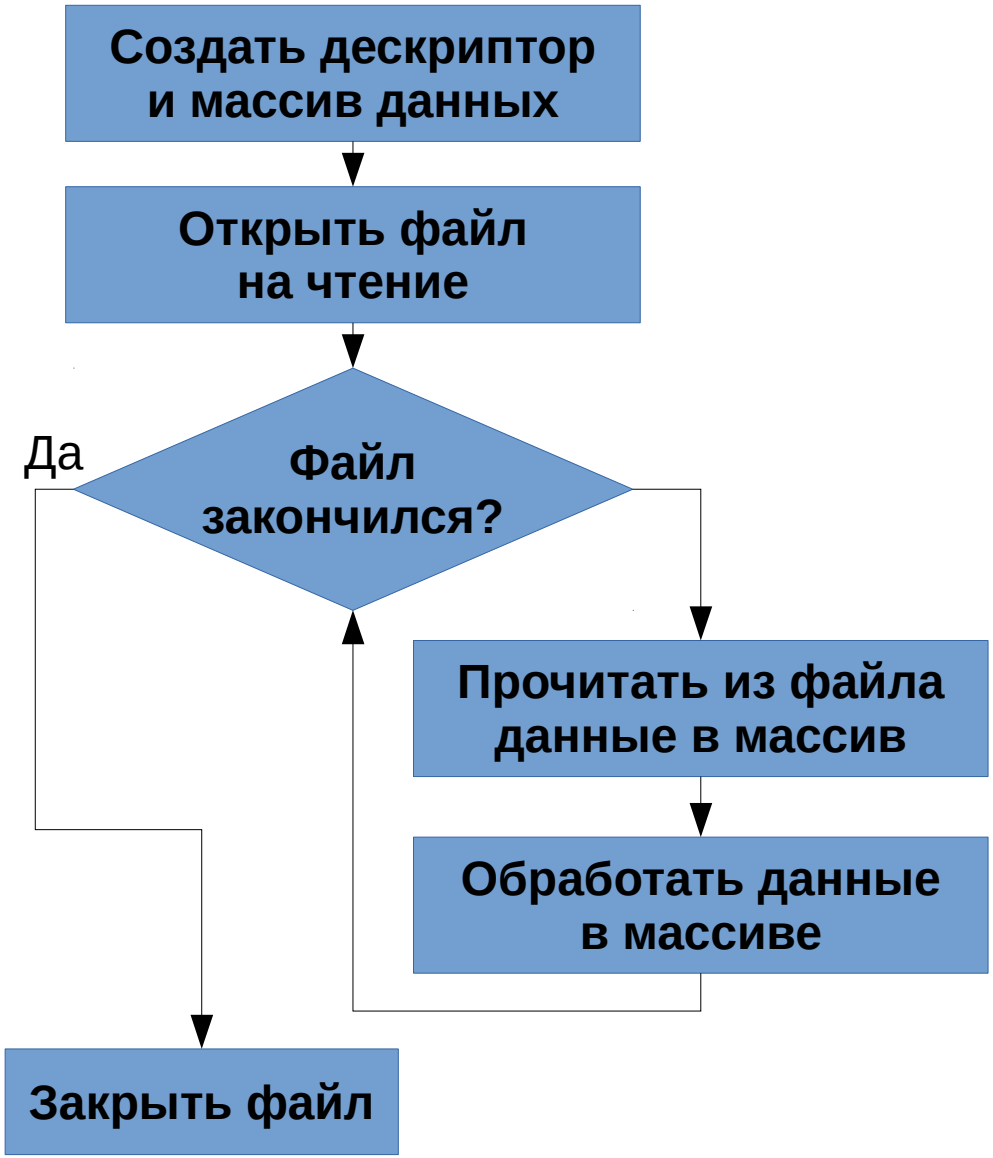


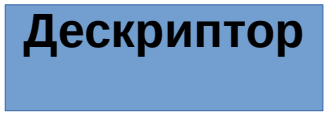
Файловый ввод-вывод





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(in, "%d", &value);
fprintf(out, "%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* F  = fopen("source.txt", "r");
while (!feof(F))
{
  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* F  = fopen("source.txt", "r");
while (!feof(F))
{
  char line[256];
  fgets(line, 256, F);
}
fclose(F);
```

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 (!fs.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
```

Пример двоичных данных

Рассмотрим пример: набор данных локатора за один интервал зондирования:

- Количество каналов дальности: 32 бита, целое беззнаковое
- Стартовая дальность в метрах: вещественное число одинарной точности
- Азимут привода в единицах энкодера: 16 бит, целое беззнаковое
- Угол места привода в единицах энкодера: 16 бит, целое беззнаковое
- Комплексные отсчеты сигнала:
  - Действительная часть, 16 бит, целое знаковое
  - Мнимая часть, 16 бит, целое знаковое

Пример двоичных данных

Реализация на C++

```
struct line
{
    unsigned int   n_channels;
    float          start_range;
    unsigned short az;
    unsigned short el;
};

struct point
{
    short re;
    short im;
};
```

Пример двоичных данных

Запись на C++

```
line L;
L.n_channels = 1024;

point* pts = new point[L.n_channels];
for (int a = 0; a < L.n_channels; ++a)
{
    pts[a].re = 0;
    pts[a].im = 0;
}

FILE* F = fopen("dump.bin", "ab");
fwrite(&L, sizeof(L), 1, F);
fwrite(pts, sizeof(point), L.n_channels, F);
fclose(F);
delete [] pts;
```

Пример двоичных данных

Чтение на C++

```
line L;

FILE* F = fopen("dump.bin", "rb");
fread(&L, sizeof(L), 1, F);

point* pts = new point[L.n_channels];
fread(pts, sizeof(point), L.n_channels, F);

// Обработка данных в pts

fclose(F);
delete [] pts;
```

Пример двоичных данных

Запись на python

```
import struct

line_fmt = "IfHH"
point_fmt = "hh"

line = struct.pack(line_fmt, 1024, 0, 0, 0)
data = bytearray()
for a in range(1024):
    data += struct.pack(point_fmt, 0, 0)

F = open("dump.bin", "ab")
F.write(line)
F.write(b)
F.close()
```

Пример двоичных данных

Чтение на python

```
import struct

line_fmt = "IfHH"
point_fmt = "hh"

F = open("dump.bin", "rb")
line_bin = F.read(struct.calcsize(line_fmt))
line = struct.unpack(line_fmt, line_bin)
n_channels = line[0]

data = []
for a in range(n_channels):
  point_bin = F.read(struct.calcsize(point_fmt))
  data.append(struct.unpack(point_fmt, point_bin))
F.close()
```