Вычислительная техника и программирование

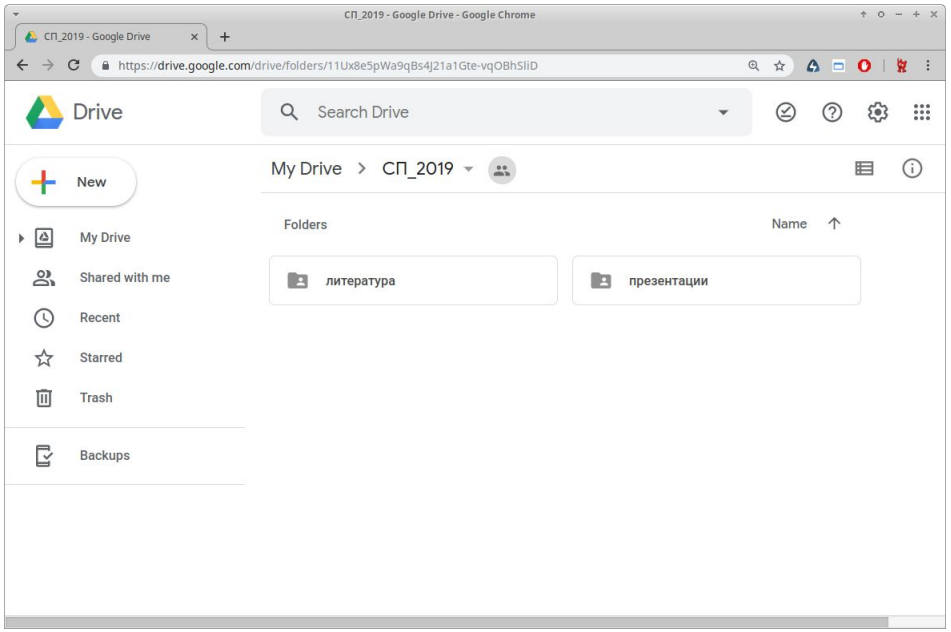
Язык С. Адреса и указатели. Динамическая память

Гирик Алексей Валерьевич

Университет ИТМО
2019

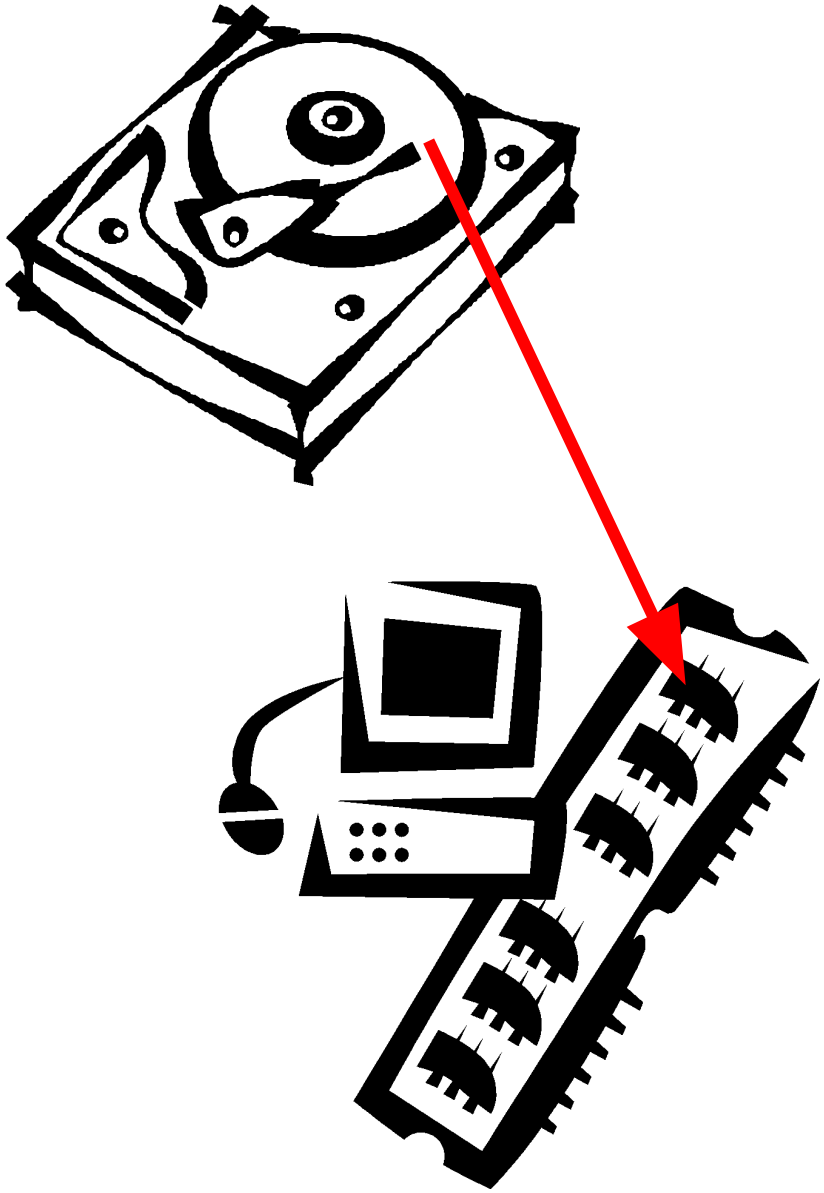
Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
  - shorturl.at/clqG3

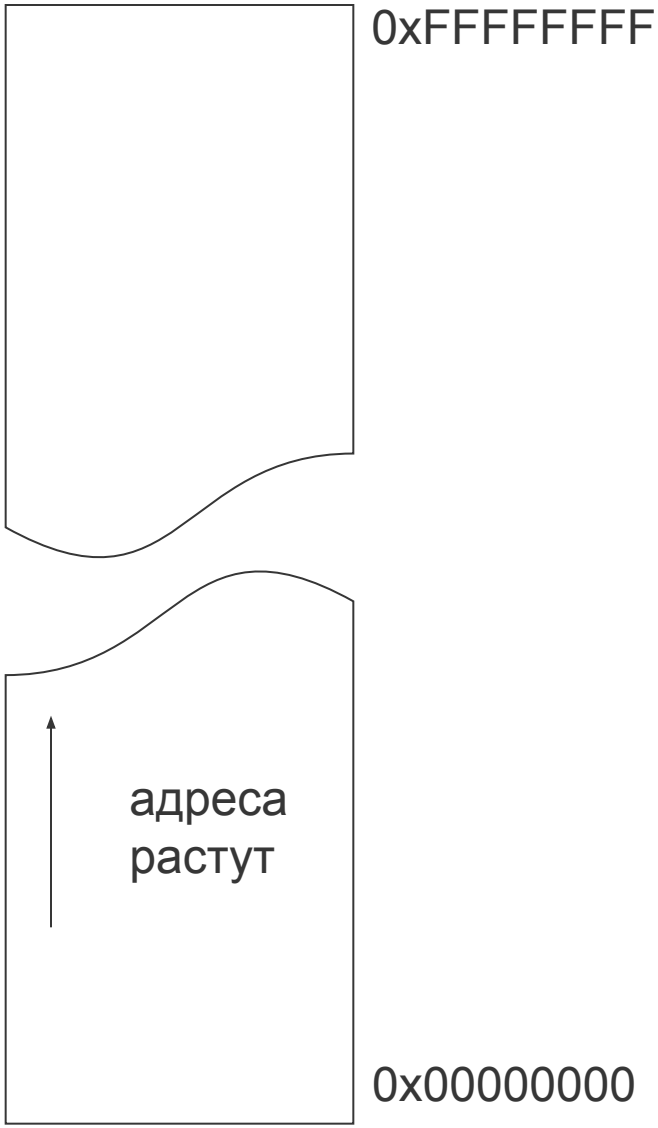


Программа и процесс

- Программа – это
  - исходный код на языке программирования
  - исполняемый файл (скомпилированный и слинкованный код)
  - ... (определений много)
- Процесс – загруженная в память программа



Адресное пространство процесса



- АП процесса (программы) – это память, доступная процессу
  - DOS – все программы делили ~ 640 Кбайт
  - 32 bit Windows, Linux и т.д. – каждой программе доступно 4 Гбайта
  - 64 bit ОС – ?
- АП может делиться на страницы, сегменты и т.д. (зависит от платформы)
  - ограничение доступа

Управление памятью в различных ЯВУ

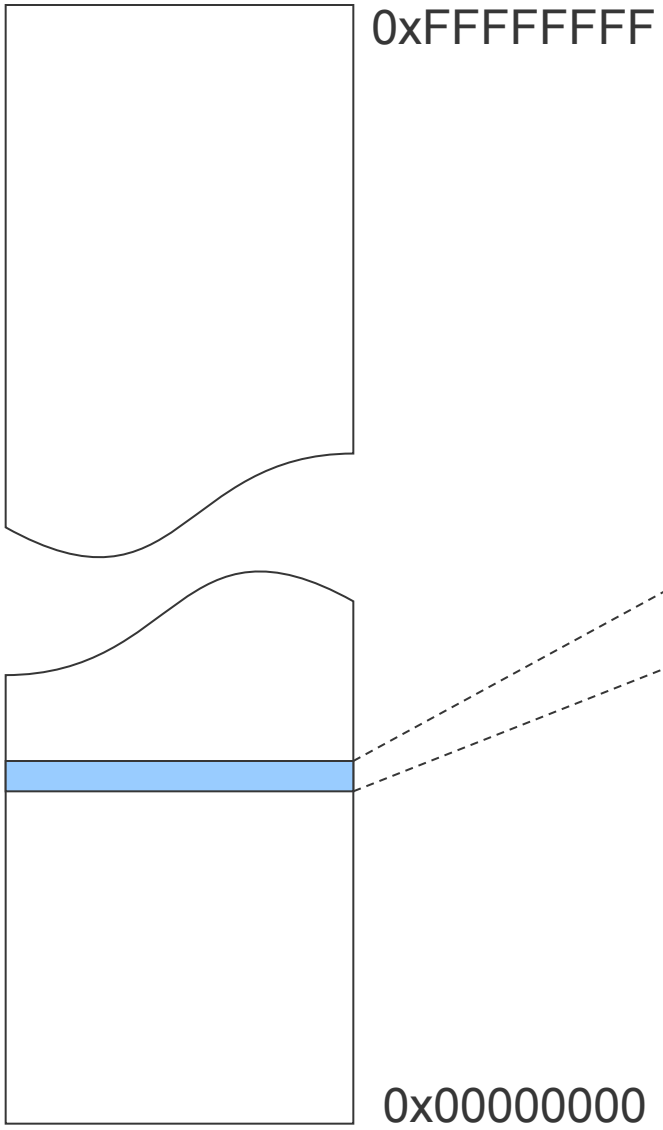
- C/C++, D
- Pascal, Delphi
- Ada
- Smalltalk

языки, допускающие ручное управление памятью

- Java, C#
- Perl, PHP, Python, Ruby
- LISP, Haskell
- …

языки с автоматическим управлением памятью (сборка мусора, подсчет ссылок и т.д.)

Получение адреса переменной



- для получения адреса объекта используется унарная операция **&**

```
int number;
...
printf("%p", &number);

unsigned int addr =
   (unsigned int)&number;
printf("%u", addr);
```

Пример использования &

```
#define MAX_ARR_SIZE 100
...
int arr[MAX_ARR_SIZE];
...

for(int i = 0; i < MAX_ARR_SIZE; ++i)
  printf("%p\n", &arr[i]);
```

Доступ к переменной по имени и по адресу

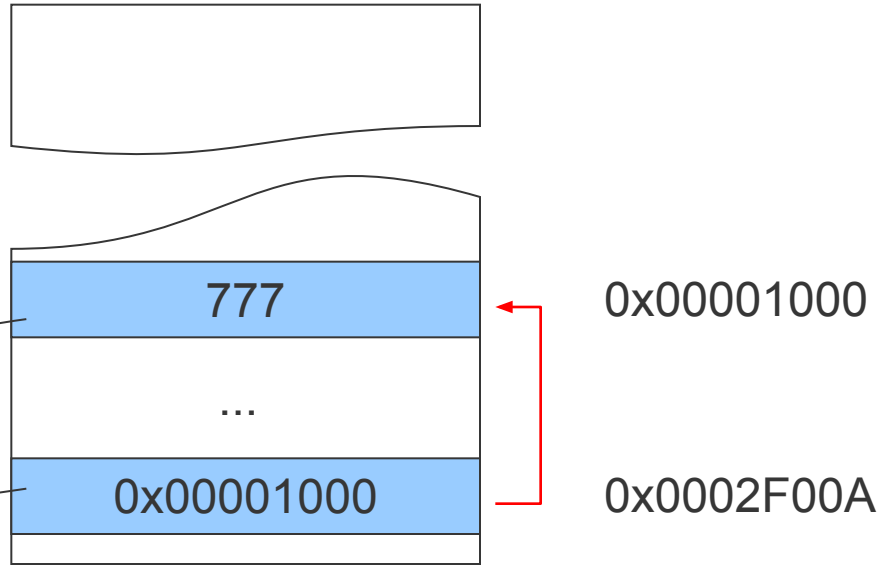
```
long number = 777;

long addr_of_number = (long)&number;
// как с помощью addr_of_number изменить
// number ?
```

Определение указателя

```
тип *имя_переменной = инициализирующее_выражение;
```

Указатель (**pointer**)– это переменная, которая содержит адрес некоторого другого объекта (переменной, массива и т.д.)

```
long    number = 777;
...
long *pointer = &number;
...
```



Разыменование указателя

- для получения доступа к переменной через указатель на нее используется унарная операция *

```
long  number = 777;

long  *pointer = &number;

printf("%ld", number);   // выводит 777

printf("%ld", *pointer);    // выводит 777

*pointer = 42;

printf("%ld", number);   // выводит 42
```

Для чего нужны указатели

- для косвенной адресации данных (чтения и записи по нужному адресу в памяти без использования переменных)
  - для адресации данных, размещенных в динамической памяти (куче) на этапе выполнения программы
- для создания сложных структур данных, элементы которых содержат ссылки друг на друга (списки, деревья, графы, ...)

Указатели и типы

```
long       *lptr;
int      *iptr;
SomeType   *st_ptr;
double   *dptr1;
...
void       *ptr;
void     *another_ptr;

...
void     thing_that_should_not_be;
```

типизированные указатели – связаны с конкретными типами (typed pointers)

нетипизированные указатели – могут указывать на что угодно (generic pointers)

Маленькая тонкость в определении указателей

```
double d = 3.14;
double*  dp1, dp2;

// Вопрос:  сколько указателей здесь
//      определено?
```

Размер указателей

- Что будет выведено на экран?

```
printf("%lu", sizeof(char*));
printf("%lu", sizeof(long*));
printf("%lu", sizeof(double*));
printf("%lu", sizeof(long double*));

printf("%lu", sizeof(void*));
```

Операции с указателями

- получение адреса и разыменование
- инициализация и присваивание
- адресная арифметика
  - сложение
  - вычитание
  - инкремент
  - декремент
  - сравнение

Разыменование указателей

```
double dvar, *dptr;
...
dvar = 3.14;
dptr = &dvar;
...
dptr    = 2.71; //??
*dptr   = 2.71;
(*dptr) = 2.71;
*(dptr) = 2.71;
...
printf("%lf %lf", dvar, *dptr);
```

Нулевой указатель

```
double dvar, *dptr;

...

dptr = NULL;
```

Разыменование указателей

```
double dvar = 0, *dptr;
*dptr = 3.14;    // ??


...
dptr = NULL;
*dptr = 3.14;    // ??
...
dptr = 0;
dptr = 0.0;    //??
dptr = 1;      //??
```

Получение адреса переменных и разыменование указателей

```
double dvar = 4.2, *dptr;

dptr = &dvar;

printf("%lf", *dptr);

printf("%lu", *&*dptr);
```

Разыменование указателей на void

- разыменование указателей типа void – синтаксическая ошибка

```
double dvar = 4.2, *dptr;
dptr = &dvar;

void *vptr;
vptr = &dvar;

printf("%p", vptr);
printf("%lf", *vptr);      //??
```

Инициализация и присваивание указателей

```
double  dvar, *dptr = &dvar;
int     ivar, *iptr = &ivar;
void    *vptr;
...
dptr = 0;
dptr = NULL;
...
vptr = dptr;
vptr = 0x00002FFF;          //??
vptr = (void*)0x00002FFF;
iptr = vptr;              //??
```

Присваивание указателей

```
float  fvar = 3.14f, *fptr = &fvar;
int    *iptr;
...
printf("%f", fvar);

iptr = fptr;            //??
iptr = (int*)fptr;
*iptr = 777;

printf("%lu", fvar);
```

Сравнение указателей

```
unsigned long ul = 0x12345678;

char   *cptr = (char*)&ul;
short  *sptr = (short*)&ul;
long   *lptr = (long*)&ul;

if(cptr == sptr) printf("Equals!");
if(lptr != NULL) printf("Not null!");

printf("%d %d %ld", *cptr, *sptr, *lptr);
```

Сравнение указателей

```
int  arr[100];
...
int  *ptr1 = &arr[1],
   *ptr2 = &arr[10];

if(ptr1 > ptr2) { ... }

if(*ptr1 > *ptr2) { ... }

if(*ptr1 > ptr2) { ... }
```

Сложение и вычитание указателей

```
int   arr[10], diff = 0;
...
int   *ptr1 = &arr[0],
   *ptr2 = &arr[2],
   *ptr3 = &arr[9];

ptr2 = ptr1 + ptr2;       //??
ptr2 = ptr3 - ptr1;       //??
diff = ptr3 - ptr1;       // diff == ?
diff = (int)ptr3 - (int)ptr1; // diff == ?
```

Инкремент и декремент указателей

```
unsigned long ul[] =
{ 0x12345678, 0x9ABCDEF0, 0x0 };

unsigned char *cptr =
  (unsigned char*)&ul[0];


while(*cptr != 0)
  printf("%d ", *cptr++);
```

Инкремент и декремент указателей

```
unsigned long ul = 0x12345678;

char     *cptr = (char*)&ul;
short    *sptr = (short*)&ul;

cptr++; cptr++;
sptr++;

if(cptr == sptr) { ... }
if(*cptr == *sptr) { ... }

// Всегда ли из равенства указателей следует
// равенство значений ? Верно ли обратное?
```

Увеличение и уменьшение указателей

```
unsigned long ul = 0x12345678;

char   *cptr = (char*)&ul;

cptr += 3;

printf("%d", *cptr);
```

Арифметические операции с указателями на void

- в C разрешены, в C++ запрещены

```
void *vptr = (void*)0x00002FFF;

vptr++;

printf("%p\n", vptr);// 0x00003000

vptr -= 1;

printf("%p\n", vptr);// 0x00002FFFF
```

Доступ к невыровненным данным

```
long l = 0x12345678, *lp = &l;

char *cp = &l;
cp += 3;
lp = (long*)cp;

*lp = 0;    // На некоторых процессорах
        // приведет к исключению
```

Указатели на указатели

```
int  n = 10;

int  *p = &n;

int  **pp = &p;
int  ***ppp = &pp;
int  ****pppp = &ppp;

printf("%d ", ****pppp); // выводится 10
```

Страшная правда о массивах

```
int  arr[10];
```

```
printf("%p", arr);
printf("%p", &arr);
printf("%p", &arr[0]);
```

Имя массива – это неизменяемый (константный) указатель на первый элемент массива!

```
arr++; // запрещено
```

Страшная правда об операции [ ]

Операция индексации – это сложение указателя со смещением (индексом) и последующее разыменование!

```
int  arr[10] = {0};

arr[5] = 42;

*(arr + 5) = 42;

5[arr] = 42;
```

Страшная правда об операции [ ]

```
int *ptr = &5[arr];

ptr[1] = 2[ptr] = *(ptr + 3) = 42;

ptr++;

ptr[-1] = -2[ptr] = *(ptr - 3) = 42;  // ??
```

Just for fun

```
int m[] = { 10, 20, 30, 40 }, j = 1;

printf("%d ", m[j]);
printf("%d ", *(m + j++));
printf("%d ", *(++j + m));
printf("%d ", j[m]);
printf("%d ", *(j-- + m));
printf("%d ", j--[m]);
printf("%d ", *(--j + m));
printf("%d ", --j[m]);
```

Страшная правда об операции [] и многомерные массивы

```
int m[3][3] = {0};


m[1][2] = 11;
(m[1])[2] = 11;
(*(m+1))[2] = 11;
*( *(m+1) + 2 ) = 11;

printf("%d ", m[1][2]);
```

Указатели на массивы

```
int a[4] = {0, 1, 2, 3};

int *p = a;
printf("%d %d", a[1], p[1]);


int m[3][3] = {{}, {0, 11, 22}, {}};

int **p = m;              // НЕ компилирутеся
int **p = (int**)m;       // компилируется, но
                  // НЕ работает
int *p[3]   = m;          // НЕ компилируется
int (*p)[3] = m;          // НЕ компилируется

int (*p)[3] = (int (*)[3])m; // работает!

printf("%d %d", m[1][2], p[1][2]);
```

Указатели на массивы

```
int *p[4];  // массив из четырех
        // указателей на int

int (*p)[4];  // указатель на массив
        // из четырех int
```

Задача

- Что будет выведено на экран?

```
#include <stdio.h>

int main() {
    int x[5];

    printf("%p\n", x);
    printf("%p\n", x + 1);
    printf("%p\n", &x);
    printf("%p\n", &x + 1);

    return 0;
}
```

Приоритеты унарных операций

```
int i[] = {0, 10, 20, 30}, *p = &i[2];

printf("%d ", *&i[2]);
printf("%d ", ++*&i[2]);
printf("%d ", *p++);
printf("%d ", *p);
printf("%d ", ++*p);
printf("%d ", *--p);
printf("%d ", ++*--p);
```

Приоритеты унарных операций

```
int i[] = {0, 10, 20, 30}, *p = &i[2];

printf("%d %d %d %d %d %d %d",
  *&i[2],
  ++*&i[2],
  *p++,
  *p,
  ++*p,
  *--p,
  ++*--p);
```

Задача

- Ввести с клавиатуры элементы массива, выполнить сортировку и вывести отсортированный массив на экран

Ограничения стандартных (встроенных) массивов

- размер задается на этапе компиляции и не может изменяться в ходе выполнения программы (кроме C99 и C++ extentions)

```
int arr[100];
```

- размер встроенного массива не может быть больше ограничения, накладываемого компилятором

```
int arr[1000000000];  // possible compiler error
```

```
error C2148: total size of array must not exceed 0x7fffffff bytes
```

- размер встроенного массива не может быть больше размера стека, если размещается в стеке

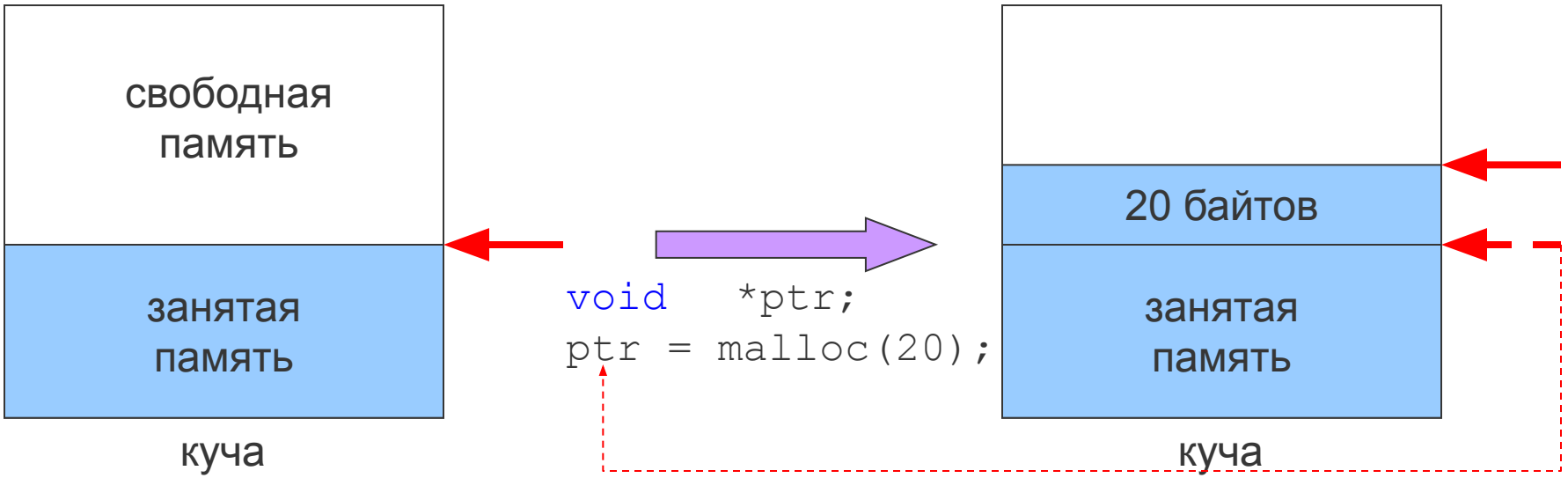
```
int arr[1000000];  // runtime error
```

Куча

- **Куча** (**heap**) – область в адресном пространстве процесса, память из которой программа может получать на этапе выполнения
- Пользоваться кучей имеет смысл, если:
  - на этапе компиляции неизвестно, сколько памяти нужно зарезервировать
  - память нужна только в определенные моменты выполнения программы

Как получить память из кучи?

```
#include <stdlib.h>

void* malloc(size_t bytesTotal);
void* calloc(size_t numOfElements,
             size_t sizeOfElem);
```



Как освободить память?

```
#include <stdlib.h>

void free(void* ptrToMemory);
```

```
  void *ptr;
  ptr = malloc(20);

  // делаем что-то полезное с ptr

  free(ptr);
```

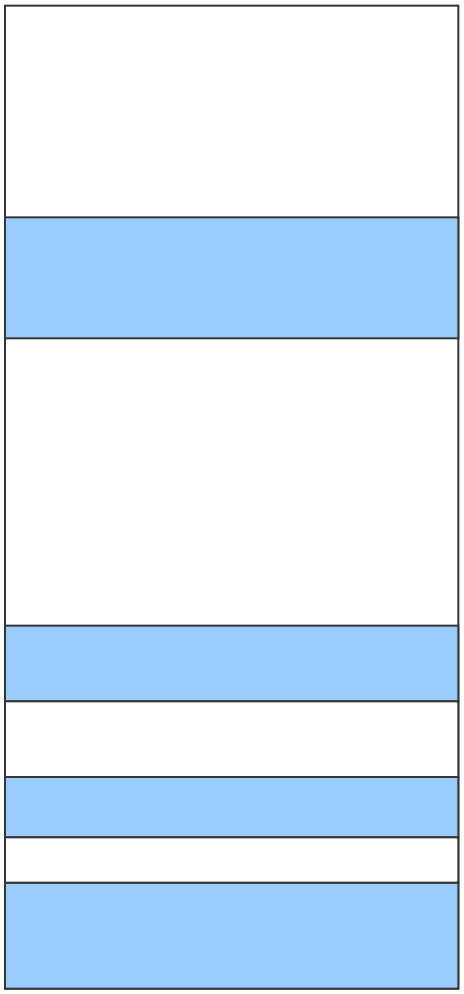
Правильный способ выделения и освобождения памяти из кучи

```
#include <stdlib.h>

int*ptr = (int*) malloc(20*sizeof(int));

if(ptr != NULL) {
   // делаем что-то полезное с памятью
   ...
   free(ptr);
}
else {
   // сообщаем об ошибке
   printf("Error: Could not allocate memory\n");
   exit(-1);
}
```

sizeof() и динамическая память

```
int arr[20];

printf("%d", sizeof(arr));  // 80

int  *ptr = (int*) malloc(20*sizeof(int));

if(ptr != NULL){
   printf("%d", sizeof(ptr)); // 4, не 80
}
```

Как на самом деле работает куча

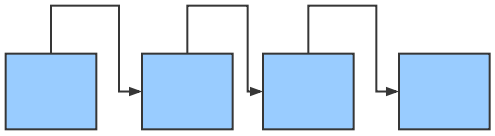


стандартная библиотека C

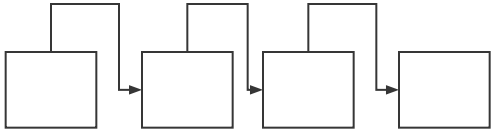
…

диспетчер кучи

```
void *malloc(…) { …
}
void free(…) { … }
```



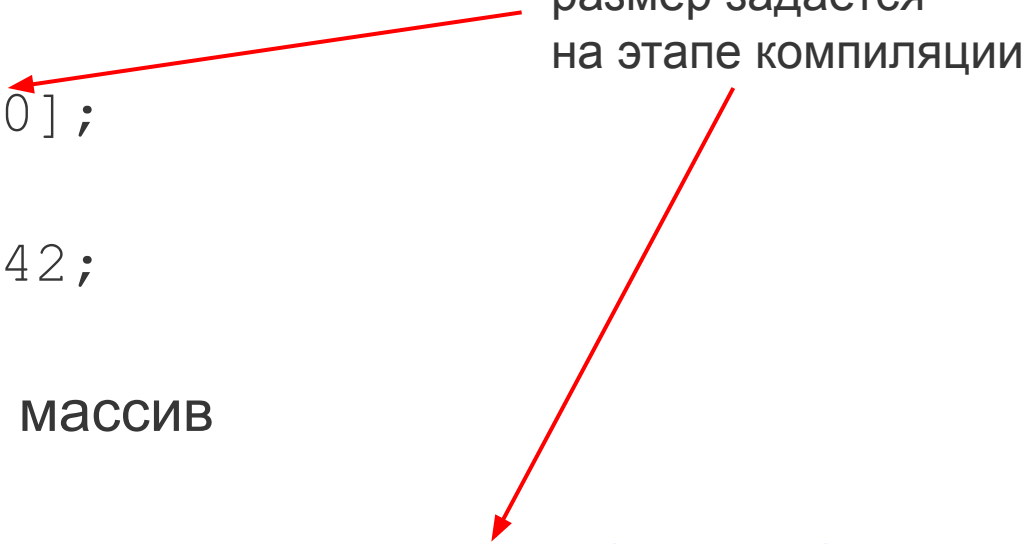
список занятых блоков



список свободных блоков

Динамические массивы

- статический массив




```
int array1[100];
...
array1[10] = 42;
```

- динамический массив

```
int *ptr = (int*) malloc(100*sizeof(int));
...
ptr[10] = 42;
...
free(ptr);
```

Динамические массивы

- VLA массив

```
int num;
scanf("%d", &num);
int array1[num];
...
array1[10] = 42;
```

размер задается
на этапе выполнения

- динамический массив

```
int num;
scanf("%d", &num);
int *ptr = (int*) malloc(num*sizeof(int));
...
ptr[10] = 42;
...
free(ptr);
```

Как перераспределить память?

```
#include <stdlib.h>

void* realloc(void *ptrToMemory,
        size_t newSize);
```

Как перераспределить память?

```
int n;
scanf("%d", &n);
double *p =
  (double*)malloc(n*sizeof(double));
...
scanf("%d", &n);
p = (double*)realloc(p, n*sizeof(double));
...
```

Наиболее частые ошибки при работе с динамической памятью

- утечки памяти (memory leaks)
- отсутствие проверки успешного выделения памяти, и, как следствие, использование неинициализированных указателей (wild pointers)
- повторное освобождение памяти
- использование участка памяти после того, как он был освобожден (dangling pointers)

Полезные функции для работы с памятью

```
#include <string.h>

void* memset(void *dest, int c, size_t count);

void* memcpy(void *dest, const void *src, size_t count);

void* memmove(void *dest, const void *src, size_t
  count);

intmemcmp(const void *buf1, const void *buf1, size_t
  count);

void* memchr(const void *buf1, int c, size_t count);
```

Функция memset()

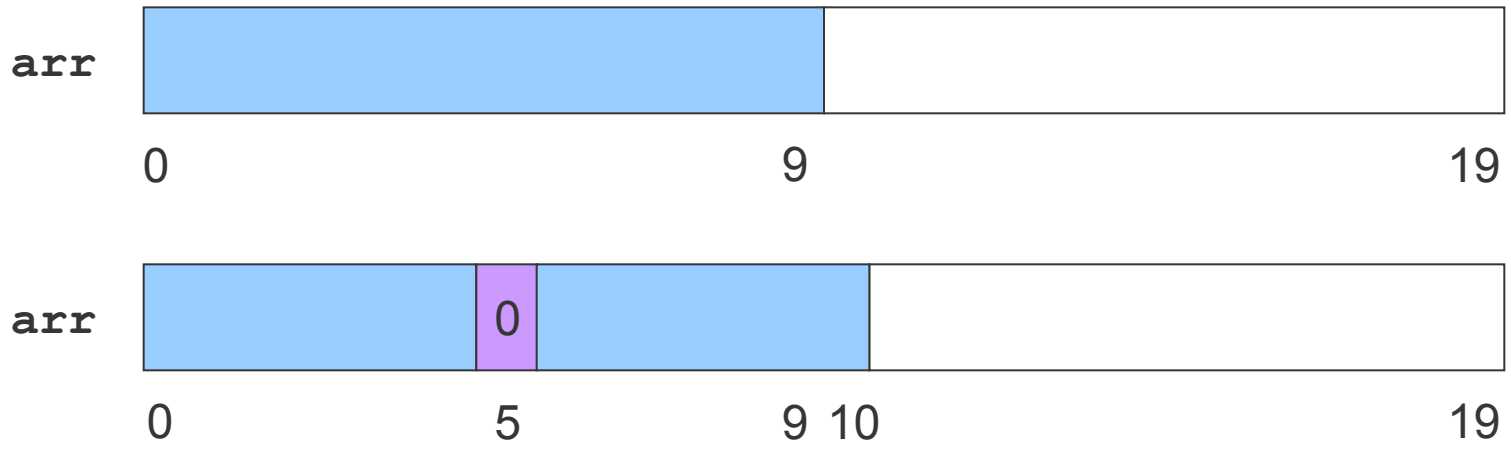
```
void* memset(void *dest, int c, size_t count);

int arr[100];
...
memset(arr, 'a', 100);          //??
memset(arr, 'a', 100 * sizeof(int));


int *arr = (int*)malloc(100 * sizeof(int));

memset(arr, 0, 100 * sizeof(int));
memset(arr, 0xAA5500, 100 * sizeof(int));
...
free(arr);
```

Функция memcpy()

```
void* memcpy(void *dest, const void *src,
  size_t count);


int arr1[10], arr2[10];

for(int i = 0; i < 10; ++i)
  arr1[i] = rand();

memcpy(arr2, arr1, 10 * sizeof(int));
```

Задача

- вставить 0 в середину массива




```
int arr[20];
// или
int *arr = (int*)malloc(20 * sizeof(int));
```

Задача

- Вставить ноль в середину массива

```
int *arr = (int*)malloc(20 * sizeof(int));

for(int i = 0; i < 10; ++i)
  arr1[i] = rand();

for(int i = 10; i > 5; --i)
  arr[i] = arr[i-1];

arr[5] = 0;
```

Задача

- Вставить ноль в середину массива

```
int *arr = (int*)malloc(20 * sizeof(int));

for(int i = 0; i < 10; ++i)
   arr1[i] = rand();

memcpy(arr + 6, arr + 5, 5 * sizeof(int)); //!!

arr[5] = 0;
```

Функция memmove()

```
void* memmove(void *dest, const void *src, size_t
   count);


int *arr = (int*)malloc(20 * sizeof(int));

for(int i = 0; i < 10; ++i)
   arr1[i] = rand();

memmove(arr + 6, arr + 5, 5 * sizeof(int)); //OK

arr[5] = 0;
```

Функция memcmp()

```
int memcmp(const void *buf1, const void *buf2, size_t
   count);
```

Возвращаемое значение

- > 0, если `buf1` > `buf2`
- = 0, если `buf1` = `buf2`
- < 0, если `buf1` < `buf2`

```
intarr1[] = {10, 20, 30},
    arr2[] = {10, 20, 31},
    arr3[] = {0, 10, 21, 30};

printf("%d ", memcmp(arr1, arr2, 3 * sizeof(int)));
printf("%d ", memcmp(arr1, arr2, 2 * sizeof(int)));
printf("%d ", memcmp(arr1, arr3 + 1, 2 * sizeof(int)));
```

Функция memcmp()

```
int  arr1[] = {255, 0, 0},
   arr2[] = {256, 0, 0};

printf("%d ",
  memcmp(arr1, arr2, 3 * sizeof(int)));
```

Функция memchr()

```
void* memchr(const void *buf1, int c, size_t
  count);
```

```
char letters[] = {'a', 'b', 'c', 'd'};

char *p = (char*)memchr(letters, 'c', 4);

printf("%sound", p == NULL ? "Not f" : "F");
```

Как динамически создать двумерный массив?

- даны
  - количество строк N и столбцов M матрицы
  - тип элементов
- требуется
  - динамически выделить память для матрицы
  - разработать схему обращения к элементам матрицы

Двумерные массивы

```
#define N 3
#define M 3

int  mtx[N][M];
...
mtx[2][1] = 42; // третья строка,
          // второй столбец
```

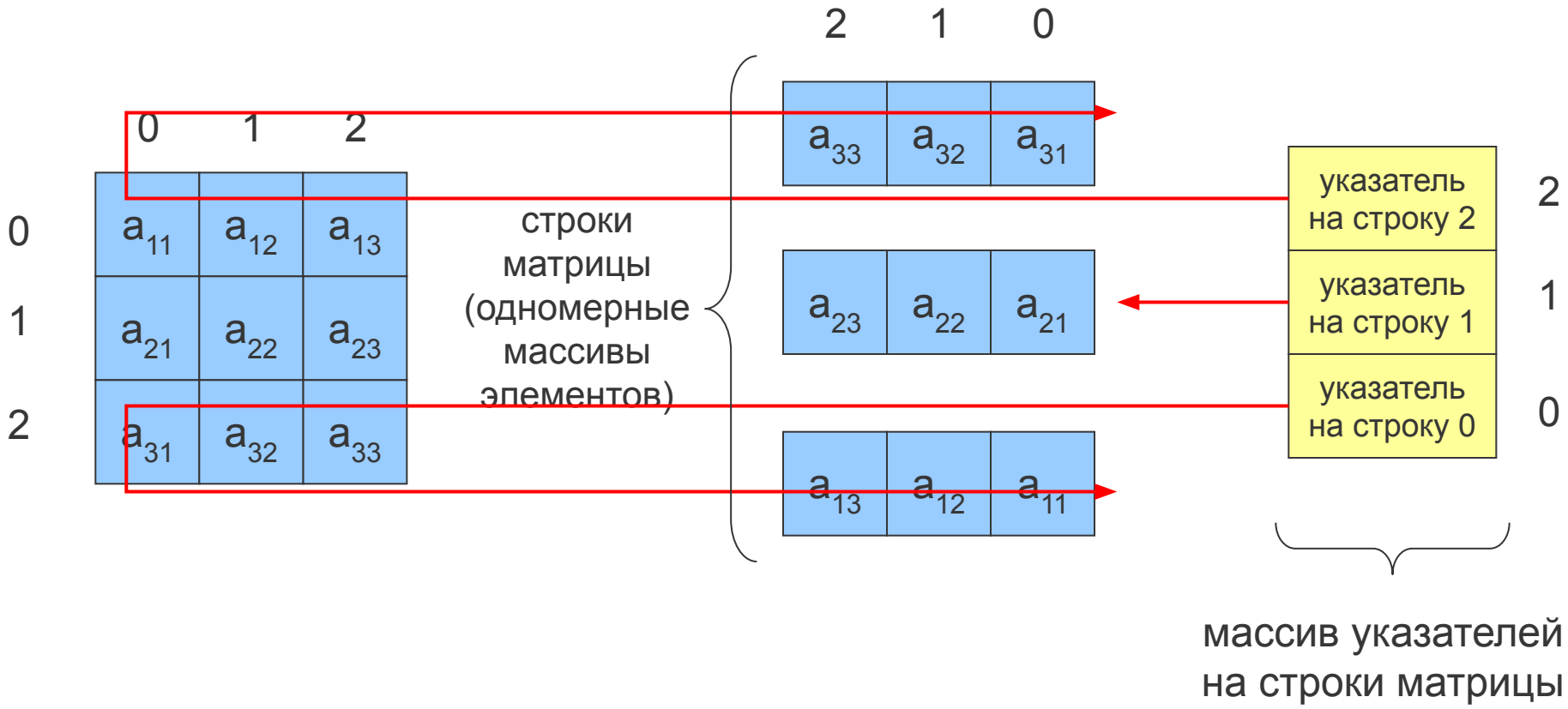
Представление двумерного массива одномерным

	0	1	2
0	a_{11}	a_{12}	a_{13}
1	a_{21}	a_{22}	a_{23}
2	a_{31}	a_{32}	a_{33}

a_{33}	a_{32}	a_{31}	a_{23}	a_{22}	a_{21}	a_{13}	a_{12}	a_{11}
8	7	6	5	4	3	2	1	0

```
int   *mtx = (int*)malloc(M * N * sizeof(int));
...
mtx[2][1] = 42;      // Не работает!

mtx[2*M + 1] = 42;
*(mtx + 2*M + 1) = 42;
...
free(mtx);
```

Представление двумерного массива массивом указателей



Представление двумерного массива массивом указателей

```
#define N 3
#define M 3

int* mtx[N] = {0};

for(int i = 0; i < N; ++i)
  mtx[i] = (int*)malloc(M * sizeof(int));
...
mtx[2][1] = 42;          // Работает!

*(*(mtx + 2) + 1) = 42;
...
for(int i = 0; i < N; ++i)
  free(mtx[i]);
```

Представление двумерного массива массивом указателей

```
int N, M;
scanf("%d %d", &N, &M);

int **mtx;

mtx = (int**)malloc(N * sizeof(int*));

for(int i = 0; i < N; ++i)
   mtx[i] = (int*)malloc(M * sizeof(int));
...
mtx[2][1] = 42;          // Работает!

*(*(mtx + 2) + 1) = 42;
...
for(int i = 0; i < N; ++i)
   free(mtx[i]);
free(mtx);
```

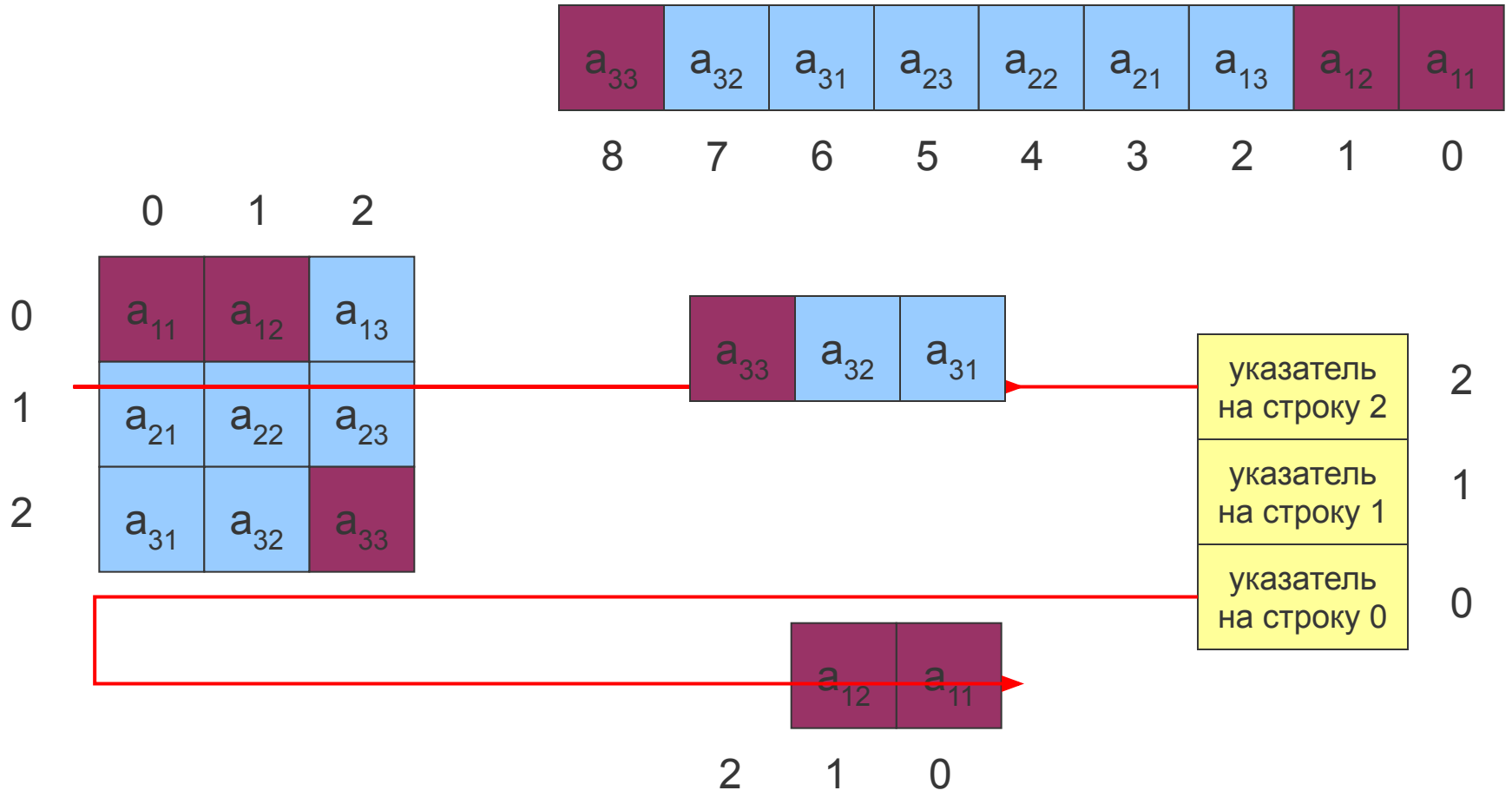
Разреженные массивы

- Массивы, в которых большинство элементов не инициализированы или имеют значение по умолчанию и не используются

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Примеры
  - листы в Excel
  - монохромные изображения
  - ...

Сравнение подходов представления двумерных массивов



Сравнение подходов представления двумерных массивов

- **Одномерным массивом**
  - более простые манипуляции с кучей
  - менее удобная нотация обращения к элементу `mtx[n*N+m]`
  - память выделяется для всего массива сразу; неудобно для представления разреженных массивов
- **Массивом указателей**
  - более сложное выделение памяти (сначала для массива указателей, потом для строк) и освобождение памяти
  - более удобная нотация обращения к элементу `mtx[n][m]`
  - есть возможность выделять память для строк по мере необходимости; более удобно для представления разреженных массивов

Размер блока

```
...
int size = 0;

scanf("%d", &size);

char *ptr = (char*) malloc(size);
...
```

- что будет, если пользователь введет
  - 0
  - -1

Размещение структур в куче

- динамическое создание структуры

```
Triangle *pt =
  (Triangle*) malloc(sizeof(Triangle));
if(pt != NULL) {
  ...

  // как обратиться к полю структуры?
  (*pt).isFilled = false;
}
```

Доступ к полям структур с помощью операции ->

`(*pt).isFilled` можно заменить на `pt->isFilled`

```
if(pt != NULL) {
   ...

  pt->isFilled = true;
}
```

Размещение структур в куче

- Динамическое создание массива структур

```
int  n; scanf("%d", &n);
Triangle *pt =
  (Triangle*) malloc(n*sizeof(Triangle));
...
(*pt).isFilled = true;
pt[0].isFilled = true;
pt[0]->isFilled = true;   //??
```

Размещение структур в куче

- Массив указателей на структуры

```
Triangle*   ptarr[10] = {0};

for(int j = 0; j < 10; j++)
  ptarr[j] =
   (Triangle*)malloc(sizeof(Triangle));
...
ptarr[0].isFilled = true;   //??
ptarr[0]->isFilled = true;
```

Размещение структур в куче

- Динамическое размещение массива указателей

```
int n; scanf("%d", &n);
Triangle **ptr =
  (Triangle**)malloc(n*sizeof(Triangle*));

for(int j = 0; j < n; j++)
  ptr[j] =
  (Triangle*)malloc(sizeof(Triangle));
...
ptr[1]->isFilled = true;
```

АТД

- **Абстрактный Тип Данных**
  - обобщенная форма представления данных
  - набор операций с данными

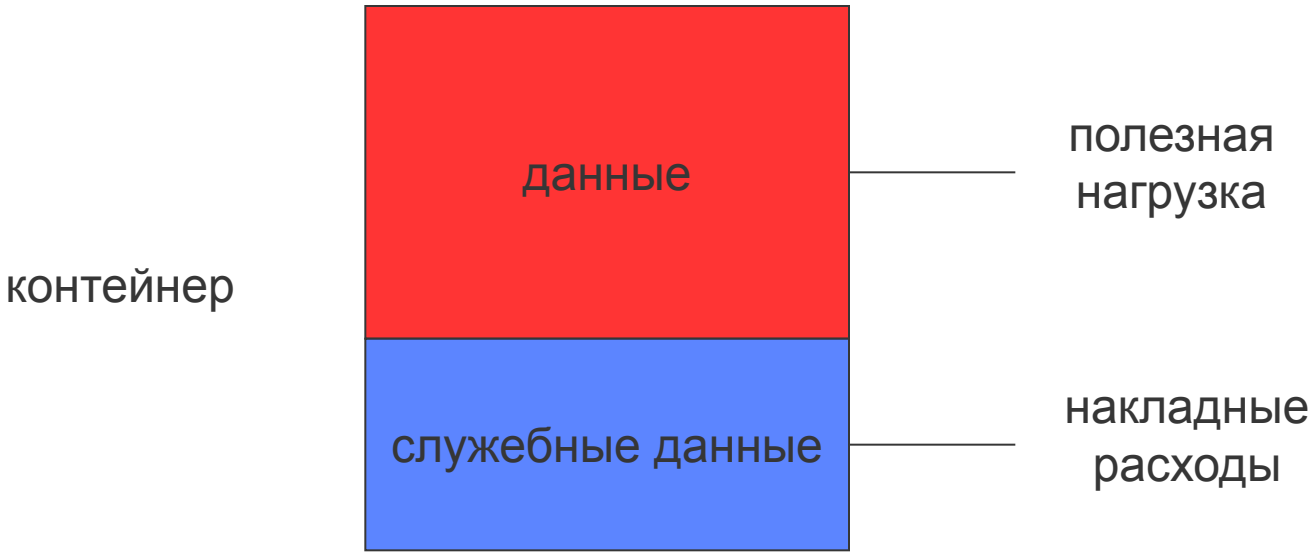
- Примеры АТД
  - списки
    - очередь
    - стек
    - ...
  - деревья
    - бинарное
    - B-дерево
    - красно-черное
    - ...
  - ассоциативные массивы
  - графы
  - ...

Списки

- Список – упорядоченное конечное множество элементов, каждый из которых имеет связь с другим элементом
  - размер списка
  - тип элементов
  - операции с элементами списка

Реализация АТД на С

- массивы
- структуры
- указатели

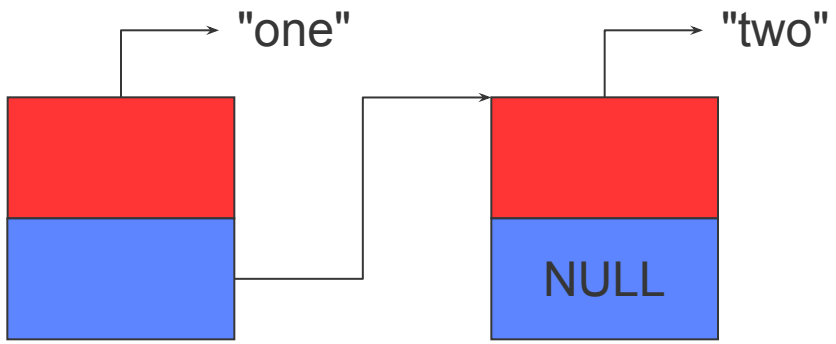


Реализация списка на основе указателей

```
struct list_item_t
{
  // полезная нагрузка

  list_item_t *next;
};
```

Создание списка

```
struct list_item_t
{
   void       *data;
   list_item_t  *next;
};

list_item_t *i1 = malloc(sizeof(list_item_t));
i1->data = "one";
i1->next = NULL;

list_item_t *i2 = malloc(sizeof(list_item_t));
i1->data = "two";
i1->next = NULL;

i1->next = i2;
```



Пример работы со списком

```
list_t l = new_list();

print(l);

append(l,"one");
append(l,"two");
int *array = (int*)malloc(sizeof(int)*100);
insert(l, array, 0);

print(l);

int idx = find(l, "two");

delete(l, 2);
delete(l, array);

delete_list(l);
```

Список на основе указателей

```
struct list_item
{
  void    *data;
  list_item   *next;
};

struct list
{
  list_item   *head, *tail;
};
```

Добавление элемента в конец

```
void append(list l, void *d) {
  list_item *li =
  (list_item*)malloc(sizeof(list_item));
  li->data = d;
  li->next = NULL;
  if(l->head == NULL)
   l->tail = l->head = li;
  else
   l->tail = l->tail->next = li;
}
```

Удаление элемента из списка

```
void delete(list *l, void *d) {
   for(list_item *li = l->head, *prev = NULL;
    li != NULL;
    prev = li, li = li->next)
    if(li->data == d) {
        if (li != l->tail && li != l->head) {
            prev->next = li->next;
        }
        else {
            if(li == l->head)
                l->head = li->next;
            if(li == l->tail)
                l->tail = prev;
        }
        free(li);
        break;
    }
}
```

Реализации АТД

- библиотеки исходных кодов
  - macros-based
- внешние библиотеки
  - Glib (from GNOME project)
    - https://developer.gnome.org/glib/

Резюме

- Указатели в C/C++ – это мощный инструмент, позволяющий осуществлять непосредственный доступ к памяти процесса (программы). Пользоваться ими нужно с осторожностью!
- если программе нужна память на этапе выполнения, то она может получить её из кучи, а затем должна вернуть её обратно