

# Указатели и одномерные динамические массивы

# Динамическое выделение памяти

Иногда в процессе выполнения программы удобно «создавать» переменные.

Для выделения памяти необходимо вызвать одну из трех функций (C99 7.20.3), объявленных в заголовочном файле `stdlib.h`:

- `malloc` (выделяет блок памяти и не инициализирует его);
- `calloc` (выделяет блок памяти и заполняет его нулями);
- `realloc` (перевыделяет предварительно выделенный блок памяти).

# Особенности malloc, calloc, realloc (1)

- Указанные функции не создают переменную, они лишь выделяют область памяти. В качестве результата функции возвращают адрес расположения этой области в памяти компьютера, т.е. указатель.
- Поскольку ни одна из этих функций не знает данные какого типа будут располагаться в выделенном блоке все они возвращают указатель на void.

# Особенности malloc, calloc, realloc (2)

- В случае если запрашиваемый блок памяти выделить не удалось, любая из этих функций вернет значение NULL.
- После использования блока памяти он должен быть освобожден. Сделать это можно с помощью функции free.

# malloc (1)

```
#include <stdlib.h>
```

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

- Функция *malloc* (C99 7.20.3.3) выделяет блок памяти указанного размера `size`. Величина `size` указывается в байтах.
- Выделенный блок памяти не инициализируется (т.е. содержит «мусор»).
- Для вычисления размера требуемой области памяти необходимо использовать операцию `sizeof`.

# malloc (2)

```
int *a = NULL;
size_t n = 5;

// Выделение памяти
a = malloc(n * sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

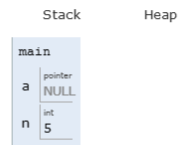
// Использование памяти
for (size_t i = 0; i < n; i++)
    a[i] = i;

// Освобождение памяти
free(a);
```

# malloc (3)

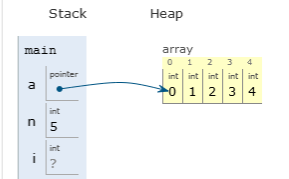
## 1. Перед выделением памяти

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = malloc(n * sizeof(int));
11
12    printf("a %p\n", a);
13
```



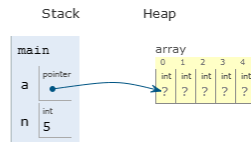
## 3. Использование выделенной памяти

```
14 // Проверка успешности выделения
15 if (a == NULL)
16 {
17     fprintf(stderr, "Memory allocation error\n");
18     return -1;
19 }
20
21 // Использование памяти
22
23 for (int i = 0; i < n; i++)
24     a[i] = i;
25
26 for (int i = 0; i < n; i++)
27     printf("%d ", a[i]);
```



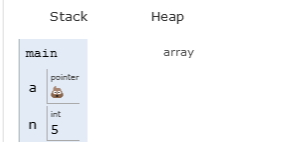
## 2. Сразу после выделения памяти

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = malloc(n * sizeof(int));
11
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
```



## 4. Сразу после освобождения

```
15 if (a == NULL)
16 {
17     fprintf(stderr, "Memory allocation error\n");
18     return -1;
19 }
20
21 // Использование памяти
22
23 for (int i = 0; i < n; i++)
24     a[i] = i;
25
26 for (int i = 0; i < n; i++)
27     printf("%d ", a[i]);
28
29 // Освобождение памяти
30 free(a);
31
32 return 0;
```



# malloc и явное приведение типа

```
a = (int*) malloc(n * sizeof(int));
```

## Преимущества явного приведения типа:

- компиляции с помощью c++ компилятора;
- у функции malloc до стандарта ANSI C был другой прототип (char\* malloc(size\_t size));
- дополнительная «проверка» аргументов разработчиком.

## Недостатки явного приведения типа:

- начиная с ANSI C приведение не нужно;
- может скрыть ошибку, если забыли подключить stdlib.h;
- в случае изменения типа указателя придется менять и тип в приведении.



# calloc (1)

```
#include <stdlib.h>
```

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

- Функция *calloc* (C99 7.20.3.1) выделяет блок памяти для массива из *nmemb* элементов, каждый из которых имеет размер *size* байт.
- Выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0.

# calloc (2)

```
int *a;
size_t n = 5;

// Выделение памяти
a = calloc(n, sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

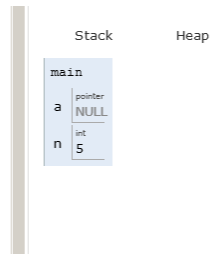
// Использование памяти
for (size_t i = 0; i < n; i++)
    printf("%d ", a[i]);

// Освобождение памяти
free(a);
```

# calloc (3)

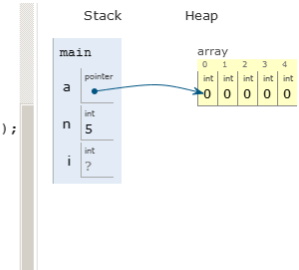
## 1. Перед выделением памяти

```
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = calloc(n, sizeof(int));
11
12    printf("a %p\n", a);
13
```



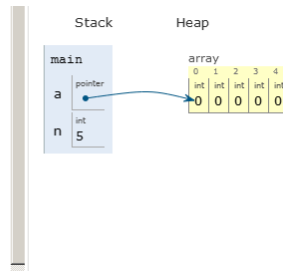
## 3. Использование выделенной памяти

```
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
15    if (a == NULL)
16    {
17        fprintf(stderr, "Memory allocation error\n");
18        return -1;
19    }
20
21    // Использование памяти
22    for (int i = 0; i < n; i++)
23        printf("%d ", a[i]);
```



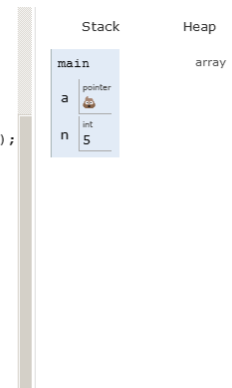
## 2. Сразу после выделения памяти

```
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = calloc(n, sizeof(int));
11
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
```



## 4. Сразу после освобождения

```
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
15    if (a == NULL)
16    {
17        fprintf(stderr, "Memory allocation error\n");
18        return -1;
19    }
20
21    // Использование памяти
22    for (int i = 0; i < n; i++)
23        printf("%d ", a[i]);
24
25    // Освобождение памяти
26    free(a);
27
28    return 0;
```



# free

```
#include <stdlib.h>
```

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

- Функция *free* (C99 7.20.3.2) освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает *ptr*.
- Если значением *ptr* является нулевой указатель, ничего не происходит.
- Если указатель *ptr* указывает на блок памяти, который не был получен с помощью одной из функций *malloc*, *calloc* или *realloc*, поведение функции *free* не определено.

# realloc

```
#include <stdlib.h>
```

```
void* realloc(void *ptr, size_t size); // C99 7.20.3.4
```

- `ptr == NULL && size != 0`

Выделение памяти (как `malloc`)

- `ptr != NULL && size == 0`

Освобождение памяти (как `free`).

- `ptr != NULL && size != 0`

Перевыделение памяти. В худшем случае:

- выделить новую область
- скопировать данные из старой области в новую
- освободить старую область

# Типичная ошибка вызова realloc

*Неправильно*

```
// pbuf и n имеют корректные значения  
pbuf = realloc(pbuf, 2 * n);
```

Что будет, если realloc вернет NULL?

*Правильно*

```
void *ptmp = realloc(pbuf, 2 * n);  
if (ptmp)  
    pbuf = ptmp;  
else  
    // обработка ошибочной ситуации
```

# Что будет, если запросить 0 байт?

Результат вызова функций `malloc`, `calloc` или `realloc`, когда запрашиваемый размер блока равен 0, зависит от реализации (implementation-defined C99 7.20.3):

- вернется нулевой указатель;
- вернется «нормальный» указатель, но его нельзя использовать для разыменования.

ПОЭТОМУ перед вызовом этих функций нужно убедиться, что запрашиваемый размер блока не равен нулю.

# Возвращение динамического массива из функции (прототип)

- Как возвращаемое значение

```
int* create_array(FILE *f, size_t *n);
```

- Как параметр функции

```
int create_array(FILE *f, int **arr, size_t *n);
```



# Возвращение динамического массива из функции (вызов)

- Как возвращаемое значение

```
int *arr;  
size_t n;  
arr = create_array(f, &n);
```

- Как параметр функции

```
int *arr, rc;  
size_t n;  
rc = create_array(f, &arr, &n);
```

# Типичные ошибки (1)

- Неверный расчет количества выделяемой памяти.
- Отсутствие проверки успешности выделения памяти
- Утечки памяти
- Логические ошибки
  - Wild (англ., дикий) pointer: использование непроинициализированного указателя.
  - Dangling (англ., висящий) pointer: использование указателя сразу после освобождения памяти.

# Типичные ошибки (2)

- Логические ошибки (продолжение)
  - Изменение указателя, который вернула функция выделения памяти.
  - Двойное освобождение памяти.
  - Освобождение невыделенной или нединамической памяти.
  - Выход за границы динамического массива.
  - И многое другое ☹

# Отладчик использования памяти (англ. memory debugger)

*Отладчик использования памяти* – специальное программное обеспечение для обнаружения ошибок программы при работе с памятью, например, таких как утечки памяти и переполнение буфера. [wiki]

- **valgrind**
- Dr. Memory

# Подходы к обработке ситуации отсутствия памяти (англ., OOM)

- Возвращение ошибки (англ., return failure)
  - Подход, который используем мы
- Ошибка сегментации (англ., segfault)
  - Обратная сторона - проблемы с безопасностью
- Аварийное завершение (англ., abort)
  - Идея принадлежит Кернигану и Ритчи (xmalloc)
- Восстановление (англ., recovery)
  - xmalloc из git