Лекция 15. Хранение, связывание и управление памятью. Часть 2.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Курс: Администрирование ОС «АЛЬТ»

https://eios.sibsutis.ru/course/view.php?id=4007

Выполнено уже 56% курса, всего 29 мест!

Кодовое слово: Basealt-2025

Содержание

- Ключевые слова: ***auto, extern, static, register, const, volatile, restricted, _Thread_local, _Atomic***
- Функции: ***rand(), srand( ), time( ), malloc( ), calloc(), free()***
- Определение в языке С области видимости переменной и времени жизни переменной
- Проектирование более сложных программ

Ключевые понятия (1)

Объект – …

Идентификатор – …

L-значение – …

Модифицируемое L-значение – …

Область видимости – …

Связывание – …

Продолжительность хранения – …

Ключевые понятия (2)

Объект – описание некоторого участка физической памяти, *определенного размера*, который может хранить одно или более значений. В зависимости от момента, объект может **содержать** или **не содержать** *сохраненного* значения.

```
int initVal = 0, val[5];
 //4byte          20byte
```

Ключевые понятия (3)

Идентификатор – отображает способ, которым программа написанная на Си, указывает на *объект*, хранящийся в аппаратной памяти. И регламентирует действия, которые могут быть выполнены с *объектом*.

```
int vals[10];  const int valConst[10];
int *pt=&vals; const *int ptConst=&vals;
```

Ключевые понятия (4)

L-значение – выражение которое обозначает *объект*. Чаще всего *L-значение* представлено идентификатором или выражением.

```
int val = 10;
int *pt=&val;
```

Ключевые понятия (5)

Модифицируемое L-значение – L-значение, которое может быть использовано для изменения *значения* внутри объекта.

```
const char * text = "Text";
```

Ключевые понятия (6)

Область видимости (ОВ) – описывает участок или участки программ, где можно обращаться к идентификатору. Например:

- в пределах блока
- в пределах функции
- в пределах прототипа функции
- в пределах файла

Ключевые понятия: связывание (7)

В Си переменная имеет одно из следующих связываний:

- внешнее связывание
- внутреннее связывание
- отсутствие связывания

Связывание – описывает видимость идентификаторов программы.

Ключевые понятия (8)

Продолжительность хранения (ПХ) – регламентирует постоянство объектов, доступных через эти идентификаторы.

Выделяют:

- статическое
- потоковое
- автоматическое
- выделенное

Выделенная память (1)

Память выделяется: статическая – во время загрузки программы, автоматическая – когда поток управления программы входит в блок.

Язык Си позволяет работать с выделенной памятью.

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

`size_t size` – значение в байтах, выделяемого блока памяти.

Возвращаемое значение, указатель на начало блока.

Выделенная память (2)

Функция `malloc()` – ищет подходящий по размеру блок свободной памяти. Память будет анонимной, т.к. `malloc()` не назначает выделенному блоку имя.

Следовательно, требуется результат (`void*`) функции `malloc()` присвоить переменной типа указателя для доступа к этой памяти.

`void*` – обобщенный указатель, тип которого должен быть изменен в соответствии с объектом данных размещаемом в выделенной памяти.

Выделенная память (3)

Благодаря этому свойству, функция `malloc` может применятся для выделение памяти под массивы и структуры и т.п. Ведь указатель можно привести к подходящему типу.

Пример: нужно создать массив в выделенной памяти из 10 элементов типа *double*.

```
double *dptr = NULL;
dptr = (double *) malloc(10 *
                         sizeof(double));
```

Выделенная память (4)

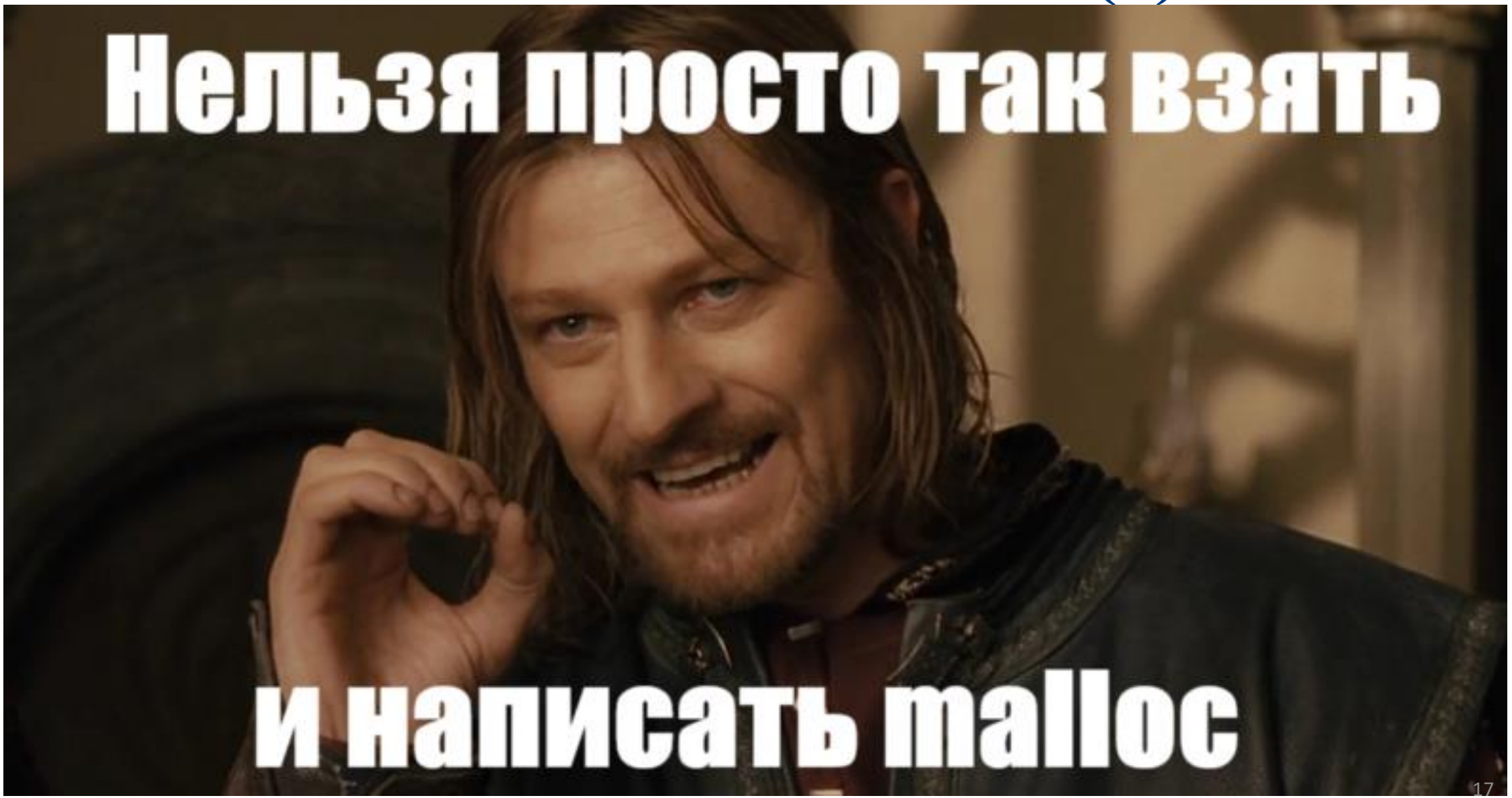
При использовании `malloc()` требуется явным образом выполнять приведение типов, но присваивание значения указателя на `void*` указателю другого типа не является конфликтом типов.

Также если не удается найти запрошенный блок памяти, `malloc()` возвращает нулевой указатель.

Выделенная память (5)

```
double *dptr = NULL;
dptr = (double*)malloc(10 * sizeof(double));
if (dptr == NULL)
{
    puts("Не удалось выделить память.
                    Программа завершена.");
    exit(EXIT_FAILURE);
}
```

Выделенная память (6)



Выделенная память (7)

Каждый вызов функции `malloc()` должен сопровождаться вызовом процедуры `free()`.

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

В качестве единственного аргумента принимает адрес, возвращенный ранее функцией `malloc()`.

Вызов процедуры `free()` освобождает память, которая была выделена. Таким образом продолжительность хранения выделяемой памяти, зависит от комбинации `malloc()…free()`.

Выделенная память (8)

Прототипы `malloc()` и `free()`, находятся в заголовочном файле stdlib.h.

Однако комбинация функций `malloc()` и `free()`требуют особого внимания от программиста, ведь слишком частый цикл их использования и/или не корректность логики использования полностью попадает под его ответственность.

Кейс 1: Утечка памяти. (1)

```
int main(void){
double glad[2000];
for (int i = 0; i < 1000; i++)
            gobble(glad, 2000);}
void gobble(double ar[], int n){
double * temp = (double *) malloc(n *
                              sizeof(double));
    ... // free(temp);
}
```

Кейс 1: Утечка памяти (решение). (2)

```
int main(void){
double glad[2000];
for (int i = 0; i < 1000; i++)
            gobble(glad, 2000);}
void gobble(double ar[], int n){
double * temp = (double *) malloc(n *
                              sizeof(double));
    ... free(temp);
}
```

Кейс 2: Memory Allocation. (1)

```
int main(void){
double glad[2000];
for (int i = 0; i < 1000; i++)
            gobble(glad, 2000);}
void gobble(double ar[], int n){
double * temp = (double *) malloc(n *
                              sizeof(double));
    ... free(temp);
}
```

Кейс 2: Memory Allocation. (2)

```
int main(void){
double glad[2000];
double * temp = (double *) malloc(n *
                                  sizeof(double));
for (int i = 0; i < 1000; i++)
            gobble(glad, 2000);
free(temp);
}
void gobble(double ar[], int n){...}
```

Способы создания массива (1)

Объявить массив, используя *константные выражения* для размерностей, и применять для доступа к элементам имя массива. Такой массив может быть создан с использованием либо *статической*, либо *автоматической* памяти.

```
double glad[2000]; //вне функции main
…
double glad[2000]; //в любой функции
```

Способы создания массива (2)

Объявить *массив переменной длины*, применяя *переменные выражения* для размерностей, и использовать для доступа к элементам имя массива. Такой вариант доступен только для *автоматической памяти*.

```
void foo(int); //прототип
void foo(int n) //реализация
{    int arr[n];
…}
```

Способы создания массива (3)

Объявить указатель, вызвать `malloc()`, присвоить возвращаемое значение *указателю* и применять для доступа к элементам указатель. Этот *указатель* может быть либо *статическим*, либо *автоматическим*.

```
int pint = NULL;
pint = (int*)malloc(2000 * sizeof(int));
free(pint);
```

Функция calloc() (1)

Функция `calloc()`, предлагает ещё один способ выделения памяти.

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

где *num* – кол-во элементов, а *size* размер элемента в байтах. Особенность `calloc()` заключается в том, что блок инициализирует все биты нулями.

```
int * pData, i;  scanf ("%d",&i);
pData = (int*) calloc (i, sizeof(int));
if (pData==NULL) exit (EXIT_FAILURE);
```

Функция calloc() (2)

Создание двумерного массива переменной длины (**VLA - Variable Length Arrays**)

```
void vla2d (int rows, int cols) {
    int array2D[rows][cols];
...
}
```

Функция calloc() (3)

Создание двумерного массива на динамической памяти с использованием calloc()

```
void dyn2d(int rows, int cols) {
 int** array2D = (int**)calloc(
                  rows, sizeof(int*));
 if(array2D==NULL) exit(EXIT_FAILURE);
…
```

Выделяем память под массив указателей типа int

Функция calloc() (4)

```
…
 for (int i = 0; i < rows; i++) {
  array2D[i] = (int*)calloc(cols,
sizeof(int));

…
```

Для каждого i-того указателя в array2D выделяем блок памяти `cols * sizeof(int)`

Функция calloc() (5)

```
…
if (array2D[i] == NULL) {
   for (int j = 0; j < i; j++)
     free(array2D[j]);
   free(array2D);exit (EXIT_FAILURE);
}}…
```

Если не получилось выделить память, освобождаем каждый i-тый указатель и двумерный указатель.

Функция calloc() (6)

```
//Обработка array2D
for (int i = 0; i < rows; i++)
                        free(array2D[i]);
free(array2D);}
}
```

Обрабатываем последовательность array2D

Освобождаем память выделенную каждому i-тому указателю и двумерный указатель array2D.

Функция calloc() (6)

Создание трехмерного массива переменной длины

```
void vla2d (int depth, int rows,
                       int cols){
int array3D[depth][rows][cols];
//Обработка массива array3D
...
}
```

Сравнение VLA и выделяемой памяти (1)

Преимущества VLA:

- Автоматическое управление памятью.
- Более производительно, т.к. выделяется на стеке.
- Читаемость и простота кода.

Недостатки VLA:

- Размеры массивов ограничены размером стека (default 8Mb), есть риск **`stack overflow`**
- Нет совместимости с C++, при компиляции

Сравнение VLA и выделяемой памяти (2)

Преимущества выделенной памяти:

- Размеры массивов ограничены размером доступной памяти.
- Ручное управление памятью выделение/освобождение.
- Совместимость с C++ компиляторами.

Недостатки выделенной памяти:

- Читаемость кода при работе с вложенными массивами.
- Необходимо явно освобождать память избегая “утечки”.
- Частое выделение памяти может снижать производительность

Понятие: конструктор (1)

Сам по себе термин берёт своё начало в объектно-ориентированных языках программирования (C++, Java, etc.)

Суть *конструкции* – в подготовке объекта данных к работе с ним: выделение памяти, инициализация его значениями, проверка полученного объекта данных.

Язык Си процедурный и не содержит конструкторов, однако создание методов типа “конструктор” позволяет лучше разобраться в использовании выделенной памяти.

Понятие: конструктор (2)

```
typedef struct stud {
    char * fname;
    char * lname;
    unsigned char age;
} STUD;
```

Понятие: конструктор (3)

```
void stud_construct(STUD * obj, const char *
fname, const char * lname, const unsigned char
age){
obj->age = age;
obj->lname = (char*)malloc((strlen(lname) + 1) *
                                    sizeof(char));
strcpy(obj->lname, lname);
obj->fname = (char*)malloc((strlen(fname) + 1) *
                                    sizeof(char));
strcpy(obj->fname, fname);}
```

Понятие: конструктор (4)

```
typedef struct stud {
    char * fname;
    char * lname;
    unsigned char age;
} STUD;
STUD * std1 = NULL;
stud_construct(std1,"Иван", "Иванов",
18);
```

Понятие: конструктор (5)

Если возникает потребность создавать объект, но при этом на этапе компиляции мы не знаем его значений или в какой-то из задач не стоит цели задавать значения объекту, возникает понятие конструктор по умолчанию.

Смысл такого конструктора в заполнении объекта пустыми/нулевыми значениями.

Понятие: конструктор (5)

```
STUD * stud_default_construct(){
    STUD * obj = (STUD *
)malloc(sizeof(STUD));
    obj->fname = NULL;
    obj->lname = NULL;
    obj->age = 0;
    return obj;
}
```

Понятие: конструктор (6)

```
typedef struct stud {
    char * fname;
    char * lname;
    unsigned char age;
} STUD;
STUD * std1 = stud_default_construct();
stud_construct(std1,"Иван", "Иванов",
18);
```

Понятие: конструктор (7)

При работе с указателями следует быть очень внимательным не забывая при этом, что часть операций между переменными и переменными типа указатель различны. Например операция копирования.

Понятие: конструктор (8)

```
STUD * std1 = NULL;
stud_construct(std1,"Иван", "Иванов",
18);
STUD * std2 = std1;
std2->age = 20;
stud_print(std1);
stud_print(std2);
```

```
Адресс стр-ры: 0x5619cd4432a0
Фамилия: Иванов
Имя: Иван
Возраст: 20
Адресс стр-ры: 0x5619cd4432a0
Фамилия: Иванов
Имя: Иван
Возраст: 20
```

Понятие: конструктор (9)

Отсюда становится понятно, что даже при извлечении указателя или при взаимодействии с ним, скопировать данные не получается. На этот случай выручает разработанный конструктор копирования. Задача которого корректно скопировать данные из одного блока данных в другой.

Понятие: конструктор (10)

```
void stud_copy_construct(STUD * left, const STUD *
right){
if(right->fname == NULL ||  right->lname == NULL) {
left->fname = NULL; left->lname = NULL;}
else {
left->fname = (char*)malloc((strlen(right->fname) + 1) *
sizeof(char));
strcpy(left->fname, right->fname);
left->lname = (char*)malloc((strlen(right->lname) + 1) *
sizeof(char));
strcpy(left->lname, right->lname);
    left->age = right->age;}
```

Понятие: деструктор (1)

Отвечает за завершение работы с объектом: очищает/освобождает память и указатели.

```
void stud_destruct(STUD *obj){
  free(obj->fname); obj->fname = NULL;
  free(obj->lname); obj->lname = NULL;
  obj->age = 0; free(obj);
}
```

Самостоятельно изучить

Раздел: Функция генерации случайных чисел и статическая переменная 502 страница, глава 12

Продолжение следует…
Спасибо за внимание

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)