Основы программирования

Логинов Иван Павлович, ассистент кафедры ИПМ.

Контакты

Email: *ivan.p.loginov@gmail.com*

Аудитория: *378 (Кронверкский пр.)*

Расписание консультаций

Понедельник – с 15:50 до 17:10.

Среда – с 17:20 до 18:40.

О курсе

Задача курса - обучение основам создания коммерческих программных продуктов:

- разработка эффективных алгоритмов;
- выбор наиболее подходящих языковых средств;
- выбор оптимальных или близких к ним структур данных;
- профессиональное кодирование.

Содержание курса

- Алгоритмизация
- Структуры данных

- Процедурное программирование
- Объектно-ориентированное программирование

- Разработка GUI

- Общие рекомендации по написанию кода

О БАРС

	Min	Max
Лабораторные работы (5 шт.)	30	50
Рубежный контроль (2 шт.)	12	20
Личностные качества (2 шт.)	6	10
Экзамен	12	20
Total:	**60**	**100**

Парадигмы программирования

Парадигма — модель вычислений, способ организации программы, то есть принцип ее построения.

Императивные языки	Аппликативные языки
Языки, основанные на системе правил	Объектно-ориентированные языки

Процедурное программирование

Парадигма, предполагающая использование императивного языка программирования, при котором выполняемые операторы сгруппированы в подпрограммы.

Процедурная декомпозиция состоит в том, что задача, реализуемая программой, делится на подзадачи, а они, в свою очередь — на более мелкие этапы, то есть выполняется пошаговая детализация алгоритма решения задачи.

Язык С

Особенности:

- Очень близок к машинному представлению программы («высокоуровневый ассемблер»);
- Программы на С переносимы (если написаны правильно);
- В С существует невероятное множество способов «выстрелить себе в ногу».

Структура программы на языке С

- Описание типов данных (структуры, новые типы и т.п.)
- Описание глобальных переменных;
- Описание различных функций;
- Комментарии (многострочные).

Пример простой программы на С

```
#include <stdio.h>
void main() {
   printf("Hello World!");
}
```

Структура языка С

Допустимые символы –

- Все символы латинского алфавита;
- Все цифры;
- Специальные символы
  - Точка (.), запятая (,), точка с запятой (;), знак вопроса (?), двоеточие (:);
  - +, -, *, /, %, &, |, ~, ^, !, >, <
  - Скобки (,), {, }, [,]
  - Знак подчёркивания (_)
  - Кавычки: ‘, “

Вместе эти символы образуют лексемы –

- Идентификаторы
  - Некоторые идентификаторы зарезервированы – это ключевые слова;
- Знаки операций;
- Предопределённые константы
  - Символьные (выделяются одинарными кавычками);
  - Строковые (выделяются двойными кавычками);
  - Числовые.

Операции в С

Операция – действие, выполняемое над операндами, возвращающее значение – результат операции.

Операнд – выражение (константа, переменная).

Операции бывают трёх видов:

- Унарные – с одним операндом;
- Бинарные – с двумя операндами;
- Тернарные.

Выражения

Выражение – набор операций над переменными.

Выражение = операнд + знаки операций.

Порядок выполнения определяется в соответствии с приоритетами.

Операторы в С

Оператор – минимально возможное действие (с точки зрения записи на языке программирования), команда, инструкция.

- Простейший оператор - ;
- Выражения – (expression);
- Условные операторы (if, switch);
- Операторы выполнения циклов (for, while)
- Оператор перехода (goto);
- Оператор возврата из подпрограммы(return).
- Блоки вычислений.

Блок вычислений

Называется любая последовательность инструкций С, заключённая в фигурные скобки.

Например, тело функции – блок, также:

- блок может быть телом цикла while или for
- одной из веток конструкции if-then-else
- может вовсе быть просто частью функции.

```
#include <stdio.h>
void main()
{
   //это - блок
}
```

Типы данных

Для компьютера всё – последовательности 0 и 1 (битовые строки).

Можно считать, что данные в компьютере не типизированы.

Тип данных – множество значений и операций на этих значениях (IEEE Std 1320.2-1998).

Типизация защищает нас от некорректных операций, в частности, от

- присваивания с разными типами данных;
- применения функций к аргументам несовместимых типов.

Виды типизации

Статическая типизация – проверка корректности выражений происходит на этапе компиляции.

Динамическая типизация – работает во время выполнения.

Строгая типизация подразумевает, что все операции должны быть произведены в точности с теми данными, которые им приходят.

Нестрогая типизация допускает возможность неявно преобразовывать типы.

Явная типизация – мы явно указываем типы данных.

Неявная типизация – компилятор может сам определить тип данных (если это возможно).

Примеры видов типизации

- Язык Ocaml обладает строгой статической типизацией
  - <float> +. <float>
  - <int> + <int>
  - 40 + 2.0 – ошибка.
- Язык C – с нестрогой статической типизацией
  - double x = 4 + 3.0;
- Пример строгой динамической типизации – Python
  - double x = 1.0f + 2;
- ‘1’ + 1 = ‘11’ (JavaScript).

Типы данных в C (1/2)

Встроенные типы данных:

- char
  - Бывает signed и unsigned (по-умолчанию *обычно* signed);
  - Занимает 1 байт;
  - Тип хранит ***число,*** которое ***может*** интерпретироваться как код таблицы ASCII;
- int
  - Целое число;
  - Бывает signed и unsigned. По-умолчанию signed;
  - Есть синонимы – signed, signed int;
  - Может быть short (2 байта), long (4 байта на 32-разрядных архитектурах, 8 на 64-разрядных) и long long (8 байт);
  - Также можно писать short, short int, signed short, signed short int;
  - Просто int – размер машинного слова (в 64-разрядных архитектурах занимает 4 байта, в то время как в 16-разрядных – 2 байта, short)

Типы данных в C (2/2)

- float
  - Число с плавающей запятой;
  - Занимает 4 байта;
  - Диапазон представления:
    - $1{,}17549435 \times 10^{-38}$ … $3{,}4028235 \times 10^{38}$
- double
  - Число с плавающей запятой;
  - Занимает 8 байт;
- Помимо вышеперечисленных типов, в C существуют также структуры, перечисления, объединения;
- В C нет типов данных bool, string.
  - Ноль считается ложью, ненулевое значение – истиной;
- Написав символ в одинарных кавычках, подразумевает ASCII-код.

Пример объявления переменных (1/2)

int variable_name;

Сначала – имя типа, потом – имя переменной.

Объявление возможно в

- Вне функции (глобальные переменные);
- В начале блока инструкций ({ … });

```
#include <stdio.h>
void main() {
    int a = 2;
    int b = 5;
    {
        int x = 4;
        int b = 0;
    }
}
```

Пример объявления переменных (2/2)

Возможно объявление констант – для этого достаточно использовать ключевое слово const:

Значение констант может быть указано только в момент объявления, и не может быть изменено в дальнейшем.

```
#include <stdio.h>
void main() {
    const int a = 2;
    int b = 5;
    {
        int x = 4;
        int b = 0;
        a = 5; //ошибка времени компиляции!
    }
}
```

Определение типов

В С можно определять новые типы данных на основе существующих.

Пример: typedef unsigned long mass_t;

Это позволяет:

- Семантически упростить восприятие кода (например, умножить ускорение на массу, а не float на unsigned long);
- Упростить процесс поддержки кода – не нужно повсюду, при изменении типа, например, с **unsigned long** на **unsigned long long**, искать использование сложного имени.

Определение типов (пример)

```
typedef int speed_t;
typedef int distance_t;
typedef int time_t;

void main()
{
	int speed = 2;
	int time = 3;
	int distance = speed * time;

	speed_t s1 = 10;
	time_t t1 = 50;

	distance_t dist1 = t1 * s1;
}
```

Преобразование типов

- Явное преобразование

Логически числовые типы организованы в иерархию, в которой данные каждого типа могут «всплывать» вверх, если в ходе вычислений требуется бОльшая точность.

Выполняется при помощи оператора (<имя_типа>)

```
int b = 127;
char c = (char)b;
```

- Неявное преобразование – выполняется автоматически.

```
char c = 2;
int a = c + 42;
long b = a;
```

Условный оператор

if (<condition>) //if true

if (<condition>)
else //if else

if (<condition-1>)
else if (<condition-2>)

Также возможно описание тела каждого оператора в блоке.

- Истина – не ноль;
- Ложь – ноль.

Никогда не проверяйте условие так: *if (a == 2).*
Вы можете опечататься, *if (a = 2)* – условие верно всегда.
Пишите так: *if (2 == a).*

Условный оператор (пример)

```
#include <stdio.h>

void main() {
        int value = 0, isNumberValid = 0;

        isNumberValid = scanf("%i", &value);
        if (0 == isNumberValid)    {
                printf("Not a number.\n");
        }
        else  {
                if (value >= 0) {
                        printf("Value is positive.\n");
                }
                else {
                        printf("Value is negative.\n");
                }
        }
}
```

Договоримся пока что просто считать это как «необходимое для работы с пользовательским вводом-выводом»

scanf – функция для чтения данных из потока ввода (stdin)

printf – функция для форматированного вывода данных в консоль (в поток стандартного вывода, stdout)

Условный оператор (пример 2)

```
#include <stdio.h>

void main()
{
	int value = 0, isNumberValid = 0;

	isNumberValid = scanf("%i", &value);

	if (!isNumberValid)
	{
		printf("Not a number.\n");
		printf("Foo!");
	}
	else if (value >= 0)
		printf("Value is positive.\n");
	else
		printf("Value is negative.\n");
}
```

Оператор множественного выбора

Позволяет выбирать вариант из нескольких:

```
switch (<variable_name>)
{
        case <constant-1>: //do something
        case <constant-2>: break;
        default: break;
}
```

default – выбор по-умолчанию.

break – прерывает «проваливание» вниз по меткам.

Оператор множественного выбора

```
int dayNumber = 0;

scanf("%i", dayNumber);

switch (dayNumber)
{
        case 1:
                printf("Monday");
                break;
        case 2:
                printf("Tuesday");
                break;
        default:
                printf("Unknown.");
                break;
}
```

Оператор goto

```
void main()
{
        int i = 0;
incl:
        i = i++;

        if (i <= 10) goto showSquare;
        if (i > 10) goto completeProgram;
completeProgram:
        printf("Completed.");
        return;
showSquare:
        printf("square if %i = %i\n", i, i * i);
        goto incl;
}
```

Перед использованием оператора goto стоит задуматься: он может привести программу очень сложной для восприятия, что, в свою очередь, приведёт к порождению ошибок при её изменении.

Операторы выполнения циклов (1/2)

- Цикл с предусловием:

```
do {
        //loop body
} while (<condition>);
```

- Цикл с постусловием:

```
while (<condition>)
        //loop body
while (<condition>)
{
//body
}
```

Операторы выполнения циклов – while

```
int working = 1;
while (working) {
        int menuItemNumber = 0;
        printf("Choose the action:\n");
        printf("1 - Do something.\n");
        printf("2 - Exit.\n");
        scanf("%i", &menuItemNumber);
        switch (menuItemNumber)
        {
        case 1:
          printf("Something is done.");
          break;
        case 2:
           working = 0;
           continue;
        default:
           printf("Wrong menu item.");
           break;
        }
}
```

Операторы выполнения циклов (2/2)

Цикл for:

for (init-block; condition; increment/decrement)

    body

init-block – блок инициализации переменных цикла;

condition – условие, в пределах истинности которого выполняется цикл;

increment-decrement – выражение, которое выполняется, как правило, для изменения переменной цикла.

Цикл for – вычисление числа Фибоначчи

```
int n = 5, i = 0;

int prevPrev = 0;
int prev = 1;
int result = 0;

for (i = 2; i <= n; i++)
{
        result = prev + prevPrev;
        prevPrev = prev;
        prev = result;
}
```

Операторы break, continue

- break используется для прерывания выполнения цикла
- Также возможно его использование в операторе switch (…), для окончания «проваливания» по меткам (case-ам) далее, вниз.

- Оператор continue применяется для перехода на следующую итерацию цикла, операторы, которые записаны после continue, не выполняются.

Тернарный оператор

```
int a = 5, b = 4;
int max = a > b ? a : b;
```

Аналогичная по смыслу форма записи:

```
if (a > b)
        max = a;
else
        max = b;
```

Структуры, перечисления, объединения

Один из основных принципов осмысления для человеческого мозга — абстракция.

Абстракция означает замену низкоуровневых понятий более удобными, высокоуровневыми.

Пример –

Человек размышляет так: «Пойти в университет на пару», «перейти через дорогу», а не «Переставить левую ногу на 44 см».

Структуры

Некий «сборный» тип данных.

```
struct {
        int x;
        int y;
} vec;
vec.x = 5;
vec.y = -4;
```

Такой способ объявления структуры «одноразовый» - каждый раз, при объявлении новой такой структуры, её заново нужно определить

Обращение к ***полям*** структуры происходит при помощи точки (.).

Пример объявления структуры

```
typedef struct { int x, int y } vector_t;


void main()
{
        vector_t a, b, c;
        a.x = 2;
        a.y = 6;
        b.x = -5;
        b.y = 9;
        c.x = a.x – b.x;
        c.y = a.y – b.y;
}
```

Объединения

Объединения очень похожи на структуры, только их поля не «укладываются» в памяти друг за другом, а занимают одно и то же место, «накладываются» друг на друга.

```
typedef union {
        long int integer;
        short shorts[2];
} dword_t;
dword_t test;
test.integer = 0xAABBCCDD;
```

Вывод значения поля integer:
AABBCCDD

Вывод значения поля shorts:
CCDD AABB

Перечисления

Простой тип данных на основе целого числа.

Например, состояния светофора, пункты консольного меню, и т.д.

Пример состояний светофора – красный, жёлтый, зелёный, красно-жёлтый, выключен.

```
typedef enum {
        red = 0,
        red_and_yellow,
        yellow,
        green,
        off
} light_t;
light_t a = off;
```

Подпрограммы

Подпрограмма – независимая совокупность объявлений и операторов, предназначенная для решения определенной задачи.

Однажды определённая подпрограмма может быть многократно ***вызвана.***

Подпрограммы могут обрабатывать различные данные, переданные в качестве аргументов.

Зачем нужны подпрограммы?

Разбиение множества действий, выполняемых программой (в частности, единообразных), на небольшие подзадачи, что позволяет:

- Избежать дублирования кода – это упрощает сопровождение;
  - Достигается некоторая экономия памяти*.
- Упростить восприятие программы;
- Выделять подпрограммы в отдельные модули;

Процедуры и функции

Условно можно разделять функции и процедуры.

- Функция – подпрограмма, возвращающая значение, её можно использовать в сложных выражениях.
- Процедура – подпрограмма, не возвращающая значения.

```
//это – процедура
void myProc(int a, int b)
{
    printf("%i", a + b);
}

//где-то в блоке
myProc(1, 5 + 1);
//вывод: 7
```

```
//это – функция
int myFunction(int a, int b)
{
    return a + b;
}
//где-то в блоке
int a = myFunction(6, -1);
printf("%i", a);
//вывод: 5
```

Объявление и определение функций

- Объявление функции – её идентификатор, а также список параметров, тип возвращаемого значения.
  - Объявление функции просто даёт понять компилятору, что она (функция) существует.
  - Объявление функции не содержит её тела (семантики функции – описания того, что функция делает).
  - Называют также прототипом функции.
  - **Пример:** `int abs(int number);`
- Определение функции – описание того, что функция делает, т.е. – исполняемый код функции.

Пример:

```
int abs(int number)
{
    return number < 0 ? -number : number;
}
```

Внимание – в примере есть недочёт: отсутствует проверка диапазона значений параметра

Сигнатура функции

Сигнатура функции – информация о функции, которая используется при её вызове (по которой её идентифицирует компилятор).

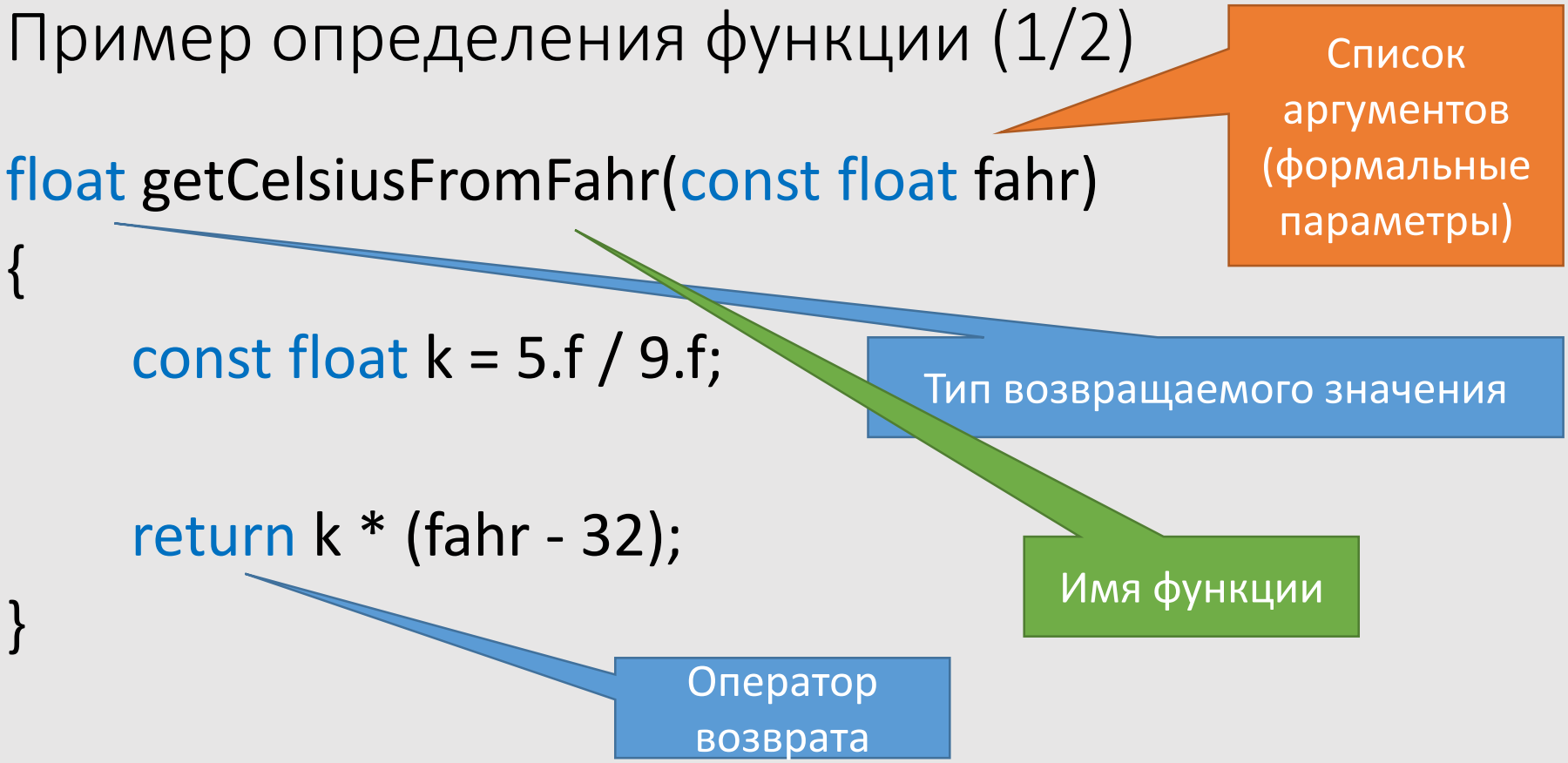
Сигнатура функции вовсе не обязательно совпадает с её объявлением *(например, в C, C++, C# тип возвращаемого значения не является частью сигнатуры).*

Схематическое определение функции

<тип> <имя_функции> (список_параметров)
{
    объявления и операторы
    (тело функции)
}

- Тип – тип возвращаемого значения:
  - любой объявленный тип данных;
  - void – специальное имя типа, означающее, что функция не возвращает значения.
- Имя функции – обычный идентификатор С.
- Список параметров может быть пустым.

Пример определения функции (1/2)



Объявленные внутри функции переменные называют **локальными**, т.е. доступны только внутри блока функции.

Пример определения функции (2/2)

```
#include <stdio.h>

double getCelsiusFromFahr(const double);

void main()
{
        double t = getCelsiusFromFahr(-40);

        printf("%f", t);
}

double getCelsiusFromFahr(const double fahr)
{
        const double k = 5.f / 9.f;

        return k * (fahr - 32);
}
```

Обратите внимание – сначала указано объявление функции

Выражение вызова функции. Оно может входить в состав более сложных выражений:
y = sin(x) + cos(x);
a = sqr(sqr(2));

Модификатор const перед параметром гарантирует, что параметр в функции не будет изменён

Выражение вызова функции

выражение(список_выражений)

Выражение, задающие адрес функции, например ее имя

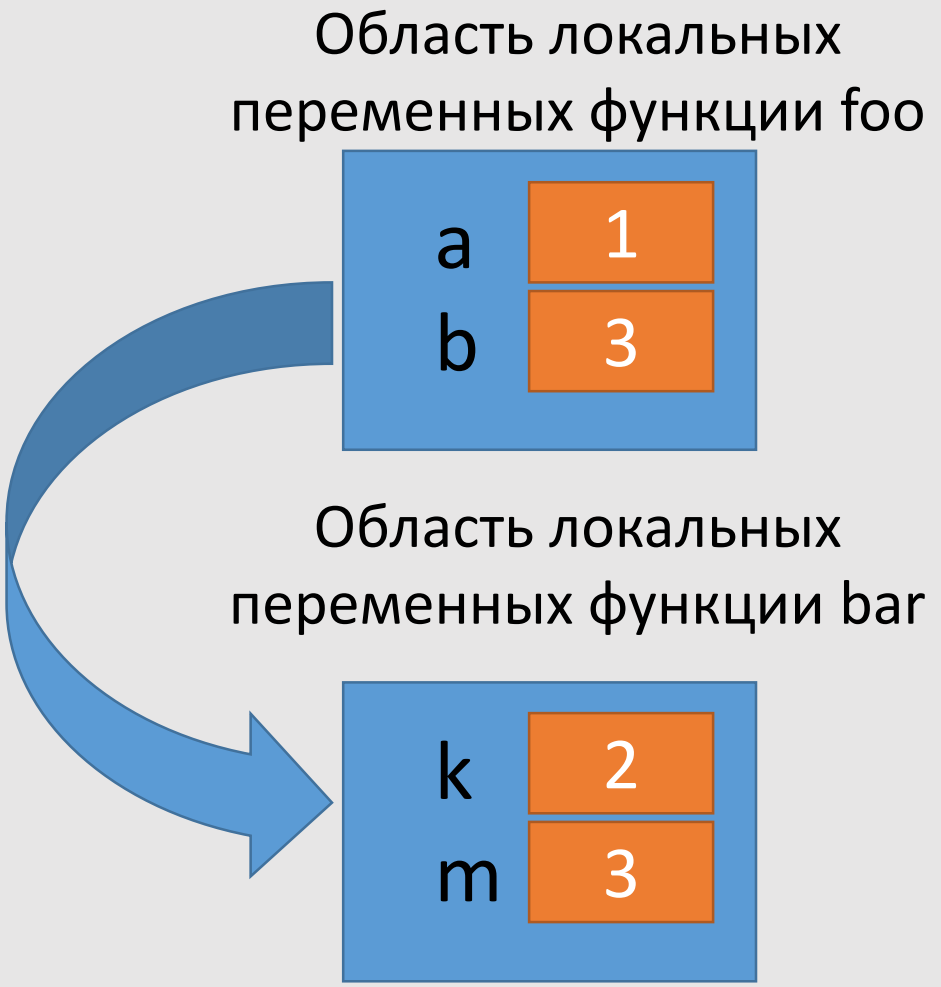
Выражения, являющиеся фактическими параметрами. Имена переменных, константы, сложные выражения

foo(a + b, 5, f(a / b + 10), f(b) + g(a))

В случае необходимости осуществляется стандартное арифметическое приведение типов

Передача параметров

```
int sum(int, int);
void foo()
{
   int a = 1, b = 3;
   int c = sum(a, b);
}
int sum(int k, int m)
{
        m++;
        return k + m;
}
```



Список параметров

- Простое перечисление параметров (тип + имя)
- Возможно объявлять функции с пустым списком параметров.
- Список параметров может быть переменной длины.
- В качестве модификатора типа параметра можно указать **const** (это важно для указателей).

Формальные и фактические параметры

- Формальный параметр – аргумент, указанный при объявлении функции.
- Фактический параметр – аргумент, переданный в функцию при её вызове.

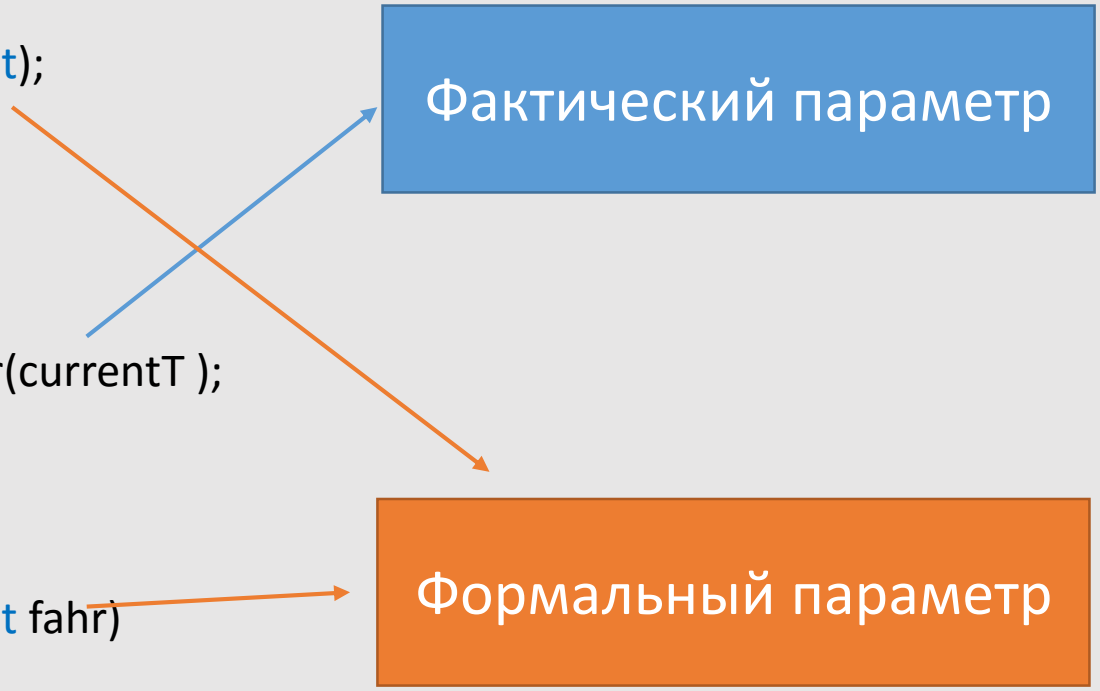
```
#include <stdio.h>

float getCelsiusFromFahr(const float);

void main()
{
        int currentT = -40;
        float t = getCelsiusFromFahr(currentT );
        printf("%f", t);
}

float getCelsiusFromFahr(const float fahr)
{
        const float k = 5.f / 9.f;

        return k * (fahr - 32);
}
```



Перегрузка функций в C

- Невозможна в «чистом» виде (по типу и/или числу параметров, по типу возвращаемого значения).
- Возможно несколько маленьких хитростей:
  - Поступить, как с функцией printf, передавая форматную строку:
    - printf(“%i”, 1); printf(“%i\n%f”, 1, 2.0);
    - Это сопряжено с риском: форматная строка и список параметров могут не соответствовать друг другу.
  - Просто давать функциям имена с постфиксом:

    abs, absf, и т.д., где:

    abs(int),

    absf(float).

Способы передачи параметров

- По значению – в функции создаётся локальная копия переданного значения (фактического параметра)
  - Изменение формального параметра (т.е. «внутри» функции) никак не отразится на фактическом параметре («вне»).
- По ссылке – в функцию передаётся адрес параметра, то есть в роли формального параметра выступает сам фактический
  - Изменение формального параметра влияет на фактический параметр.
- Другие – см. самостоятельно.

Фактически, в С возможна передача параметров по значению, однако в качестве значения может выступать адрес переменной или функции.

Передача параметра по значению – пример

```
float getFahrFromCelsius(const float cels)
{
        cels = 5;
        const float k = 9.0 / 5.0;

        return k * cels + 32;
}
void main()
{
        int currentT = 25;
        float t = getFahrFromCelsius(currentT );

        printf("%f", t);
}
```

Изменение параметра cels никак не отразится на значении переменной – фактического параметра – currentT

Передача параметра по ссылке – пример

```
float getFahrFromCelsius(float *cels)
{
        (*cels) = 5;
        const float k = 9.0 / 5.0;

        return k * (*cels) + 32;
}
void main()
{
        float currentT = 25.0;
        float t = getFahrFromCelsius(&currentT );

        printf("%f", t);
}
```

В этом примере модификатор const перед параметром не позволил бы изменить значение cels.

Не стоит забывать, что в действительности в функцию передаётся значение – число (адрес) переменной типа float.

Список параметров переменной длины*

Список параметров переменной длины объявляется при помощи многоточия (“…”).

```
#include <stdarg.h>
#include <stdio.h>
double average(int num, ...) {
        va_list vaList;
        double sum = 0.0;
        int i;
        //инициализация списка с переменным числом параметров
        va_start(vaList, num);
        //последовательный доступ ко всем параметрам
        for (i = 0; i < num; i++)
                sum += va_arg(vaList, int);
        va_end(vaList);
        return sum / num;
}

void main()
{
        printf("Average of 1, 3, 4, 5 = %f\n", average(1, 2, 3, 4, 5));
}
```

*детально этот пример следует изучить самостоятельно

Рекурсивные функции

Рекурсия – описание объекта при помощи его самого.

Пример – фракталы, соотношения, заданные рекуррентным образом.

Рекурсивная функция – функция, вызывающая саму себя (напрямую или косвенно).

Опасны возникновением нехватки памяти, используемой при вызове функции (т.н. переполнение стека вызовов).

Существует хвостовая рекурсия – когда рекурсивный вызов в функции – это последняя выполняемая операция.

Могут быть оптимизированы – преобразованы компилятором в цикл.

Пример рекурсии – число Фибоначчи

$$F_n = \begin{cases} 1, \text{при } n = 1 \\ 1, \text{при } n = 2 \\ F_{n-1} + F_{n-2}, \text{при } n \geq 2 \end{cases}$$

```
unsigned int fib(unsigned int n){
  if (n < 2)
    return n;

 return fib(n - 1) + fib(n - 2);
}
```

Пример хвостовой рекурсии – число Фибоначчи

```
unsigned int fib(unsigned int n, unsigned int a, unsigned int b)
{
    if (0 == n)
        return a;
    if (1 == n)
        return b;
    return fib(n - 1, b, a + b);
}
```

Пример рекурсии в определении типов

```
typedef struct {
        int value;
        struct linkedListNode* next; /*указатель на узел, следующий за текущим узлом */
} linkedListNode;
```

Замечания по использованию функций

- Функции нужно передать все нужные ей значения
  double sin(void);
- Функция должна вернуть результат своей работы void sin(double)
  - Значение
    - Адрес
    - Структура
- Функция должна решать одну задачу
- Функция не должна выводить на печать результат своей работы (как правило)
  - Для x = 3.14 sin(x) = 0
- Буквально ничего!
  - Значение sin(x) вычислено успешно!

Побочные эффекты выражений

Любые изменения состояния программы, не связанные с конкретным значением, возвращаемым данной функцией, но инициированные изнутри неё, называются ***побочными эффектами.***

Пример –

- Любые действия ввода-вывода.
- Изменение глобальных переменных.

Золотое правило хорошего стиля:

все объекты, которые нужны функции для работы, она должна получать через свои параметры, а не использовать глобальные переменные!

Глобальные переменные

Глобальная переменная – переменная, к которой можно обратиться из любого (или почти любого) места в программе.

Могут использоваться для обмена данными между подпрограммами.

Создают зависимости разных частей кода от взаимного состояния.

Имена глобальных переменных могут совпадать с именами локальных переменных.

Широко используются, например, в многопоточном программировании.

Использование глобальных переменных

```
int isInputValid = 0;

void func1()
{
        //…
        isInputValid = 1;
        func2();
        if (isInputValid)  { … }
}

void func2()
{
        isInputValid = 0;
}
```

Указатели и адресная арифметика

Указатель – тип данных, предназначенный для хранения адресов объектов программы.

Если существует тип X, то существует и тип X*, хранящий адрес переменной типа X, а также

тип X**, хранящий адрес переменной, хранящий адрес переменной типа X...

В C используются типизированные указатели, но есть и нетипизированные (void).

Адресная арифметика

Внимание! Всегда будьте внимательны при выполнении арифметических операций с указателями.

Допустимые операции:

- Сложение с целыми числами.
- Получение адреса.
- Разыменование.
- Сравнение указателей.
- Вычитание указателей.

Пример работы с указателями (1 / 2)

```
int a = 2;              //объявляем переменную типа int
int *p_a = &a;          //объявляем указатель на a
*p_a = 10;              //обращаемся по адресу в p_a
```

&a – получение адреса при помощи оператора &

*pa – обращение «по указателю» при помощи оператора *, иначе говоря – разыменование.

```
int* a, b;      //Определён один указатель!
int *a, *b;     //Определено два указателя.
```

Пример работы с указателями (2/2)

```
int a, *p;
p = &a;
a = 100;       //прямая адресация
*p = 100;      //косвенная адресация
```

Немного казуистики:

```
int i;                          //переменная целого типа
const int ci = 1, c2 = 2;       //константы целого типа
int *pi;                        //указатель на переменную
const int *pci;                 //указатель на  константу
int* const cp = &i;             //константный указатель на переменную

cp = &c2;                       //Ошибка! Константу нельзя изменить!

const int* const cpc = &ci; //константный указатель на константу
```

Получение адреса и разыменование

Указатель – это обычная переменная, которая тоже где-то располагается в памяти, поэтому можно получить её адрес.

```
int a = 42, b = 0;
int *ptr;
int *p_a = &a;              //в p_a – адрес a
int **ptr_pa = &p_a;        //в ptr_pa – адрес p_a

ptr = *ptr_pa          //получили значение указателя *p_a;
b = *ptr;              //получили значение по ссылке из p_a
printf("%i", **ptr_pa);  //можно разыменовать сразу, 42
```

Сложение указателей с целыми числами (1/3)

```
int a = 10;
int *p = &a;
```

Выражение вида p + 1

Тип результата:

int* (указатель на целое)

Значение:

АДРЕС объекта типа int СЛЕДУЮЩЕГО за a

Общий случай: p + n

p + n * sizeof(int)

n может быть и отрицательным!

Сложение указателей с целыми числами (2/3)

Прибавление числа к указателю смещает его на столько же размеров того типа, который имеет указатель,

а не количество байт

Пример:

```
int a[5] = { 1, 2, 3, 4, 5 };        //массив из пяти элементов
int *p_a = a;                        //в p_a – адрес массива

printf("%i", *p_a);          //вывод: 1

p_a += 3;
printf("%i", *p_a);          //вывод: 4
```

Сложение указателей с целыми числами (3/3)

Частный случай – операторы инкремента и декремента

```
int a = 2;
int *p = &a;

*p++;      //инкремент указателя на sizeof(int), не на 1
*p--;      //адрес = адрес – sizeof(int)
```

Обратите внимание на приоритет операций!

```
(*p)++;          //инкремент значения по адресу p
(*p)--;          //декремент значения по адресу p
```

Вычитание указателей

При работе с двумя указателями, которые указывают на одну область памяти, можно

вычесть меньший из большего и

получить количество элементов в интервале

от меньшего включительно до большего не включая его.

Обратите внимание, количество элементов, а

не количество байт!

Вычитание указателей – пример

```
int a[] = { 1, 2, 3, 4, 5 };
int *p_a = a + 4;

printf("%i", p_a - a);      //4

int *p1, *p2;
...
```

p2 – p1

Тип значения: целое число.

Значение: количество объектов, расположенных между адресами, хранящимися в интервале между p2 и p1.

Сравнение указателей

```
int a = 42;                 //просто переменная
int *p_a = &a;              //ссылка на a
int *p_a2 = p_a + 5; //p_a2 ссылается на p_a + 5*sizeof(int)

printf("%i", p_a > p_a2);       //вывод: 0
printf("%i", p_a < p_a2);       //вывод: 1

printf("%i", p_a == p_a2);      //вывод: 0
printf("%i", p_a != p_a2);      //вывод: 1
```

Тип void*

void* указывает на данные неопределённого типа и размера.

С ним не пройдёт адресная арифметика (потому что размер данных, на которые он указывает, неизвестен).

Прежде, чем нормально работать с таким указателем, необходимо явно определить, на данные какого типа он указывает, конвертировав его в соответствующий тип-указатель:

```
void *a = (void*)4;
int *b = (int*)a;

a++  //ошибка компиляции
b++; //ОК, увеличение значения на sizeof(int)
```

NULL

В С с помощью препроцессора выделяется специальная константа с именем NULL для указателя, который «ведёт в никуда»

Если указатель равен NULL, то он не ссылается ни на какой корректный объект и обращаться по нему не следует.

NULL обычно равен нулю.

На самом деле, он указывает на какую-то ячейку

памяти с номером 0, но мы условились, что в ней ничего не лежит.

Указатели на функции

Указатель на функцию предназначен для хранения адресов функций, и их косвенного вызова.

```
// указатель на функцию, возвращающую целое
int (*pfi)(int);
// указатель на функцию, возвращающую указатель на int
int *(*pfpi)();
// функция, возвращающая указатель на функцию,
//возвращающую значение типа int
 int (*pfpfi())();
// функция, возвращающая указатель на массив  из трёх целых
int (*fai())[3];
 // массив из трёх указателей на функции,
// возвращающие значения типа int
int (*apfi[3])();
```

Применение указателей на функцию - сортировка

```
void sort(int *vec, int length, int(*comparer)(int, int))
```

```
int cmpByAbs(a, int b)
{
        return abs(a) - abs(b);
}

int simpleCompare(int a, int b)
{
        return a - b;
}
int reverseCompare(int a, int b)
{
        return b – a;
}
```

```
void foo()
{
        int(*cmp)(int, int) = cmpByAbs;
        sort(arr, a, cmp);

        sort(arr, a, simpleCompare);
}
```

функции, выполняющие сравнение различным образом, позволяют сортировать вектор по разным критериям.

Массивы в С

Массив – составной тип данных, предназначенный для хранения некоторого (конечного) количества переменных одного типа, имеющих общее имя.

Доступ к конкретной переменной осуществляется с использованием имени массива и номера (индекса) конкретной переменной в нём.

Линейный массив (вектор)

Для определение вектора необходимо указать:

1. Тип элементов вектора.
2. Имя вектора.
3. Число элементов в векторе.

type arrayName[*arraySize*];

```
int a[10];
```

Инициализация линейных массивов

```
int a[5] = { 1, 2, 3, 4, 5 };
int a[5] = { 1, 2, 3 };
int a[5] = { 0 };
int a[5] = { 1, 2, 3, 4, 5, 6 }; //Ошибка!
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Доступ к элементам массивов

Для доступа к элементу массива используется индексное выражение : array[i]

```
int a[5] = { 0 };
```

Имеют смысл: a[0], a[1], a[2], a[3], a[4].

Допустимы: a[100], a[-52], но:

1. бессмысленно;
2. может привести к серьёзным ошибкам в ходе работы программы.

Индексное выражение

В состав индексного выражения входят:

- указатель;
- целое число.

Вычисление индексного выражения:

1. указатель складывается с целым a + 3
2. Разыменование полученного значения *(a + 3)

   a[i] эквивалентно *(a + i)

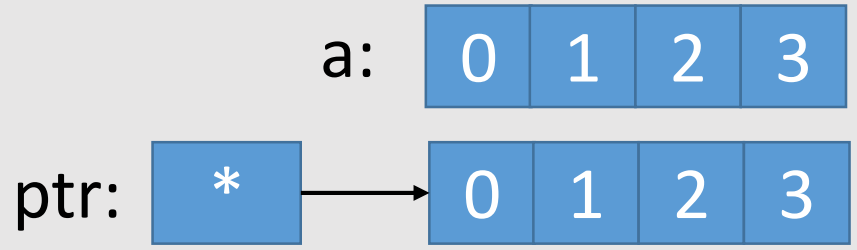
```
a[3] = -20;       //записать значение по адресу
int b = a[3];     //считать значение по адресу
```

Немного об имени массива

Имя массива (вектора) интерпретируется компилятором как константный указатель на тип, соответствующий типу элементов массива.

Значение этого константного указателя совпадает с адресом первого элемента массива (вектора).

В дальнейшем это константное значение используется для вычисления индексных выражений, в состав которых входит имя массива (вектора).



Можно:

```
int a[5] = { 0, 1, 2, 3 };
int *ptr = a;        //можно присвоить указателю адрес массива
ptr = &a;            //совершенно равносильная форма записи
```

Копирование линейных массивов

```
int a[5] = { 1, 2, 3, 4, 5 }, b[5];
```

Необходимо скопировать элементы вектора a в вектор b.

1. Наивное: a = b;

Ошибка! Присваивать значения константному указателю запрещено.

2. Поэлементное копирование в цикле:

```
int i = 0;
for (i = 0; i < 5; i++)
        b[i] = a[i];
```

3. С использованием функций стандартной библиотеки:

```
        memmove(b, a, 5 * sizeof(int));
```

Многомерные массивы

```
int a[5][10];
```

a – это вектор из пяти элементов, каждый из которых является вектором из десяти элементов целого типа.

```
int b[3][3];
int b[3][3]      = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
int b[][]        = { {1, 1, 1}, {0, 2, 2}, {1, 0, 3} };
```

Доступ к элементам массива int a[5][10]

a[i][j] – индексное выражение, которое вычисляется слева направо.

1. a + i
2. *(a + i)
3. *(a + i) + j
4. *(*(a + i) + j)

Переход к i-у вектору

Переход к j-му элементу i-го вектора

Вычисление индексного выражения:

a + 10 * sizeof(int) * i + sizeof(int) * j

Передача в функцию линейного массива.

Задача: найти максимальный элемент вектора.

```
int findMax(int a[], int n)
{
        int max = a[0], i = 1;

        for (i = 1; i < n; i++)
                if (a[i] > max)
                        max = a[i];

        return max;
}
```

```
void foo()
{
        int max, m[100];
        max = findMax(m, 100);
}
```

```
int findMax(int*, int);

int findMax(const int*, int);
```

Передача в функцию статически определённой матрицы

```
#define ncol 20

int mmax(int mas[][ncol], int n, int m) {
        int max = mas[0][0];
        int i = 0, j = 0;

        for (i = 0; i < n; i++) {
                for (j = 0; j < m; j++) {
                        if (mas[i][j] > max)
                                max = mas[i][j];
                }
        }

        return max;
}
```

Про #define и другие директивы препроцессора мы ещё поговорим ☺

```
void foo()
{
        int a[10][ncol];
        int m = 0;
        m = mmax(a, 10, ncol);
}
```

1. int mmax(int* mas, int n, int m);
2. mas[i][j] ⇨ mas[i * m + j]
3. m = mmax(&a[0][0], 10, ncol);

Работа с динамической памятью

- Статическое выделение памяти – этап компиляции.
- Динамическое выделение памяти – этап выполнения.

Для выделения памяти используются функции:

- malloc();
- calloc();
- realloc();

После того, как память больше не требуется, её необходимо освободить.

free() – освобождает память, либо помечает как свободную.

Динамическое выделение памяти под массив (1/3)

```
int length = 10, i = 0;
int *vector = (int*)malloc(sizeof(int) * length);

if (NULL == vector)
        //действия по обработке ошибки

//обнуление элементов
for (i = 0; i < length; i++)
        vector[i] = 0;

free(vector);   //освобождение использованной памяти
```

Динамическое выделение памяти под массив (2/3)

```
int length = 10, i = 0;
int *vector = (int*)calloc(length, sizeof(int));

if (NULL == vector)
        //действия по обработке ошибки

/*
Не требуется выполнять обнуление элементов массива,
так как функция calloc сделала это сама.
*/

free(vector);   //освобождение использованной памяти
```

Динамическое выделение памяти под массив (3/3)

```
int length = 10, i = 0, *ptr;
int *vector = (int*)malloc(sizeof(int) * length);

if (NULL == vector)
        //действия по обработке ошибки

//обнуление элементов
for (i = 0; i < length; i++)
        vector[i] = 0;
//Обратите внимание: старый указатель сохранён.
*ptr = (int*)realloc(vector, sizeof(int) * 100);
if (NULL == ptr)
        //действия по обработке ошибок.
else
        vector = ptr; //всё ОК, нам не нужен «висячий» (старый) vector
free(ptr);     //освобождение использованной памяти
```

Особенности функций-аллокаторов (1/2)

- malloc просто выделяет память указанного размера, значение которого не определено.
  - Принимает размер типа size_t (результат вычисления sizeof(T)).
  - Возвращает адрес выделенного блока, либо NULL, если память выделить не удалось.
- calloc выделяет память указанного размера и заполняет её нулями.
  - Принимает количество выделяемых блоков и размер одного.
  - Возвращаемый результат – аналогичен malloc.

Особенности функций-аллокаторов (2/2)

realloc

- принимает указатель на объект, а также размер нового объекта.
- освобождает старый объект (по переданному указателю)
- выделяет память указанного размера, содержимое старого объекта копируется:
  - если новый объект больше старого, то оно идентично
  - если новый объект меньше старого, то значение любых байт не определено.
- использование realloc вместо malloc / free может привести к падению производительности за счёт копирования.

Немного о sizeof, size_t

Функции-аллокаторы принимают аргументы типа size_t

size_t – это специальный тип данных, описывающий возвращаемое оператором sizeof(T) значение.

size_t – по сути, целое число без знака, но размер его зависит от платформы: оно может занимать 16 бит, 32 бита или 64.

Не стоит относиться к **size_t** как **unsigned int**, поскольку это приведёт к потере переносимости программы на другие платформы.

Динамическое выделение памяти для размещения матрицы m x n (1/2)

```
int cols = 3, rows = 2;

int **matrix, i = 0, j = 0;

if ((matrix = (int**)malloc(sizeof(int*) * rows)) == NULL) {
        //ошибка!
}

for (i = 0; i < rows; i++)
        matrix[i] = (int*)calloc(cols, sizeof(int));
//так следует освободить память
for (i = 0; i < rows; i++)
        free(matrix[i]);

free(matrix);
```

Динамическое выделение памяти для размещения матрицы m x n (2/2)

```
const int cols = 3, rows = 2;
int i = 0, j = 0;
int *matrix = (int*)malloc(rows * cols * sizeof(int));
if (NULL == matrix) {
        //ошибка!
}
for (i = 0; i < rows; i++)
{
        for (j = 0; j < cols; j++)
        {
                int offset = i * cols + j; //смещение от начала массива
                printf("%i ", *(matrix + offset));
        }
        printf("\n");
}
//…
free(matrix);
```