

Краткое содержание предыдущей серии

```
1
2 // подключение заголовочного файла
3 #include <stdio.h>
4
5 // Определение макроса
6 #define MATH_PI 3.141592653589793238
7
8 // Определение макроса с параметром
9 #define RAD_TO_DEG(arg) arg*180.0/MATH_PI
10 #define DEG_TO_RAD(arg) arg*MATH_PI/180.0
11
12
13 // функция с двумя аргументами и с возвращаемым значением типа int
14 int summ(int arg1, int arg2) {
15     return arg1 + arg2;
16 }
17
18
19 // функция без аргументов и без возвращаемого значения
20 void printDoubledValue(int value) {
21     int doubledValue = value*2;
22     printf("Печатаю удвоенное значение: ", doubledValue);
23     // return не обязателен
24 }
25
```

Краткое содержание предыдущей серии

```
26
27 // Функция main, с которой начинается программа
28 int main(){
29 ... int someVariable; // определение переменной.
30 ... int someVariable, someAnotherVariable; // определение нескольких переменных
31
32 ... someVariable = 42; // Присвоение значения переменной
33 ... someVariable = someAnotherVariable = 42; // присвоение значения переменной цепочкой
34
35 ... // определение и инициализация переменной
36 ... int yetAnotherVariable = 42 + 42 - 10; // математические операции
37
38 ... // вызов функции
39 ... int summedValue = summ(someVariable, someAnotherVariable);
40 ... summ(10, 15); // возвращаемое значение не обязательно забирать
41 ... printDoubledValue(12);
42
43 ... // Использование макросов
44 ... double circleRadius = 10.0;
45 ... double circleArea = MATH_PI * circleRadius * circleRadius;
46 ... // перед компиляцией заменится на
47 ... // double circleArea = 3.141592653589793238 * circleRadius * circleRadius
48
49
50 ... return 0;
51 }
52
```

Краткое содержание предыдущей серии

```
1
2#include <stdbool.h>
3#include <stdio.h>
4
5int main() {
6    int value = 10;
7
8    // Условный оператор
9    if (value > 10) {
10        printf("value больше 10ти");
11    } else {
12        printf("value не больше 10ти");
13    }
14
15    return 0;
16}
17
18
```

Так же мы рассмотрели:

- Азы работы в эклипсе;
- Базовые типы данных языка C;
- Преобразования типов и переполнения;
- Операторы (+, -, >>, ~, !);

Работа над ошибками

Базовые спецификаторы функций семейства `printf / scanf`

Код Формат

%c Символ

%d Десятичное целое число со знаком

%e Экспоненциальное представление числа (в виде мантиссы и порядка) (е на нижнем регистре)

%f Десятичное число с плавающей точкой

%s Символьная строка

%i Десятичное целое число без знака

%x Шестнадцатеричное без знака (строчные буквы)

%X Шестнадцатеричное без знака (прописные буквы)

%p Выводит указатель

%% Выводит знак процента

Это лишь базовые спецификаторы. Возможностей форматирования функций `printf / scanf` намного больше. Подробнее написано, например, на вики: <https://ru.wikipedia.org/wiki/Printf>

Работа над ошибками

Краткий (очень краткий) справочник по стандартной библиотеке

<code><stdbool.h></code>	Для булевых типов данных. (Появилось в C99)
<code><float.h></code>	Содержит заранее определенные константы, описывающие специфику реализации свойств библиотеки для работы с числами с плавающей точкой, как, например, минимальная
<code><stdint.h></code>	Для определения различных типов целых чисел. (Появилось в C99)
<code><stddef.h></code>	Для определения нескольких стандартных типов и макросов.
<code><stdio.h></code>	Реализует основные возможности ввода и вывода в языке Си. Этот файл содержит весьма важную функцию <code>printf</code> .
<code><stdlib.h></code>	Для выполнения множества операций, включая конвертацию, генерацию псевдослучайных чисел, выделение памяти, контроль процессов, окружения, сигналов, поиска и сортировки.
<code><string.h></code>	Для работы с различными видами строк.
<code><math.h></code>	Для вычисления основных математических функций
<code><limits.h></code>	Содержит заранее заданные константы, определяющие специфику реализации свойств целых типов, как, например, область допустимых значений (<code>_MIN</code> , <code>_MAX</code>).

Снова – стандартная библиотека намного больше. Опять, много можно почесть на вики:

[https://ru.wikipedia.org/wiki/Стандартная библиотека языка Си](https://ru.wikipedia.org/wiki/Стандартная_библиотека_языка_Си)

а лучше тут <http://www.cplusplus.com/reference/clibrary/>

Работа над ошибками

Пояснения к символам и строкам в Си

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("какой-то текст\n"); // в двойных кавычках задается строка
6     char a = 'a'; // в одинарных кавычках символ
7     char a_code = 97; // 97 - код символа 'a' в кодировке ASCII
8
9     if (a == a_code)
10         printf("'a' и 97 - это одно и тоже\n");
11
12     if ('b' - 1 == 'a') // можно делать арифметические операции с номерами символов
13         printf("'b' - 1 == a\n");
14
15
16     return 0;
17 }
18
```


Работа над ошибками

Таблица ASCII

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	_	127	7F	DEL

Обратите внимание, что в char у нас Помещается 255 вариантов символов, а таблица заполнена лишь до 127.

Кодировка ASCII не подразумевает иных символов, но все прочие кодировки, как правило используют ASCII и расширяют её своими символами с номерами от 128 до 255

Стандарт языка си ничего не упоминает о кодировках в которых он принимает символьные литералы, поэтому Во избежание проблем лучше использовать только английские.

(наш компилятор работает в кодировке utf-8)

Программирование на языке С

ЛЕКЦИЯ 2

Типы литералов и целочисленное деление

```
6 int main() {
7
8 ... {
9 ... // если аргументы целочисленные - деление тоже целочисленное
10 ... int x = 5;
11 ... int y = x/2; // y == 2
12
13 ... // остаток от деления можно получить при помощи оператора %
14 ... int z = 5%2; // z == 1
15 ... }
16
17
18 ... {
19 ... float x = 5;
20 ... float y = x/2; // y == 2.5
21 ... y = 5/2; // y == 2.0 (потому что 5 и 2 все равно целочисленные)
22 ... y = (float)5/2; // y == 2.5
23 ... y = 5.0f/2; // y == 2.5
24 ... }
25
26 ... return 0;
27 }
28
```

```
6 int main() {
7
8 ... {
9 ... double a = 5.0;
10 ... float b = 5.0f;
11 ... int c = 5;
12 ... long d = 5L;
13 ... }
14
15 ... // аналогично
16 ... {
17 ... double a = (double)5;
18 ... float b = (float)5;
19 ... int c = (int)5;
20 ... long d = (long)5;
21 ... }
22
23 ... return 0;
24 }
25
26
```

```
4 {
5 char a = 'a';
6 char b = 'b';
7 char c = 'c';
8 // ...
9
```

<- строковые литералы

Константы

Константа ничем не отличается от обычной переменной, кроме того, что она должна быть инициализированна при определении и ей нельзя присводить значение.

Чтобы сделать переменную константой нужно использовать модификатор **const**

```
1
2 int main() {
3
4     const int x = 10;
5     const int y; // ошибка, константа должна быть инициализированна
6
7     x = 20; // ошибка, нельзя изменять константу после инициализации
8     x += 20; // тоже ошибка
9
10    int a = x; // Можно читать значение константы
11    return 0;
12 }
13
```

Прочие модификаторы

Static, volatile

Особенности передачи аргументов в функции

```
1 #include <stdio.h>
2
3
4 void argModificationTest(int arg) {
5     printf("Аргумент в функции %d", arg);
6     arg = arg + 10;
7     printf("Аргумент в функции после изменения %d", arg);
8 }
9
10
11 int main() {
12     int x = 0;
13     printf("Аргумент в main %d", x);
14     argModificationTest(x);
15     printf("Аргумент в main после вызова функции: %d", x);
16
17     return 0;
18 }
19
20
21
```

Вывод:

Аргумент в main 0

Аргумент в функции 0

Аргумент в функции после
изменения 10

Аргумент в main после вызова
функции: 0

Область видимости переменных

- ▶ Переменные видны от их определения, до закрывающей скобочки блока { }, в котором они определены

```
5
6 int main() {
7
8     printf("y = %d", y); // Ошибка: y еще не определена
9     int y;
10
11     {
12         int x = 20;
13         printf("y = %d", y); // все в порядке
14     }
15
16     printf("x = %d", x); // ошибка: x здесь не видна
17     printf("y = %d", y); // все в порядке
18
19     return 0;
20 }
21
```


Область видимости переменных

- ▶ Имена переменных можно перекрывать на более глубоких вложенностях

```
5
6 int main() {
7
8     int x;
9     x = 0;
10
11     {
12         int x;
13         x = 20;
14
15         printf("x = %d", x);
16     }
17
18     printf("x = %d", x);
19
20     return 0;
21 }
22
```

- ▶ Какой будет вывод?

Область видимости переменных

- ▶ Переменные определенные внутри функций или на более глубоких уровнях вложения блоков { } называются **локальными**.
- ▶ Переменные определенные на самом первом уровне вложенности (вообще вне { } называются **глобальными**. Глобальные переменные видны от их определения, до конца файла.

```
6 // определение глобальной переменной
7 int globalX = 10;
8
9
10 void printGlobalX() {
11     printf("globalX в функции = %d", globalX);
12 }
13
14 int main() {
15     printf("globalX в main = %d", globalX);
16
17     globalX += 10;
18     printf("globalX в main = %d", globalX);
19
20     globalX += 10;
21     printGlobalX();
22
23     return 0;
24 }
25
26
27
```

globalX в main = 10

globalX в main = 20

globalX в функции = 30

Область видимости функций

- ▶ Подобно глобальным переменным функции видны от их определения до конца файла

```
1 #include <stdio.h>
2
3 void f1() {
4     printf("Вызвана функция f1");
5 }
6
7
8 void f2() {
9     f1();
10    printf("Вызвана функция f2");
11 }
12
13
14 void f3() {
15     f2();
16    printf("Вызвана функция f3");
17 }
18
19
20 int main() {
21     f3();
22
23     return 0;
24 }
```

Массивы и циклы

- Представим, что нам нужно сделать несколько измерений температуры и посчитать её среднее значение. Можно сделать так:

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 uint8_t getTemperature() {
5     uint8_t tempValue;
6     // ...
7     // операции с датчиком
8     // ...
9     return tempValue;
10 }
11
12 int main() {
13     uint8_t temp0 = getTemperature();
14     uint8_t temp1 = getTemperature();
15     uint8_t temp2 = getTemperature();
16     uint8_t temp3 = getTemperature();
17     uint8_t temp4 = getTemperature();
18
19     uint8_t averageTemp = (temp0 + temp1 + temp2 + temp3 + temp4);
20     averageTemp /= 5;
21
22     printf("Среднее значение температуры: %d", averageTemp);
23     return 0;
24 }
```

Массивы и циклы

- ▶ Гораздо лучше будет использовать массив и цикл

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define TEMP_ARRAY_SIZE 500
5
6 uint8_t getTemperature() {
7     uint8_t tempValue;
8     // ...
9     // операции с датчиком
10    // ...
11    return tempValue;
12 }
13
14 int main() {
15     uint8_t tempValues[TEMP_ARRAY_SIZE];
16
17     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
18         tempValues[i] = getTemperature();
19     }
20
21     uint8_t averageTemp = 0;
22     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
23         averageTemp += tempValues[i];
24     }
25
26     averageTemp /= TEMP_ARRAY_SIZE;
27     printf("Среднее значение температуры: %d", averageTemp);
28     return 0;
29 }
```

Массив – это набор переменных
Одного и того же типа. Объявляется как
Переменная, но в конце указывается
размер
Массива в скобках [].

Можно использовать модификаторы
переменных (например **const**).

Для доступа к элементу массива нужно
написать его имя и затем в скобках []
номер элемента. Нумерация с нуля.

Массивы и циклы

Многомерные массивы и инициализация.

```
5
6 int main() {
7
8     // многомерные массивы
9     int twoDimensionalArray[5][10];
10    int threeDimensionalArray[5][10][15];
11
12    // доступ к элементу
13    printf("arrayValue: %d", twoDimensionalArray[0][0]);
14    printf("arrayValue: %d", threeDimensionalArray[0][0][0]);
15
16    // инициализация массивов
17    int array1[10]; // в элементах массива что угодно;
18    int array2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // все элементы массива указаны
19    int array3[10] = {1, 2, 3}; // первые три элемента массива 1, 2, 3 - остальные ноль
20
21    // инициализация многомерных массивов
22    int array4[5][5] = {
23        {0, 1, 2, 3, 4},
24        {0, 1, 2, 3},
25        {}
26    };
27
28    return 0;
29 }
30
```

Массивы нельзя
использовать как
аргументы функций

Массивы и циклы

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define TEMP_ARRAY_SIZE 500
5
6 uint8_t getTemperature() {
7     uint8_t tempValue;
8     // ...
9     // операции с датчиком
10    // ...
11    return tempValue;
12 }
13
14 int main() {
15     uint8_t tempValues[TEMP_ARRAY_SIZE];
16
17     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
18         tempValues[i] = getTemperature();
19     }
20
21     uint8_t averageTemp = 0;
22     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
23         averageTemp += tempValues[i];
24     }
25
26     averageTemp /= TEMP_ARRAY_SIZE;
27     printf("Среднее значение температуры: %d", averageTemp);
28     return 0;
29 }
```

Циклов в Си несколько: **for**, **while** и **do-while**. Они аналогичны паскалевским `for`, `while` и `repeat-until`.

```
for(
    то, что происходит перед первой итерацией
;
    условие == true
;
    то, что происходит после каждой итерации
)
{
    Тело цикла
}
```

Массивы и циклы

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <stdbool.h>
4
5
6 bool isLegDeployed() {
7     bool status;
8     // ... работа с датчиками
9     return status;
10 }
11
12 void stepLeg(int stepsCount) {
13     // ... работа с двигателем ...
14 }
15
16 int main() {
17
18     while (!isLegDeployed()) {
19         stepLeg(1);
20     }
21
22     // do-while почти полностью аналогичен
23     // но отрабатывает как минимум один раз
24     do {
25         stepLeg(1);
26     } while (isLegDeployed());
27
28     return 0;
29 }
```

while (условие == true) {
 тело цикла
}

do {
 тело цикла
} while (условие == true);

Массивы и циклы

```
5 int main() {  
6  
7     bool someEventFlag;  
8  
9     // бесконечный цикл  
0     while(1) {  
1         if (someEventFlag) {  
2             break;  
3         }  
4     }  
5  
6  
7     while(1) {  
8         if (someEventFlag)  
9             continue;  
0  
1         // действия, которые не произойдут, если continue сработал  
2     }  
3  
4     return 0;  
5 }  
6
```

Специальные операции в циклах

break == безусловный выход из цикла
continue == переход к следующей итерации

Массивы и циклы

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define TEMP_ARRAY_SIZE 500
5
6 uint8_t getTemperature() {
7     uint8_t tempValue;
8     // ...
9     // операции с датчиком
10    // ...
11    return tempValue;
12 }
13
14 int main() {
15     uint8_t tempValues[TEMP_ARRAY_SIZE];
16
17     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
18         tempValues[i] = getTemperature();
19     }
20
21     uint8_t averageTemp = 0;
22     for (int i = 0; i < TEMP_ARRAY_SIZE; i++) {
23         averageTemp += tempValues[i];
24     }
25
26     averageTemp /= TEMP_ARRAY_SIZE;
27     printf("Среднее значение температуры: %d", averageTemp);
28     return 0;
29 }
```

В этой программе есть логическая (не синтаксическая) ошибка. Найдите её

Массивы и циклы

Строковые литералы в двойных кавычках – это на самом деле массивы

```
5
6 // эти определения эквивалентны
7 const char string_literal[] = "Какой-то текст";
8 const char string_literal2[] = {
9     'К', 'а', 'к', 'о', 'й', '-', 'т', 'о',
10    ' ',
11    'т', 'е', 'к', 'с', 'т',
12    0
13 };
14
15 printf(string_literal);
16 printf(string_literal2);
17
```

Одиночные символы указываются в одинарных кавычках

Структуры

```
4
5 int main() {
6
7     struct {
8         int x;
9         double y;
10        float z;
11        int a[100];
12    } value;
13
14    value.x = 0;
15    value.y = 1.0;
16    value.z = 2.0f;
17    value.a[0] = 10;
18    value.a[1] = 20;
19
20
21    return 0;
22 }
23
```

Простая структура

```
5 int main() {
6
7     struct {
8         int x;
9         struct {
10            int xx;
11            double yy;
12            int zz;
13        } innerStruct;
14    } value;
15
16    value.x = 0;
17    value.innerStruct.xx = 10;
18    value.innerStruct.yy = 10;
19    // и так далее
20
21    return 0;
22 }
23
```

Вложенная структура

```
6 int main() {
7     struct {
8         int x;
9         double y;
10        float z;
11        int a[100];
12    } value = {1, 2.0, 3.0f, {10, 20, 30}};
13
14
15
16    return 0;
17 }
18
19
```

Инициализация структур

Со структурами как с любыми типами можно использовать модификаторы, например **const**

Структуры

Использование структуры и typedef

```
5 // пример простого typedef
6 typedef int temp_t;
7
8 // пример typedef структуры
9 typedef struct {
10     int x, y, z;
11 } my_struct_t;
12
13
14 // функция, возвращающая структуру
15 my_struct_t getStruct() {
16     my_struct_t retval;
17     retval.x = 10;
18     return retval;
19 }
20
21 // функция, принимающая структуру как аргумент
22 void useStruct(my_struct_t value) {
23     printf("struct value x = %d", value.x);
24 }
25
26
27 int main() {
28     my_struct_t myStruct = getStruct();
29     useStruct(myStruct);
30
31     return 0;
32 }
```

Модель памяти языка C



Модель памяти языка C

Работа стека

```
⊖ uint16_t getSum(uint8_t arg1, uint16_t arg2) {  
    » uint16_t retval := arg1+arg2;  
    » return retval;  
}  
  
⊖ int main() {  
    » uint8_t a := 0;  
    » uint16_t b := 10;  
  
    » {  
    »     » uint16_t c,d;  
    »     » uint32_t e := c + d;  
    » }  
  
    » uint16_t sum := getSum(a, b);  
  
    » return 0;  
}
```

0xFFFF1	
0xFFFF2	
0xFFFF3	
0xFFFF4	
0xFFFF5	
0xFFFF6	
0xFFFF7	
0xFFFF8	
0xFFFF9	
0xFFFFA	
0xFFFFB	
0xFFFFC	
0xFFFFD	uint16_t b
0xFFFFE	
0xFFFFF	uint8_t a

Код программы

Состояние стека

Модель памяти языка C

Работа стека

```
uint16_t getSum(uint8_t arg1, uint16_t arg2) {
    uint16_t retval := arg1+arg2;
    return retval;
}

int main() {
    uint8_t a := 0;
    uint16_t b := 10;

    {
        uint16_t c, d;
        uint32_t e := c + d;
    }

    uint16_t sum := getSum(a, b);

    return 0;
}
```

0xFFFF	uint8_t a
0xFFFE	uint16_t b
0xFFFD	uint16_t b
0xFFFC	uint16_t c
0xFFFB	uint16_t c
0xFFFA	uint16_t d
0xFFFF9	uint16_t d
0xFFFF8	uint32_t e
0xFFFF7	
0xFFFF6	
0xFFFF5	uint32_t e
0xFFFF4	
0xFFFF3	
0xFFFF2	
0xFFFF1	

Код программы

Состояние стека

Модель памяти языка C

Работа стека

```
uint16_t getSum(uint8_t arg1, uint16_t arg2) {  
    uint16_t retval := arg1+arg2;  
    return retval;  
}  
  
int main() {  
    uint8_t a := 0;  
    uint16_t b := 10;  
  
    {  
        uint16_t c, d;  
        uint32_t e := c + d;  
    }  
  
    uint16_t sum := getSum(a, b);  
  
    return 0;  
}
```

0xFFFF	uint8_t a
0xFFFE	uint16_t b
0xFFFFD	uint16_t sum
0xFFFFC	
0xFFFFB	
0xFFFFA	
0xFFFF9	
0xFFFF8	
0xFFFF7	
0xFFFF6	
0xFFFF5	
0xFFFF4	
0xFFFF3	
0xFFFF2	
0xFFFF1	

Код программы

Состояние стека

Модель памяти языка C

Работа стека

```
⊖ uint16_t getSum(uint8_t arg1, uint16_t arg2) {  
    » uint16_t retval := arg1+arg2;  
    » return retval;  
}  
  
⊖ int main() {  
    » uint8_t a := 0;  
    » uint16_t b := 10;  
  
    » {  
    »     » uint16_t c,d;  
    »     » uint32_t e := c + d;  
    » }  
  
    » uint16_t sum := getSum(a, b);  
  
    » return 0;  
}
```

0xFFEC	uint16_t retval
0xFFED	
0xFFEF	uint16_t arg2
0xFFEF	uint8_t arg1
0xFFF0	*return_addr
0xFFF1	
0xFFF2
0xFFF3	R6
0xFFF4	R5
0xFFF5	R4
0xFFF6	R3
0xFFF7	R2
0xFFF8	R1
0xFFF9	R0
0xFFFA	PC
0xFFFB	uint16_t sum
0xFFFC	
0xFFFD	uint16_t b
0xFFFE	
0xFFFF	uint8_t a


Код программы

Состояние стека

Модель памяти языка C

Работа стека

```
⊖ uint16_t getSum(uint8_t arg1, uint16_t arg2) {  
    » uint16_t retval := arg1+arg2;  
    » return retval;  
}  
  
⊖ int main() {  
  
    » uint8_t a := 0;  
    » uint16_t b := 10;  
  
    » {  
    »     » uint16_t c,d;  
    »     » uint32_t e := c + d;  
    » }  
  
    » uint16_t sum := getSum(a, b);  
  
    » return 0;  
}
```



0xFFFF	uint8_t a
0xFFFE	uint16_t b
0xFFFFD	uint16_t sum
0xFFFFC	
0xFFFFB	
0xFFFFA	
0xFFFF9	
0xFFFF8	
0xFFFF7	
0xFFFF6	
0xFFFF5	
0xFFFF4	
0xFFFF3	
0xFFFF2	
0xFFFF1	

Код программы

Состояние стека

Указатели языка C

Доступ к памяти через указатели

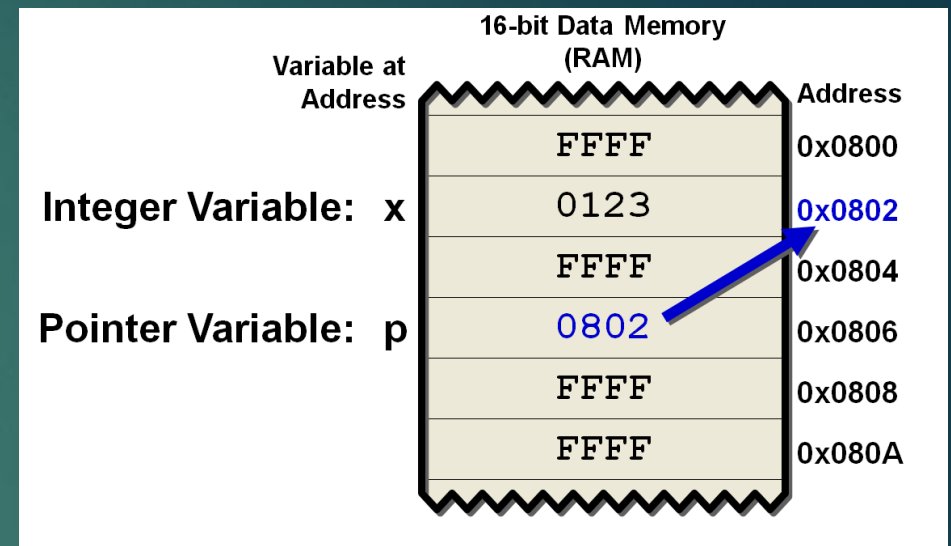
```
int main() {  
    int x;  
    int *p;  
  
    p = &x;  
    printf("p = %p\n", p);  
  
    x = 0;  
    printf("x до изменения = %d\n", x);  
  
    *p = 10;  
    printf("x после изменения = %d\n", x);  
  
    return 0;  
}
```

Вывод программы:

```
p = 0x7ffed142f464  
x до изменения = 0  
x после изменения = 10
```

Указатель, это переменная, которая хранит в себе адрес другой переменной определенного типа.

Фактически, указатель это целое число типа **size_t** из файла `<stddef.h>` (которое как правило определено как: **typedef unsigned int size_t**).



Указатели языка C

Базовые операции с указателями

```
typedef struct {  
    int x,y,z;  
} CustomStruct;  
  
int main() {  
    // объявление указателей  
    int *ptr;  
    double *ptr1, *ptr2;  
    CustomStruct *structPtr;  
  
    // операция взятия адреса  
    // и инициализация указателей  
    int x;  
    ptr = &x;  
    ptr1 = 0x10;  
  
    // Разыменование указателя  
    *ptr = 10;  
    int y = *ptr + 10;  
  
    return 0;  
}
```

Определяется указатель как
(Тип) * (имя_указателя);

Для указателей определена операция
разыменовывания. Она описывается
в коде как *имя_указателя.

Разыменовывания указателя
возвращает объект, на который он
указывает.

Для всех переменных определена
операция взятия адреса. Эта
операция возвращает адрес
переменной в памяти

Указатели языка C

Указательная арифметика и массивы

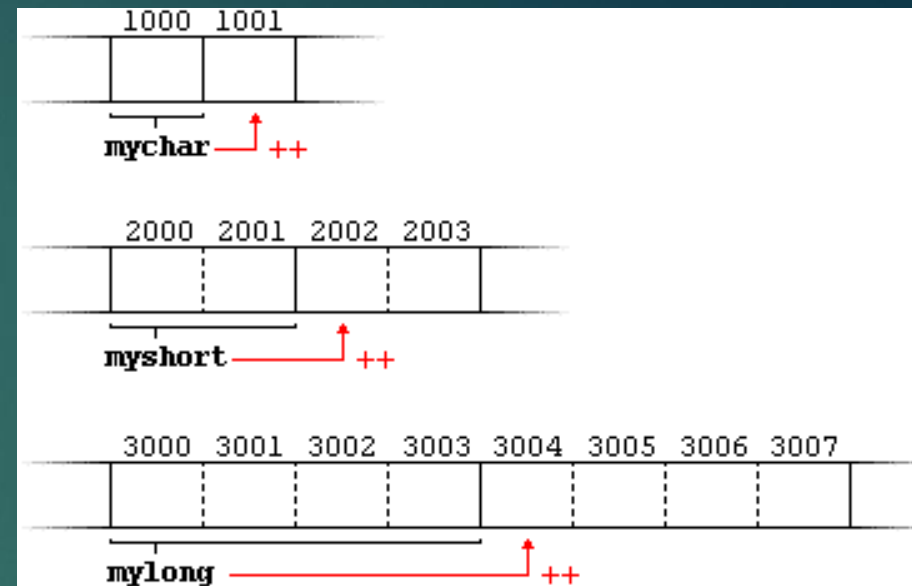
```
#include <stdint.h>
#include <stdbool.h>

int main() {
    // Указательная арифметика
    int a;
    int * ptr = &a;
    ptr++; // теперь ptr показывает на int следом за a

    // Элементы массива располагаются в памяти подряд
    // а сам массив - это фактически указатель на его начало
    int array[10];
    int * arrayPtr = array;

    // Доступ к элементу массива через указатели
    bool isTrue = (arrayPtr+5 == &array[5]);
    // или *(arrayPtr+5) и array[5] это одно и тоже

    // Указатели даже можно использовать как массивы и наоборот
    int elem = arrayPtr[5];
    // одно и тоже, что
    elem = *(array+5);
}
```



Указатели языка C

Преобразования типов указателей

```
6 int main() {
7     // uint32_t на стеке
8     uint32_t x = 0x12345678;
9     // указатель uint32_t* на него
10    uint32_t *intPtr = &x;
11
12    // указатель uint8_t* на ту же область памяти
13    uint8_t *bytesPtr = (uint8_t *)intPtr;
14
15    // печатаем байты uint8_t
16    for (size_t i = 0; i < sizeof(x); i++) {
17        printf("bytesPtr[%d] @ %p = 0x%X\n", i, bytesPtr+i, *(bytesPtr+i));
18    }
19
20
21    return 0;
22 }
```

```
<terminated> example [C/C++ Application] /home/snork/p
bytesPtr[0] @ 0x7ffd7550dda4 = 0x78
bytesPtr[1] @ 0x7ffd7550dda5 = 0x56
bytesPtr[2] @ 0x7ffd7550dda6 = 0x34
bytesPtr[3] @ 0x7ffd7550dda7 = 0x12
```

<- вывод программы

Указатели языка C

Указатель void *

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <stdbool.h>
4
5 // Структура статуса зонда
6 typedef struct {
7     uint8_t legsSensorOk: 1, // Статус выдвижения ног
8     parachuteSensorOk: 1, // Статус парашюта
9     radioModuleOk: 1, // Статус радиомодуля
10    thermometerOk: 1, // Статус термометра
11    pressuremeterOk: 1, // Статус барометра
12    gpsModuleOk: 1, // Статус GPS-модуля
13 };
14
15 uint16_t position[3]; // Положение WGS84 [x, y, z]
16 uint16_t velocity[3]; // Скорость WGS84 [vx, vy, vz]
17 } SpaceshipState;
18
19
20 void printBytes(void *dataPtr, size_t dataSize) {
21     // C void* нельзя совершать математических операций
22     // и разыменовывать, поэтому преобразуем его в uint8_t*
23     uint8_t *bytesPtr = (uint8_t *)dataPtr;
24
25     for (size_t i = 0; i < dataSize; i++) {
26         printf("dataBytes[%zd] = 0x%02X\n", i, bytesPtr[i]);
27     }
28 }
29
30 int main() {
31     SpaceshipState state = {1, 0, 1, 0, 1, 0, {1, 2, 3}, {4, 5, 6}};
32
33     // любые указатели неявно преобразуются в void*
34     // поэтому явное преобразование (void*) тут не нужно
35     printBytes(&state, sizeof(state));
36
37     return 0;
38 }
39
```

С указателем void * нельзя совершать математических операций и операций разыменовывания (компилятор укажет на ошибку)

Это чистая абстракция – указатель указывающий на «нечто»

Вывод программы:

```
<terminated> example [C/C++ Application] /home/snork/prog/
dataBytes[0] = 0x95
dataBytes[1] = 0xC2
dataBytes[2] = 0x01
dataBytes[3] = 0x00
dataBytes[4] = 0x02
dataBytes[5] = 0x00
dataBytes[6] = 0x03
dataBytes[7] = 0x00
dataBytes[8] = 0x04
dataBytes[9] = 0x00
dataBytes[10] = 0x05
dataBytes[11] = 0x00
dataBytes[12] = 0x06
dataBytes[13] = 0x00
```

Указатели языка C

Доступ к полям структуры через указатель и «возвращаемые» аргументы функций

```
5 typedef struct {  
6     int a, b, c, d;  
7 } MyCustomStruct;  
8  
9  
10 void printStructFieldAndInc(MyCustomStruct *ptr) {  
11     printf("field a = %d\n", ptr->a);  
12     // для доступа к полям структуры через указатель можно использоваться символ -> вместо .  
13     ptr->a = 10; // тоже самое что и (*ptr).a = 10;  
14 }  
15  
16 int main() {  
17     // выделение памяти под пользовательскую структуру  
18     MyCustomStruct data = {0, 1, 2, 3};  
19     printf("Поля структуры до вызова %d %d %d %d\n", data.a, data.b, data.c, data.d);  
20     printStructFieldAndInc(&data);  
21     printf("Поля структуры после вызова %d %d %d %d\n", data.a, data.b, data.c, data.d);  
22  
23     return 0;  
24 }  
25
```

В отличие от изменений аргументов «переданных по значению», изменения аргументов переданных по указателю возвращаются в подпрограмму верхнего уровня. Сам указатель при этом не меняется, так как он передается «по значению».

Указатели языка C

Указатели и модификатор **const**

В некоторых случаях, помимо возможности «возвращения» изменений аргументов передача аргументов по указателю еще и более эффективна в плане производительности и объемов памяти.

Например, при передаче большой структуры как аргумента «по значению», в стек копируются все её поля. При передаче структуры «по указателю» копируется только лишь указатель на нее.

Если «возвращение изменений» аргумента из функции при этом является не желательным, его можно явно запретить, объявив аргумент указателем на константу

```
4
5 typedef struct {
6     int a,b,c,d;
7 } MyCustomStruct;
8
9 void cantTouchThis(const MyCustomStruct * constPtr) {
10     int innerValue = constPtr->a; // все в порядке
11     constPtr->a = 10; // Ошибка, нельзя изменять константу
12
13     // Если для логики программы нужны изменения в структуре - можно сделать локальную копию и работать с ней
14     MyCustomStruct copy = *constPtr;
15     constPtr = &copy; // изменения указателя (а не того, что на он указывает) обратно не передаются
16 }
```


Указатели языка C

Указатели и модификатор **const**

При том, что объект, на который указывается указатель обозначенный как **const T *** **ptr** изменять нельзя, сам указатель при этом изменять можно (например переуказать на другой объект в памяти).

Эту возможность так же можно ограничить, но для этого нужно указать модификатор **const** после *

```
4
5 int main() {
6     int value1, value2;
7
8     const int * constPtr = &value1;
9     *constPtr = 10; // ошибка, нельзя изменить константу
10    constPtr = &value2; // перенаправить сам указатель при этом можно
11
12    int * const ptrConst = &value1;
13    *ptrConst = 10; // без проблем. Значение, на которое указывает указатель? не защищено как константа
14    ptrConst = &value2; // А вот это нельзя. Указатель константен
15
16    const int * const constPtrConst = &value1;
17    *constPtrConst = 10; // нельзя
18    constPtrConst = &value2; // тоже нельзя
19
20    return 0;
21 }
```


Указатели языка C

Значение NULL

Это специальный макрос, определенный в стандартной библиотеке, который обозначает указатель «в никуда». Такой указатель нельзя разыменовывать.

Как правило, под значением NULL используется обычный 0

При помощи этого значения удобно делать опциональные аргументы функций.

Указатели языка C

Указатель на указатель

Поскольку указатель это тоже переменная и тоже хранится в памяти – его адрес так же можно взять. Получится тип «указатель на указатель», который определяется как

тип ** имя_указателя

```
5 int main() {  
6     int value; // int на стеке  
7     int *valuePtr = &value; // указатель на value  
8     int **valuePtrPtr = &valuePtr; // указатель на указатель на value  
9  
10    return 0;  
11 }  
12
```

Полная аналогия двумерных массивов. Используется редко, либо в случаях, когда функция в аргументе должна вернуть указатель, тогда нужно указатель передать по указателю, либо при передаче двумерного массива в функцию.

Двумерные массивы (и соответственно указатели на указатели) это как правило списки строк, так как строка это уже массив типа char.

Указатели языка C

Указатель на указатель на указатель

Поскольку указатель на указатель это тоже переменная для которой определена операция взятия адреса...



Допустимыми являются конструкции

```
int *** ptr;
```

И даже `int ***** ptr;`

На практике такие указатели, как и массивы размером с количеством измерений более двух, используются крайне редко (считай не используются вовсе)

Работа с кучей

Для управления памятью в куче нужен специальный программный компонент – «аллокатор».

Это сложная программа, которая управляет динамическими переменными, создаваемыми и удаляемыми во время выполнения программы.



Работа с кучей

Для доступа к куче используются две функции из файла **<stdlib.h>**

```
void * malloc(size_t memBlockSize);  
void free(void * memBlockSize);
```

malloc (от memory allocate) выделяет в куче блок памяти указанного размера и возвращает на него «обезличенный» (void*) указатель. Если выделение памяти не удалось (скорее всего это значит, что она просто закончилась) **malloc** вернет NULL

Когда выделенный блок становится не нужен приложению, оно должно вызвать функцию **free** и передать ей указатель на не нужный блок. После этого блок возвращается в кучу и может быть заново аллокирован.

Из-за высоких накладных расходов в плане производительности и проблемы фрагментации памяти использование кучи не рекомендуется в приложениях для встраиваемых устройств.

Работа с кучей

Пример

```
1 #include <stdlib.h>
2
3
4 typedef struct {
5     int a,b,c,d;
6 } MyCustomStruct;
7
8
9 int main() {
10     // выделение памяти под пользовательскую структуру
11     MyCustomStruct* structPtr = (MyCustomStruct*) malloc(sizeof(MyCustomStruct));
12
13     // ... работа со structPtr;
14
15     // освобождение памяти
16     free(structPtr);
17 }
```

Сам по себе блок памяти не освободится (если не вызвана free). Ошибки с неосвобожденными блоками памяти называются «утечками памяти» и являются одними из самых трудно устранимых ошибок в программировании на С/С++ и других языках с подобной моделью памяти

Опасность указателей

```
3
4 int main() {
5     int array[5];
6     array[0] = 1;
7     array[1] = 1;
8     array[2] = 1;
9     array[3] = 1;
10    array[4] = 1;
11    array[5] = 1; // UNDEFINED BEHAVIOR
12
13
14    int *x = 0x55CC;
15    *x = 10; // ???
16
17    return 0;
18 }
19
```

Помимо опасности с утечками памяти, указатели опасны сами по себе. Чтение и запись по указателю, который указывает непонятно куда может привести к самым неожиданным ошибкам, которые очень тяжело отлавливать.

Поэтому при работе с указателями и массивами нужно быть предельно внимательным.

Выравнивание структур

Компилятор может вставлять паразитные поля в структуры между её членами для оптимизации обращений процессора к памяти, занимаемой ими. Правила по которым происходит это выравнивание достаточно сложны и зависят от компилятора и его настроек. Один из способов «борьбы» с этим – использование директивы компилятора **#pragma pack**

Хорошая статья на тему:

<http://habrahabr.ru/post/142662/>

```
1
2
3
4
5
6 #include <stdio.h>
7 #include <stdint.h>
8
9 typedef struct {
10     ...uint8_t x;
11     ...uint16_t y;
12     ...uint32_t z;
13     ...uint16_t a;
14 } CustomStruct;
15
16 int main() {
17     ...printf("sizeof = %zd\n", sizeof(CustomStruct));
18
19     ...return 0;
20 }
21
```

Вывод: sizeof = 12

```
1
2
3 #include <stdio.h>
4 #include <stdint.h>
5
6
7 #pragma pack(push, 1)
8
9 typedef struct {
10     ...uint8_t x;
11     ...uint16_t y;
12     ...uint32_t z;
13     ...uint16_t a;
14 } CustomStruct;
15
16 #pragma pack(pop)
17
18 int main() {
19     ...printf("sizeof = %zd\n", sizeof(CustomStruct));
20
21     ...return 0;
22 }
23
```

Вывод: sizeof = 9