

Краткое содержание предыдущих серий

Работа с макросами и функциями

```
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 предыдущих серий

Работа с переменными

```
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 предыдущих серий

Операторы ветвления

```
1
2 #include <stdbool.h>
3 #include <stdio.h>
4
5 int 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
```

```
7
8 #include <stdio.h>
9 #include <stdbool.h>
10
11 int main()
12 {
13     int value;
14
15     switch (value)
16     {
17         case 9:
18         case 10:
19             printf("value равно 10 или 8\n");
20             break;
21
22         case 12:
23             printf("я сработаю только если value == 12\n");
24             /* no break */
25         case 14:
26             printf("я сработаю если value == 12 или 14\n");
27
28         case 16:
29             printf("я сработаю только если value == 12\n");
30             break;
31
32         default:
33             printf("value - это что-то непонятное");
34             break;
35     }
36
37     return 0;
38 }
```

Краткое содержание предыдущих серий

Массивы и циклы

```
10
11 int main()
12 {
13     // одномерный массив
14     int array1[10];
15
16     // многомерный массив
17     int array2[10][20][30];
18
19     // инициализация массивов
20     int array3[] = {0};
21     int array4[5] = {0, 2, 3};
22     int array5[5][3] = {
23         {0, 1, 2},
24         {0, 1},
25     };
26
27     // доступ к элементам массива
28     array1[5] = 10;
29     printf("%d", array1[2][4]);
30 }
31
```

```
8 #include <stdio.h>
9 #include <stdbool.h>
10
11 int main()
12 {
13     int i;
14
15     // =====
16     for (i = 0; i < 10; i++) {
17         printf("i = %d\n", i);
18     }
19
20     // =====
21     i = 0;
22     while (i < 10) {
23         printf("i = %d\n", i);
24         i += 1;
25     }
26
27     // =====
28     i = 0;
29     do {
30         printf("i = %d\n", i);
31     } while (i < 10);
32
33
34     return 0;
35 }
36
```

Краткое содержание предыдущих серий

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

Строки

```
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     return 0;
16 }
17
18
```

```
8 #include <limits.h>
9
10 int main()
11 {
12     char a = 'a';
13     char b = a + 1;
14 }
15
16 char to_upper(char target)
17 {
18     return target + ('A' - 'a');
19 }
20
21
22
```

Краткое содержание предыдущей серии

Так же мы рассмотрели:

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

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/>

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
```

<- СИМВОЛЬНЫЕ
литералы

строковые ->
литералы

```
int main()
{
... const char *string = "это строка!";
... const char string_too[] = "это тоже строка";
}
```


Модификаторы типов

const – запрещает изменение значения

```
13 {  
14 ... const int x = 10; // константа должна быть инициализированна  
15 ... x = 50; // так нельзя  
16
```

signed, unsigned – задает «знаковость» целочисленных типов

```
13 {  
14 ... signed char x = 10; // тоже что и int8_t [0, 255]  
15 ... unsigned char y; // тоже, что и uint8_t [-128, 127]  
16  
17 ... signed float z; // только для целых типов  
18
```

volatile – запрещает оптимизации (грубо говоря)

```
...  
... for (int i = 0; i < 10000; i++)  
... {  
... .. volatile int x = 0; // не будет оптимизированно  
... }
```

Модификаторы типов

static – бесконечное время жизни для локальной переменной

```
11
12 int f()
13 {
14     static int x = 0; // инициализируется лишь однажды
15     x++;
16     return x;
17 }
18
19
```

Программирование на языке С

ЛЕКЦИЯ 2

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ

```
8 #include <stdio.h>
9 #include <math.h>
10
11 // создание нового типа на базе float
12 typedef float velocity_t;
13
14
15 float norm(velocity_t x, velocity_t y, velocity_t z) {
16     return sqrtf(x*x + y*y + z*z);
17 }
18
19 int main()
20 {
21     // используем новый тип
22     velocity_t v[3] = {0.f, 0.f, 0.f};
23     velocity_t v_norm = norm(v[0], v[1], v[2]);
24     printf("%f", v_norm);
25
26     return 0;
27 }
28
```

Пользовательский тип **velocity_t** – тип для хранения значений скорости.

Если на дальнейших этапах разработки ПО, мы решим заменить тип для скорости, это легко сделать изменив **typedef**

Работает почти как макрос

Перечисления

```
10
11 // Перечисление дней недели
12 enum {
13     ... MONDAY = 0, // можно присваивать конкретные значения
14     ... TUESDAY, ... // если значение не присвено явно - оно увеличивается на 1 от предыдущего
15     ... WEDNESDAY,
16     ... THURSDAY,
17     ... FRIDAY = 100,
18     ... SATURDAY, // = 101
19     ... SUNDAY // = 102
20 };
21
```

```
... // значения перечисления неявно приводятся в int
... int x = MONDAY;
... int y = MONDAY + 10;
```

Перечисления удобны для работы с данными, возможные значения которых можно «перечислить»

```
30
31 ... // часто используются со switch
32 ... switch (x)
33 ... {
34     ... case MONDAY:
35         ... printf ("сегодня понедельник");
36         ... break;
37     ... case TUESDAY:
38     ... case WEDNESDAY:
39     ... case THURSDAY:
40     ... case FRIDAY:
41         ... printf ("сегодня будний день");
42         ... break;
43
44     ... case SATURDAY:
45     ... case SUNDAY:
46         ... printf ("сегодня выходной");
47     ... }
```


Перечисление как тип

```
10
11 typedef enum {
12     ... MONDAY = 0,
13     ... TUESDAY,
14     ... WEDNESDAY,
15     ... THURSDAY,
16     ... FRIDAY = 100,
17     ... SATURDAY,
18     ... SUNDAY
19 } day_of_the_week_t;
20
21
22 float calc_business(day_of_the_week_t day)
23 {
24     ... // .....
25     ... return 0.4;
26 }
27
```

```
28
29 int main()
30 {
31     ... // можно так
32     ... day_of_the_week_t the_day = MONDAY;
33     ... calc_business(the_day);
34
35     ... // к сожалению так тоже можно
36     ... the_day = 10;
37     ... calc_business(1000);
38
39
40     ... printf("%d", the_day);
41 }
```

Структуры

```
7
8 int main()
9 {
10
11     struct
12     {
13         int32_t a;
14         int8_t b;
15         uint64_t c;
16     } x;
17
18
19     x.a = 10;
20     x.b = 20;
21     x.c = 30;
22
23
24     return 0;
25 }
26
```

Структура – это тип данных, который является совокупностью нескольких переменных

Доступ к полям структуры осуществляется через точку. С полями структуры можно работать как с обычными переменными (читать, писать, инкрементировать и т.п.

Структуры

```
5 #include <stdio.h>
6 #include <stdint.h>
7
8 int main()
9 {
10
11     struct
12     {
13         int32_t a;
14         int8_t b;
15         uint64_t c;
16
17         float d[500];
18
19         struct {
20             int x, y, z;
21         } e;
22
23     } x;
24
25     x.e.x = 10;
26     x.d[0] = 50;
27     printf("%f", x.d[0]);
28
29     return 0;
30 }
```

В структуру можно вкладывать любые типы данных, такие как массивы или даже другие структуры

Структуры

```
4
5 #include <stdio.h>
6 #include <stdint.h>
7 #include <math.h>
8
9 struct vector
10 {
11     float x, y, z;
12 };
13
14
15 float norm(struct vector value)
16 {
17     return sqrt(value.x*value.x + value.y*value.y + value.z*value.z);
18 }
19
20 int main()
21 {
22
23     struct vector v;
24     v.x = v.y = v.z = 0;
25
26     return 0;
27 }
28
```

Структуре можно
дать имя, чтобы
можно было
создавать
переменные такого
типа в нескольких
местах

Структуры

```
4
5 #include <stdio.h>
6 #include <stdint.h>
7 #include <math.h>
8
9 typedef struct vector
10 {
11     float x, y, z;
12 } vector_t;
13
14
15 float norm(vector_t value)
16 {
17     return sqrt(value.x*value.x + value.y*value.y + value.z*value.z);
18 }
19
20 int main()
21 {
22
23     vector_t v;
24     v.x = v.y = v.z = 0;
25     float v_norm = norm(v);
26
27     return 0;
28 }
29
```

Чтобы не писать
везде struct, можно
определить
структуру как
пользовательский тип

Структуры

```
7
8 typedef struct vector
9 {
10     float x, y, z;
11 } vector_t;
12
13
14 int main()
15 {
16     vector_t v = {0, 0, 0};
17
18     vector_t a = {
19         .x = 10,
20         .z = 30,
21     };
22
23     return 0;
24 }
25
```

Инициализируются
структуры, как
массивы

Имеется
специальная
инициализация
отдельных полей

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	
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

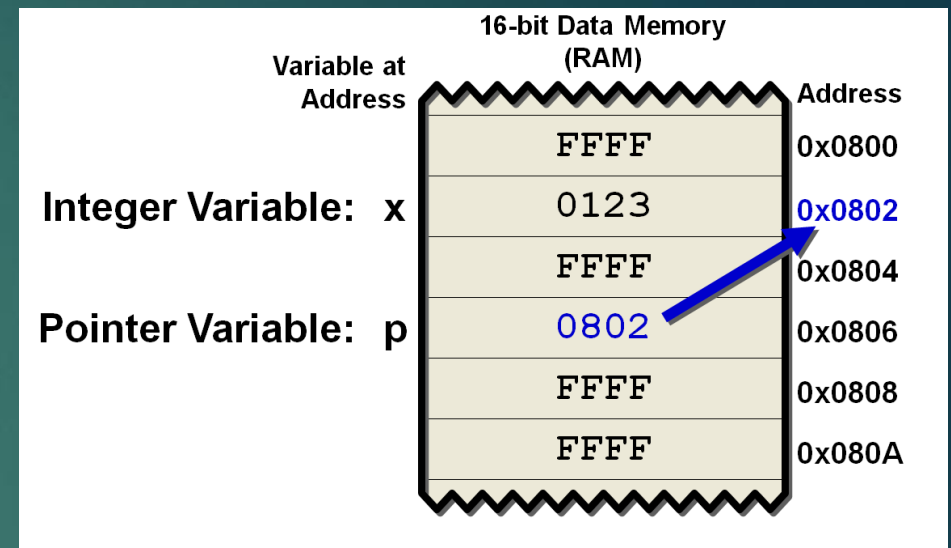
```
int main() {  
    int x;  
    int *p;  
  
    p = &x;  
    printf("p = %p\n", p);  
  
    x = 0;  
    printf("x до изменения = %d\n", x);  
  
    *p = 10;  
    printf("x после изменения = %d\n", x);  
  
    return 0;  
}
```

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

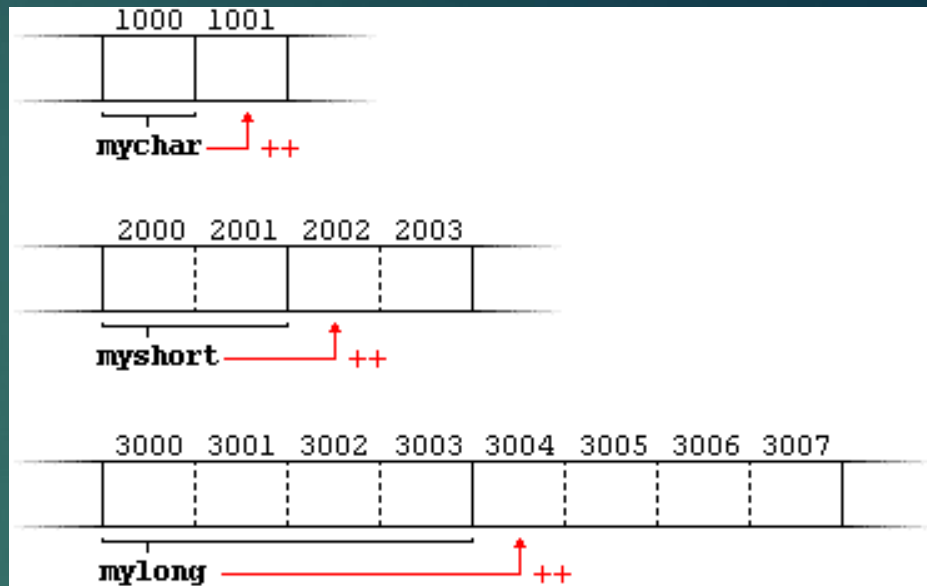
```
#include <stdint.h>
#include <stdbool.h>

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 }  
23
```

```
<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, например, удобно делать опциональные аргументы функций.

```
8 int sum(const int *x, const int *y)
9 {
10     if (y != NULL)
11         return *x + *y;
12     else
13         return *x;
14 }
15
16
17 int main()
18 {
19     int x = 10;
20     int y = 20;
21
22     int sum1 = sum(&x, &y);
23     int sum2 = sum(&x, NULL);
24
25     printf("sum1=%d sum2=%d\n", sum1, sum2);
26
27     return 0;
28 }
29
```

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