

Так. Опишу свои попытки разобраться в том, как компилятор GCC работает с массивами и указателями.

С чего все началось.

С простого задания «Описать трёхмерный целочисленный массив, размером 3x3x3 и вывести элемент с индексом [1][1][1] при помощи арифметики указателей.»

Я, недолго думая, сделал это так:

```
int cube[3][3][3];
int *ptr = &cube[0][0][0];
cube[1][1][1] = 777;
int i = 1, j = 1, k = 1;
std::cout << *(ptr + i*3*3 + j*3 + k) << "\n";
```

Идея заключается в присвоении указателю адреса первого элемента массива и отступить от него на соответствующее смещение.

Первый индекс шагает на 3\*3 элементов (то есть размером 2-мерного массива)

Второй шагает на 3 элемента.

Третий отступает по элементу.

Преподаватель сказал, что такая адресация не очень прозрачная. И пожалуй был прав. (зато такое представление отражает тот факт, что массив выделенный на стэке идет «сплошняком», то есть непрерывно, элемент за элементом. ИМХО).

Преподаватель предложил такой вариант:

```
printf("%d\n", (*(*(cube + 1) + 1) + 1));
```

Тут у меня в голове возник «Kernel Panic». Как так?! Ведь *cube* это адрес (в моем понимании адрес == указатель, позже станет понятно почему), прибавив к адресу значение (то есть увеличив адрес) мы получим адрес, и наконец разыменовав адрес мы должны получить значение! А у нас там еще целых 2 разыменования, мы же по-любому вылезем за пределы программы (в смысле за пределы диапазона адресов выделенных в оперативной памяти для нашей программы)!!! Так бы и случилось, если бы *cube* был самым обычным указателем, как например *int \*\*\*ptr*;. Однако *cube* не совсем обычный указатель. Да *cube* == *адрес\_начала\_массива*, и служит указателем начала массива, но компилятор GCC знает что это *cube* это массив и потому работает с ним несколько по-другому.

**Оговорюсь сразу, что что бы достоверно узнать КАК компилятор работает с массивом, нужно изучать сорсы компилятора или реверсить. Все мои мысли являются только догадкой.**

Итак. Первое, что я сделал, это накидал такой код:

```
#include <stdio.h>
```

```
int main(int argc, char* argv[]){
    int a = 10;
    int cube[3][3][3] = {
        {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
        {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}},
        {{19, 20, 21}, {22, 23, 24}, {25, 26, 27}}
    };
    int *p = &a;
    a = *p + 10;
    int *ptr = &cube[0][0][0];
    printf("%d\n", *(ptr + 1*3*3 + 1*3 + 1));
    printf("%d\n", (*(*(cube + 1) + 1) + 1));
    printf("%d\n", (*(*(cube + 2) + 1) + 1));
}
```

```
printf("%p\n", *(cube + 1));
return 0;
}
```

Теперь я думал, что запущу ее в отладчике и увижу как все эти разыменования могут выдавать корректный результат.

Но не тут то было.

По порядку.

Первое, что мы увидим в отладчике (или можно использовать какой-либо дизассемблер), это инициализацию переменных. Все переменные у нас локальные, используются только в одной функции, потому место для них выделяется на стеке.

0000000000401550	55	push rbp
0000000000401551	48:89E5	mov rbp, rsp
0000000000401554	48:81EC A0000000	sub rsp, A0
0000000000401558	894D 10	mov dword ptr ss:[rbp+10], ecx
000000000040155E	48:8955 18	mov qword ptr ss:[rbp+18], rdx
0000000000401562	E8 09020000	call t.401770
0000000000401567	C745 EC 0A000000	mov dword ptr ss:[rbp-14], A
000000000040156E	C745 80 01000000	mov dword ptr ss:[rbp-80], 1
0000000000401575	C745 84 02000000	mov dword ptr ss:[rbp-7C], 2
000000000040157C	C745 88 03000000	mov dword ptr ss:[rbp-78], 3
0000000000401583	C745 8C 04000000	mov dword ptr ss:[rbp-74], 4
000000000040158A	C745 90 05000000	mov dword ptr ss:[rbp-70], 5
0000000000401591	C745 94 06000000	mov dword ptr ss:[rbp-6C], 6
0000000000401598	C745 98 07000000	mov dword ptr ss:[rbp-68], 7
000000000040159F	C745 9C 08000000	mov dword ptr ss:[rbp-64], 8
00000000004015A6	C745 A0 09000000	mov dword ptr ss:[rbp-60], 9
00000000004015AD	C745 A4 0A000000	mov dword ptr ss:[rbp-5C], A
00000000004015B4	C745 A8 0B000000	mov dword ptr ss:[rbp-58], B
00000000004015BB	C745 AC 0C000000	mov dword ptr ss:[rbp-54], C
00000000004015C2	C745 B0 0D000000	mov dword ptr ss:[rbp-50], D
00000000004015C9	C745 B4 0E000000	mov dword ptr ss:[rbp-4C], E
00000000004015D0	C745 B8 0F000000	mov dword ptr ss:[rbp-48], F
00000000004015D7	C745 BC 10000000	mov dword ptr ss:[rbp-44], 10
00000000004015DE	C745 C0 11000000	mov dword ptr ss:[rbp-40], 11
00000000004015E5	C745 C4 12000000	mov dword ptr ss:[rbp-3C], 12
00000000004015EC	C745 C8 13000000	mov dword ptr ss:[rbp-38], 13
00000000004015F3	C745 CC 14000000	mov dword ptr ss:[rbp-34], 14
00000000004015FA	C745 D0 15000000	mov dword ptr ss:[rbp-30], 15
0000000000401601	C745 D4 16000000	mov dword ptr ss:[rbp-2C], 16
0000000000401608	C745 D8 17000000	mov dword ptr ss:[rbp-28], 17
000000000040160F	C745 DC 18000000	mov dword ptr ss:[rbp-24], 18
0000000000401616	C745 E0 19000000	mov dword ptr ss:[rbp-20], 19
000000000040161D	C745 E4 1A000000	mov dword ptr ss:[rbp-1C], 1A
0000000000401624	C745 E8 1B000000	mov dword ptr ss:[rbp-18], 1B

Немного инфы. Регистр rsp (полагаю sp – stack pointer) всегда указывает на вершину стека. Стек работает по принципу LIFO (last in, first out), то есть последнее значение которое в него положили, будет извлечено первым. Стек растет в сторону меньших адресов (от 0xFFFFFFFF до 0x00000000 (для x32 систем, для x64 расширьте значения до 8 байт)). Инструкция push помещает значение на вершину стека (и соответственно *уменьшает* значение rsp), инструкция pop снимает значение с вершины стека, *увеличивая* rsp. Взглянем на ассемблерный код.

```
push rbp
mov rbp, rsp
sub rsp, A0
```

Это вполне стандартный пролог, который можно встретить в начале большинства функций. Регистр rbp (полагаю bp – base pointer) в данном случае используется, как указатель кадра стека. Деление на кадры условное, чтобы было удобней обращаться к локальным переменным и аргументам функции (но это не точно).

Командой push rbp мы сохраняем значение регистра в на вершине стека. То есть сохраняем адрес предыдущего кадра стека, чтобы потом, после того как наша функция отработает вернуть стек в исходное состояние.

Командой mov rbp, rsp, помещаем указатель на вершину стека в регистр rbp, тем самым обозначая начало кадра стека нашей функции.

Командой sub rsp, A0, увеличиваем значение регистра rsp на 0xA0 (160), тем самым резервируем место для наших локальных переменных. На вскидку нам нужно 4\*27 байт для массива, 4 байта для переменной a, 8 байт для указателя, итого 120 байт. Компилятор

выделил больше, но наверняка у него были причины, больше не меньше (полагаю еще может на выравнивание потребоваться).

Сразу после инструкции call идет длинная череда присваиваний. Это как раз инициализация переменных. К примеру `mov [rbp-14], A` означает что по адресу `rbp-14` нужно поместить значение 10 (0xA). Это инициализация переменной `int a = 10`; То есть мы знаем адрес `a` и он равен `rbp-14`. Как можно было заметить [ ] это и есть операция разыменования.

Сразу за инициализацией переменной `a`, следует инициализация массива `cube`. Его элементы расположатся с адреса `rbp-80` до `rbp-14`. После инициализации стек будет выглядеть так:

00000000000061FDCC	00 00 00 00	01 00 00 00	02 00 00 00	03 00 00 00	.....
00000000000061FDDC	04 00 00 00	05 00 00 00	06 00 00 00	07 00 00 00	.....
00000000000061FDEC	08 00 00 00	09 00 00 00	0A 00 00 00	0B 00 00 00	.....
00000000000061FDFF	0C 00 00 00	0D 00 00 00	0E 00 00 00	0F 00 00 00	.....
00000000000061FE0C	10 00 00 00	11 00 00 00	12 00 00 00	13 00 00 00	.....
00000000000061FE1C	14 00 00 00	15 00 00 00	16 00 00 00	17 00 00 00	.....
00000000000061FE2C	18 00 00 00	19 00 00 00	1A 00 00 00	1B 00 00 00	.....
00000000000061FE3C	1C 00 00 00	1D 00 00 00	1E 00 00 00	1F 00 00 00	.....
00000000000061FE4C	20 00 00 00	21 00 00 00	22 00 00 00	23 00 00 00	.....
00000000000061FE5C	24 00 00 00	25 00 00 00	26 00 00 00	27 00 00 00	.....
00000000000061FE6C	28 00 00 00	29 00 00 00	2A 00 00 00	2B 00 00 00	.....
00000000000061FE7C	2C 00 00 00	2D 00 00 00	2E 00 00 00	2F 00 00 00	.....
00000000000061FE8C	30 00 00 00	31 00 00 00	32 00 00 00	33 00 00 00	.....
00000000000061FE9C	34 00 00 00	35 00 00 00	36 00 00 00	37 00 00 00	.....
00000000000061FEAC	38 00 00 00	39 00 00 00	3A 00 00 00	3B 00 00 00	.....
00000000000061FEBC	3C 00 00 00	3D 00 00 00	3E 00 00 00	3F 00 00 00	.....
00000000000061FEC	40 00 00 00	41 00 00 00	42 00 00 00	43 00 00 00	.....
00000000000061FED	44 00 00 00	45 00 00 00	46 00 00 00	47 00 00 00	.....
00000000000061FEE	48 00 00 00	49 00 00 00	4A 00 00 00	4B 00 00 00	.....
00000000000061FEF	4C 00 00 00	4D 00 00 00	4E 00 00 00	4F 00 00 00	.....
00000000000061FF0	50 00 00 00	51 00 00 00	52 00 00 00	53 00 00 00	.....
00000000000061FF1	54 00 00 00	55 00 00 00	56 00 00 00	57 00 00 00	.....
00000000000061FF2	58 00 00 00	59 00 00 00	5A 00 00 00	5B 00 00 00	.....
00000000000061FF3	5C 00 00 00	5D 00 00 00	5E 00 00 00	5F 00 00 00	.....
00000000000061FF4	60 00 00 00	61 00 00 00	62 00 00 00	63 00 00 00	.....
00000000000061FF5	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00	.....
00000000000061FF6	68 00 00 00	69 00 00 00	6A 00 00 00	6B 00 00 00	.....
00000000000061FF7	6C 00 00 00	6D 00 00 00	6E 00 00 00	6F 00 00 00	.....
00000000000061FF8	70 00 00 00	71 00 00 00	72 00 00 00	73 00 00 00	.....
00000000000061FF9	74 00 00 00	75 00 00 00	76 00 00 00	77 00 00 00	.....
00000000000061FFA	78 00 00 00	79 00 00 00	7A 00 00 00	7B 00 00 00	.....
00000000000061FFB	7C 00 00 00	7D 00 00 00	7E 00 00 00	7F 00 00 00	.....
00000000000061FFC	80 00 00 00	81 00 00 00	82 00 00 00	83 00 00 00	.....
00000000000061FFD	84 00 00 00	85 00 00 00	86 00 00 00	87 00 00 00	.....
00000000000061FFE	88 00 00 00	89 00 00 00	8A 00 00 00	8B 00 00 00	.....
00000000000061FFF	8C 00 00 00	8D 00 00 00	8E 00 00 00	8F 00 00 00	.....
0000000000006200	90 00 00 00	91 00 00 00	92 00 00 00	93 00 00 00	.....
0000000000006201	94 00 00 00	95 00 00 00	96 00 00 00	97 00 00 00	.....
0000000000006202	98 00 00 00	99 00 00 00	9A 00 00 00	9B 00 00 00	.....
0000000000006203	9C 00 00 00	9D 00 00 00	9E 00 00 00	9F 00 00 00	.....
0000000000006204	A0 00 00 00	A1 00 00 00	A2 00 00 00	A3 00 00 00	.....
0000000000006205	A4 00 00 00	A5 00 00 00	A6 00 00 00	A7 00 00 00	.....
0000000000006206	A8 00 00 00	A9 00 00 00	AA 00 00 00	AB 00 00 00	.....
0000000000006207	AC 00 00 00	AD 00 00 00	AE 00 00 00	AF 00 00 00	.....
0000000000006208	B0 00 00 00	B1 00 00 00	B2 00 00 00	B3 00 00 00	.....
0000000000006209	B4 00 00 00	B5 00 00 00	B6 00 00 00	B7 00 00 00	.....
000000000000620A	B8 00 00 00	B9 00 00 00	BA 00 00 00	BB 00 00 00	.....
000000000000620B	BC 00 00 00	BD 00 00 00	BE 00 00 00	BF 00 00 00	.....
000000000000620C	C0 00 00 00	C1 00 00 00	C2 00 00 00	C3 00 00 00	.....
000000000000620D	C4 00 00 00	C5 00 00 00	C6 00 00 00	C7 00 00 00	.....
000000000000620E	C8 00 00 00	C9 00 00 00	CA 00 00 00	CB 00 00 00	.....
000000000000620F	CC 00 00 00	CD 00 00 00	CE 00 00 00	CF 00 00 00	.....
0000000000006210	00 00 00 00	01 00 00 00	02 00 00 00	03 00 00 00	.....
0000000000006211	04 00 00 00	05 00 00 00	06 00 00 00	07 00 00 00	.....
0000000000006212	08 00 00 00	09 00 00 00	0A 00 00 00	0B 00 00 00	.....
0000000000006213	0C 00 00 00	0D 00 00 00	0E 00 00 00	0F 00 00 00	.....
0000000000006214	10 00 00 00	11 00 00 00	12 00 00 00	13 00 00 00	.....
0000000000006215	14 00 00 00	15 00 00 00	16 00 00 00	17 00 00 00	.....
0000000000006216	18 00 00 00	19 00 00 00	1A 00 00 00	1B 00 00 00	.....
0000000000006217	1C 00 00 00	1D 00 00 00	1E 00 00 00	1F 00 00 00	.....
0000000000006218	20 00 00 00	21 00 00 00	22 00 00 00	23 00 00 00	.....
0000000000006219	24 00 00 00	25 00 00 00	26 00 00 00	27 00 00 00	.....
000000000000621A	28 00 00 00	29 00 00 00	2A 00 00 00	2B 00 00 00	.....
000000000000621B	2C 00 00 00	2D 00 00 00	2E 00 00 00	2F 00 00 00	.....
000000000000621C	30 00 00 00	31 00 00 00	32 00 00 00	33 00 00 00	.....
000000000000621D	34 00 00 00	35 00 00 00	36 00 00 00	37 00 00 00	.....
000000000000621E	38 00 00 00	39 00 00 00	3A 00 00 00	3B 00 00 00	.....
000000000000621F	3C 00 00 00	3D 00 00 00	3E 00 00 00	3F 00 00 00	.....
0000000000006220	40 00 00 00	41 00 00 00	42 00 00 00	43 00 00 00	.....
0000000000006221	44 00 00 00	45 00 00 00	46 00 00 00	47 00 00 00	.....
0000000000006222	48 00 00 00	49 00 00 00	4A 00 00 00	4B 00 00 00	.....
0000000000006223	4C 00 00 00	4D 00 00 00	4E 00 00 00	4F 00 00 00	.....
0000000000006224	50 00 00 00	51 00 00 00	52 00 00 00	53 00 00 00	.....
0000000000006225	54 00 00 00	55 00 00 00	56 00 00 00	57 00 00 00	.....
0000000000006226	58 00 00 00	59 00 00 00	5A 00 00 00	5B 00 00 00	.....
0000000000006227	5C 00 00 00	5D 00 00 00	5E 00 00 00	5F 00 00 00	.....
0000000000006228	60 00 00 00	61 00 00 00	62 00 00 00	63 00 00 00	.....
0000000000006229	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00	.....
000000000000622A	68 00 00 00	69 00 00 00	6A 00 00 00	6B 00 00 00	.....
000000000000622B	6C 00 00 00	6D 00 00 00	6E 00 00 00	6F 00 00 00	.....
000000000000622C	70 00 00 00	71 00 00 00	72 00 00 00	73 00 00 00	.....
000000000000622D	74 00 00 00	75 00 00 00	76 00 00 00	77 00 00 00	.....
000000000000622E	78 00 00 00	79 00 00 00	7A 00 00 00	7B 00 00 00	.....
000000000000622F	7C 00 00 00	7D 00 00 00	7E 00 00 00	7F 00 00 00	.....
0000000000006230	80 00 00 00	81 00 00 00	82 00 00 00	83 00 00 00	.....
0000000000006231	84 00 00 00	85 00 00 00	86 00 00 00	87 00 00 00	.....
0000000000006232	88 00 00 00	89 00 00 00	8A 00 00 00	8B 00 00 00	.....
0000000000006233	8C 00 00 00	8D 00 00 00	8E 00 00 00	8F 00 00 00	.....
0000000000006234	90 00 00 00	91 00 00 00	92 00 00 00	93 00 00 00	.....
0000000000006235	94 00 00 00	95 00 00 00	96 00 00 00	97 00 00 00	.....
0000000000006236	98 00 00 00	99 00 00 00	9A 00 00 00	9B 00 00 00	.....
0000000000006237	9C 00 00 00	9D 00 00 00	9E 00 00 00	9F 00 00 00	.....
0000000000006238	A0 00 00 00	A1 00 00 00	A2 00 00 00	A3 00 00 00	.....
0000000000006239	A4 00 00 00	A5 00 00 00	A6 00 00 00	A7 00 00 00	.....
000000000000623A	A8 00 00 00	A9 00 00 00	AA 00 00 00	AB 00 00 00	.....
000000000000623B	AC 00 00 00	AD 00 00 00	AE 00 00 00	AF 00 00 00	.....
000000000000623C	B0 00 00 00	B1 00 00 00	B2 00 00 00	B3 00 00 00	.....
000000000000623D	B4 00 00 00	B5 00 00 00	B6 00 00 00	B7 00 00 00	.....
000000000000623E	B8 00 00 00	B9 00 00 00	BA 00 00 00	BB 00 00 00	.....
000000000000623F	BC 00 00 00	BD 00 00 00	BE 00 00 00	BF 00 00 00	.....
0000000000006240	C0 00 00 00	C1 00 00 00	C2 00 00 00	C3 00 00 00	.....
0000000000006241	C4 00 00 00	C5 00 00 00	C6 00 00 00	C7 00 00 00	.....
0000000000006242	C8 00 00 00	C9 00 00 00	CA 00 00 00	CB 00 00 00	.....
0000000000006243	CC 00 00 00	CD 00 00 00	CE 00 00 00	CF 00 00 00	.....
0000000000006244	00 00 00 00	01 00 00 00	02 00 00 00	03 00 00 00	.....
0000000000006245	04 00 00 00	05 00 00 00	06 00 00 00	07 00 00 00	.....
0000000000006246	08 00 00 00	09 00 00 00	0A 00 00 00	0B 00 00 00	.....
0000000000006247	0C 00 00 00	0D 00 00 00	0E 00 00 00	0F 00 00 00	.....
0000000000006248	10 00 00 00	11 00 00 00	12 00 00 00	13 00 00 00	.....
0000000000006249	14 00 00 00	15 00 00 00	16 00 00 00	17 00 00 00	.....
000000000000624A	18 00 00 00	19 00 00 00	1A 00 00 00	1B 00 00 00	.....
000000000000624B	1C 00 00 00	1D 00 00 00	1E 00 00 00	1F 00 00 00	.....
000000000000624C	20 00 00 00	21 00 00 00	22 00 00 00	23 00 00 00	.....
000000000000624D	24 00 00 00	25 00 00 00	26 00 00 00	27 00 00 00	.....
000000000000624E	28 00 00 00	29 00 00 00	2A 00 00 00	2B 00 00 00	.....
000000000000624F	2C 00 00 00	2D 00 00 00	2E 00 00 00	2F 00 00 00	.....
0000000000006250	30 00 00 00	31 00 00 00	32 00 00 00	33 00 00 00	.....
0000000000006251	34 00 00 00	35 00 00 00	36 00 00 00	37 00 00 00	.....
0000000000006252	38 00 00 00	39 00 00 00	3A 00 00 00	3B 00 00 00	.....
0000000000006253	3C 00 00 00	3D 00 00 00	3E 00 00 00	3F 00 00 00	.....
0000000000006254	40 00 00 00	41 00 00 00	42 00 00 00	43 00 00 00	.....
0000000000006255	44 00 00 00	45 00 00 00	46 00 00 00	47 00 00 00	.....
0000000000006256	48 00 00 00	49 00 00 00	4A 00 00 00	4B 00 00 00	.....
0000000000006257	4C 00 00 00	4D 00 00 00	4E 00 00 00	4F 00 00 00	.....
0000000000006258	50 00 00 00	51 00 00 00	52 00 00 00	53 00 00 00	.....
0000000000006259	54 00 00 00	55 00 00 00	56 00 00 00	57 00 00 00	.....
00000000000062					

Адрес	Шестнадцатеричное	ASCII
0000000000061FDBC	00 00 00 00	08 00 00 00
0000000000061FDCC	00 00 00 00	01 00 00 00
0000000000061FDDC	04 00 00 00	05 00 00 00
0000000000061FDEC	08 00 00 00	0A 00 00 00
0000000000061FDFC	0C 00 00 00	0E 00 00 00
0000000000061FE0C	10 00 00 00	11 00 00 00
0000000000061FE1C	14 00 00 00	15 00 00 00
0000000000061FE2C	18 00 00 00	19 00 00 00
0000000000061FE3C	1A 00 00 00	1A 00 00 00
	EA 14 B1 00	00 00 00 00

Указатель занимает 8 байт. В нем лежит адрес переменной **a**. Все как и ожидалось (за исключением того, что я ожидал увидеть 3CFE6100 00000000, видимо какие-то нюансы работы процессора со стеком). Почему порядок байт изменен? Процессор работает с памятью в порядке little endian(от младшего байта к старшему).

Далее

```
mov rax, [rbp-8]    //получаем адрес указателя
mov eax, dword [rax] //то самое разыменованное *p
                    //dword указывает что хотим прочитать 4 байта
```

Вот именно так в моей голове происходит разыменованное *любого* указателя.

Если даже это будет тройной указатель, например `int ***p`; операция разыменованного `***p` в инструкциях процессора будет выглядеть примерно так:

```
mov rax, [ptr_address]
mov rax, [rax]
mov rax, [rax]
mov eax, dword [rax]
```

(Может в конце добавлю код с тройным указателем и проверю)

Далее

```
add eax, 0xA        // *p+10
mov [rbp-14], eax    // a = *p + 10
lea rax, [rbp-80]    //получаем адрес массива
mov [rbp-10], rax    // *ptr = &cube[0][0][0];
mov rax, [rbp-10]    //и все таки нам нужен адрес массива
add rax, 34          //добавляем смещение до нужного элемента
mov eax, dword [rax] //получаем значение элемента массива, разыменовывая его адрес
                    //dword указывает что хотим прочитать 4 байта
```

Вот так. Все вот это выражение  $*(ptr + 1*3*3 + 1*3 + 1)$  преобразовалось в последние 3 строки ассемблерного кода. GCC еще на этапе компиляции рассчитал смещение и оптимизировал код.

Неужели и с этим выражением  $*( (* (cube + 1) + 1) + 1)$  он поступит также? .

Следующий фрагмент кода

```
mov edx, eax
lea rcx, ds:[404000]
call printf
```

Здесь у нас передача параметров и вызов функции `printf`. Согласно соглашению о вызовах функций для ОС Windows x64 параметры передаются через регистры `rcx`, `rdx`, `r8`, `r9` остальные параметры (если их  $> 4$ ) через стек, если я все правильно помню. В нашем случае это выглядит как то так: `printf(rcx, rdx)`. В `rdx` у нас адрес элемента массива, в `rcx` указатель на строку `«%d\n»`. (Кстати в соглашении о вызове в Linux x64 параметры передаются через регистры `rdi`, `rsi`, `rdx`, `rcx`. Опять же могу ошибаться)

Далее

```
lea rax, [rbp-80]    //адрес массива
add rax, 34          //смещение
mov eax, dword [rax] //разыменованное
```

Вот и выражение  $*( (* (cube + 1) + 1) + 1)$  свелось к получению адреса массива, сдвигу до нужного элемента и разыменованию.



Вызов функции ничем не отличается

Далее.  $*(*(cube + 2) + 1)$

`lea rax, [rbp-80]` //адрес массива

`add rax, 34` //смещение

`mov eax, dword [rax]` //разыменование

вызов функции

Наконец добрались до  $*(cube + 1)$

`lea rax, [rbp-80]` //адрес массива

`add rax, 24` //смещение

вызов функции

И все? Даже разыменования нет?! Но ведь оно же явно вызывается!

Очевидно, что пользование отладчиком, не прольет свет на то, как же компилятор обрабатывает арифметику указателей касательно массивов.

`mov eax, 0` //return 0

`add rsp, 40` //возвращаем стек в исходное состояние

`pop rbp` //возвращаем указатель на кадр стека вызывающе функции

`ret` //переходим по адресу возврата

Так хотел добавить тройной указатель.

Сделал.

`#include <stdio.h>`

```
int main(int argc, char* argv[]){
    int a = 10;
    int cube[3][3][3] = {
        {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
        {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}},
        {{19, 20, 21}, {22, 23, 24}, {25, 26, 27}}
    };
    int *p = &a;
    a = *p + 10;
    int *ptr = &cube[0][0][0];
    int **m = &ptr;
    int ***q = &m;
    printf("%d\n", *(ptr + 1*3*3 + 1*3 + 1));
    printf("%d\n", (*(cube + 1) + 1) + 1);
    printf("%d\n", (*(cube + 2) + 1));
    printf("%p\n", *(cube + 1));
    printf("%d\n", ***q);
    return 0;
}
```

00000000004016AE	48: 8D45 80	<code>lea rax, qword ptr ss: [rbp-80]</code>
00000000004016B2	48: 83C0 24	<code>add rax, 24</code>
00000000004016B6	48: 89C2	<code>mov rdx, rax</code>
00000000004016B9	48: 8D0D 44290000	<code>lea rcx, qword ptr ds: [404004]</code>
00000000004016C0	E8 0B150000	<code>call &lt;JMP.&amp;printf&gt;</code>
00000000004016C5	48: 8B45 F0	<code>mov rax, qword ptr ss: [rbp-10]</code>
00000000004016C9	48: 8B00	<code>mov rax, qword ptr ds: [rax]</code>
00000000004016CC	48: 8B00	<code>mov rax, qword ptr ds: [rax]</code>
00000000004016CF	8B00	<code>mov eax, dword ptr ds: [rax]</code>
00000000004016D1	89C2	<code>mov edx, eax</code>
00000000004016D3	48: 8D0D 2A290000	<code>lea rcx, qword ptr ds: [404004]</code>
00000000004016DA	E8 F1140000	<code>call &lt;JMP.&amp;printf&gt;</code>

Красивое, логичное тройное разыменование, а главное что без сюрпризов.

Ладно возвращаюсь к массивам.

**И внимание! Речь пойдет только о массивах статических, выделенных на стэке. То есть о простых массивах типа `int arr[n][m][k]`**

Попробую написать программку, в которой буду перебирать интересующие меня варианты работы с массивами через арифметику указателей.

Идея заключается в переборе вариантов с выводом типов и значений выражений.

Написал такую простыню:

```
#include <stdio.h>
#include <typeinfo>
#include <iostream>

int main(int argc, char* argv[]){
    int line[3] = {1,2,3};
    int square[2][2] = {
        {1, 2},
        {3, 4}
    };
    int cube[3][3][3] = {
        {{1, 2, 3}, {4, 5, 6},{7, 8, 9}},
        {{10, 11, 12}, {13, 14, 15},{16, 17, 18}},
        {{19, 20, 21}, {22, 23, 24},{25, 26, 27}}
    };
    std::cout << "int line[3];\nint square[2][2];\nint cube[3][3][3];" << '\n';
    std::cout << "1: line has type: " << typeid(line).name() << '\n';
    std::cout << "   square has type: " << typeid(square).name() << '\n';
    std::cout << "   cube has type: " << typeid(cube).name() << '\n';
    std::cout << "2: &line has type: " << typeid(&line).name() << '\n';
    std::cout << "   &square has type: " << typeid(&square).name() << '\n';
    std::cout << "   &cube has type: " << typeid(&cube).name() << '\n';
    std::cout << "3: &line[0] has type: " << typeid(&line[0]).name() << '\n';
    std::cout << "   &square[0][0] has type: " << typeid(&square[0][0]).name() << '\n';
    std::cout << "   &cube[0][0][0] has type: " << typeid(&cube[0][0][0]).name() << '\n';
    std::cout << "3: *line has type: " << typeid(*line).name() << '\n';
    std::cout << "   *square has type: " << typeid(*square).name() << '\n';
    std::cout << "   *cube has type: " << typeid(*cube).name() << '\n';
    std::cout << '\n';
    printf("line == %p  &line == %p  &line[0] == %p\n", line, &line, &line[0]);
    printf("square == %p  &square == %p  &square[0][0] == %p\n", square, &square,
&square[0][0]);
    printf("cube == %p  &cube == %p  &cube[0][0][0] == %p\n", cube, &cube, &cube[0]
[0][0]);
    std::cout << "cube has type: " << typeid(cube).name() << '\n';
    std::cout << "(cube+1) has type: " << typeid((cube+1)).name() << '\n';
    std::cout << '\n';
    return 0;
}
```

```

int line[3];
int square[2][2];
int cube[3][3][3];
1: line has type: A3_i
   square has type: A2_A2_i
   cube has type: A3_A3_A3_i
2: &line has type: PA3_i
   &square has type: PA2_A2_i
   &cube has type: PA3_A3_A3_i
3: &line[0] has type: Pi
   &square[0][0] has type: Pi
   &cube[0][0][0] has type: Pi
3: *line has type: i
   *square has type: A2_i
   *cube has type: A3_A3_i

line == 000000000061FE34   &line == 000000000061FE34   &line[0] == 000000000061FE34
square == 000000000061FE20   &square == 000000000061FE20   &square[0][0] == 000000000061FE20
cube == 000000000061FDB0   &cube == 000000000061FDB0   &cube[0][0][0] == 000000000061FDB0
cube has type: A3_A3_A3_i
(cube+1) has type: PA3_A3_i

```

Вывод программы:

Тааак. Интересно.

Буду рассуждать про *cube*, с остальными все обстоит так же.

Что же именно для меня интересно.

1. *cube* и *&cube* равны по значениям! Но ведь *cube* имеет свой адрес (как мы уже видели в отладчике)! Пусть будет 0x1000, тогда в памяти это бы выглядело так:

адрес : значение

0x1000 : 0x61FDB0

Но тогда *cube* должен быть равен 0x61FDB0, а *&cube* равен 0x1000.

А их значения равны. Но не типы. Операция *&* изменила тип, но значение не тронула.

**Вывод:** (это только мои рассуждения и все может быть полным бредом):

Операция *&* по отношению к имени массива компилятором рассматривается больше как приведение типа, чем взятие адреса.

С выражением *&cube[0][0][0]* все красиво. Первый элемент, берем адрес, получаем указатель.

2. Разные типы у *cube* и *(cube+1)*. Это что за магия? Берем значение/адрес/не\_важно\_что прибавляем 1 и получаем другой тип данных???

3. *\*cube* возвращает двумерный массив. Нет комментариев. Хотя если представить эту запись в другом виде *\*(cube + 0)*, то можно привести к пункту 2.

Из-за пунктов 2-3 я подвис. Да нет, не подвис. Впал в ступор. Хорошего объяснения я не нашел, для себя решил, что компилятор в тихую незаметно для нас выполняет приведение типов. В итоге он все оптимизирует и приводит к простому ассемблерному коду, как мы видели в отладчике. Но сами операции не очень прозрачны.

Полагаю все это сделано, чтобы для разработчика не было разницы при работе с динамическими или статическими массивами. Нужно только понимать, что при работе со статическим массивом, не все операции выполняются явным образом.

Жаль, что не хватило сил/желания/ума/времени разобраться досконально.

P.S. В порядке бреда. Где-то когда-то слышал (или статье читал), что компилятор при работе с массивом для многих операций приводит массив к указателю на первый элемент.

Если с этой точки зрения рассмотреть запись *(cube + 1)*, то

*(&cube[0][0][0] + 1)*, но 1 в этом случае не означает единицу, на сколько же нужно отступить?

Видимо компилятор предполагает, что ему нужно вернуть указатель на массив, в котором на 1 измерение меньше (т. е. из 2-мерного вернуть 1-мерный, из 3-мерного 2-мерный и т.д.).

Если принять, что так оно и есть, то итоговая запись для *(cube + 1)*

*(int(\*)[3][3])((char\*)&cube[0][0][0] + 1 \* sizeof(int[3][3]))* и все вроде стало понятнее))).