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));
```

```
    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 стэке.

```
0000000000401550    55                  push rbp
0000000000401551    48:89E5             mov rbp,rsp
0000000000401554    48:81EC A0000000    sub rsp,A0
000000000040155B    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 стэка. Стэк работает по принципу LIFO (last in, first out), то есть последнее значение которое в него положили, будет извлечено первым. Стэк 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 стэка (и соответственно *уменьшает* значение rsp), инструкция pop снимает значение с вершины стэка, *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 стэка. 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 стэка. То есть сохраняем адрес предыдущего кадра стэка, чтобы потом, после того как наша функция отработает вернуть стэк в исходное состояние.
Командой mov rbp, rsp, помещаем указатель на вершину стэка в регистр rbp, тем самым обозная начало кадра стэка 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

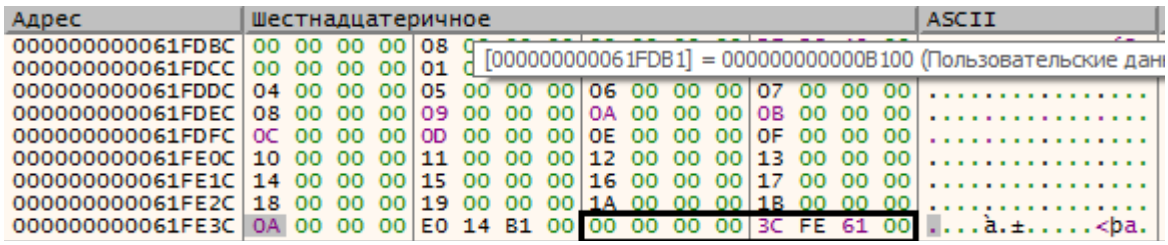
выделил больше, но наверняка у него были причины, больше не меньше (полагаю еще может на выравнивание потребоваться).
Сразу после инструкции call идет длинная череда присваиваний. Это как раз инициализация переменных. К примеру mov [rbp-14], A означает что по адресу rbp-14 нужно поместить значение 10 (0xA). Это инициализация переменой 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 ицициализация массива **cube**. Его елементы расположатся с адреса rbp-80 до rbp-14. После инициализации стэк будет выглядеть так:

```
0000000000061FDCC | 00 00 00 00 | 01 00 00 00 | 02 00 00 00 | 03 00 00 00 | ................
0000000000061FDDC | 04 00 00 00 | 05 00 00 00 | 06 00 00 00 | 07 00 00 00 | ................
0000000000061FDEC | 08 00 00 00 | 09 00 00 00 | 0A 00 00 00 | 0B 00 00 00 | ................
0000000000061FDFC | 0C 00 00 00 | 0D 00 00 00 | 0E 00 00 00 | 0F 00 00 00 | ................
0000000000061FE0C | 10 00 00 00 | 11 00 00 00 | 12 00 00 00 | 13 00 00 00 | ................
0000000000061FE1C | 14 00 00 00 | 15 00 00 00 | 16 00 00 00 | 17 00 00 00 | ................
0000000000061FE2C | 18 00 00 00 | 19 00 00 00 | 1A 00 00 00 | 1B 00 00 00 | ................
0000000000061FE3C | 0A 00 00 00 | E0 14 B1 00 | 00 00 00 00 | 10 00 00 00 | ....à.±.........
```

Переменная a имеет адрес 0x61FE3C. По идее мы потом должны будем увидеть этот адрес в ячейке памяти отведенной под указатель p (ведь int * = &a;).
Далее.

```
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
000000000040162B    48:8D45 EC          lea rax,qword ptr ss:[rbp-14]
000000000040162F    48:8945 F8          mov qword ptr ss:[rbp-8],rax
0000000000401633    48:8B45 F8          mov rax,qword ptr ss:[rbp-8]
0000000000401637    8B00                mov eax,dword ptr ds:[rax]
0000000000401639    83C0 0A             add eax,A
000000000040163C    8945 EC             mov dword ptr ss:[rbp-14],eax
000000000040163F    48:8D45 80          lea rax,qword ptr ss:[rbp-80]
0000000000401643    48:8945 F0          mov qword ptr ss:[rbp-10],rax
0000000000401647    48:8B45 F0          mov rax,qword ptr ss:[rbp-10]
000000000040164B    48:83C0 34          add rax,34
000000000040164F    8B00                mov eax,dword ptr ds:[rax]
0000000000401651    89C2                mov edx,eax
0000000000401653    48:8D0D A6290000    lea rcx,qword ptr ds:[404000]
000000000040165A    E8 41150000         call <JMP.&printf>
000000000040165F    48:8D45 80          lea rax,qword ptr ss:[rbp-80]
0000000000401663    48:83C0 34          add rax,34
0000000000401667    8B00                mov eax,dword ptr ds:[rax]
0000000000401669    89C2                mov edx,eax
000000000040166B    48:8D0D 8E290000    lea rcx,qword ptr ds:[404000]
0000000000401672    E8 29150000         call <JMP.&printf>
0000000000401677    48:8D45 80          lea rax,qword ptr ss:[rbp-80]
000000000040167B    48:83C0 4C          add rax,4C
000000000040167F    8B00                mov eax,dword ptr ds:[rax]
0000000000401681    89C2                mov edx,eax
0000000000401683    48:8D0D 76290000    lea rcx,qword ptr ds:[404000]
000000000040168A    E8 11150000         call <JMP.&printf>
000000000040168F    48:8D45 80          lea rax,qword ptr ss:[rbp-80]
0000000000401693    48:83C0 24          add rax,24
0000000000401697    48:89C2             mov rdx,rax
000000000040169A    48:8D0D 63290000    lea rcx,qword ptr ds:[404004]
00000000004016A1    E8 FA140000         call <JMP.&printf>
00000000004016A6    B8 00000000         mov eax,0
00000000004016AB    48:81C4 A0000000    add rsp,A0
00000000004016B2    5D                  pop rbp
00000000004016B3    C3                  ret
```

Сразу за инициализацией массива видим следующие инструкции:
lea rax, [rbp-14]
mov [rbp-8], rax
Инструкция lea (load effective address по-моему) помещает вычисленный *адрес* в регистр rax (в отличие от инструкции mov которая помещает *значение*). А адрес rbp-14 принадлежит переменной **a**. То есть после инструкции lea rax, [rbp-14] в регистре rax будет лежать адрес переменной **a**. Следующая инструкция поместит содержимое регистра rax (то есть адрес переменной) в память по адресу rbp-8. Этот адрес и принадлежит указателю **p**.
Взглянем на стэк после этих инструкций.



Указатель занимает 8 байт. В нем лежит адрес переменной **a**. Все как и ожидалось (за исключением того, что я ожидал увидеть 3CFE6100 00000000, видимо каки-то нюансы работы процессора со стэком). 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 разыменование **p***
//dword указывает что хотим прочитать 4 байта
Вот именно так в моей голове происходит разыменование *любого* указателя.
Если даже это будет тройной указатель, например int ***p; операция разыменования ***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 отпимизировал 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 стэк, 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] //разыменование
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, A0 //возвращаем стэк в исходное состояние
pop rbp //возвращаем указатель на кадр стэка 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;
}
```

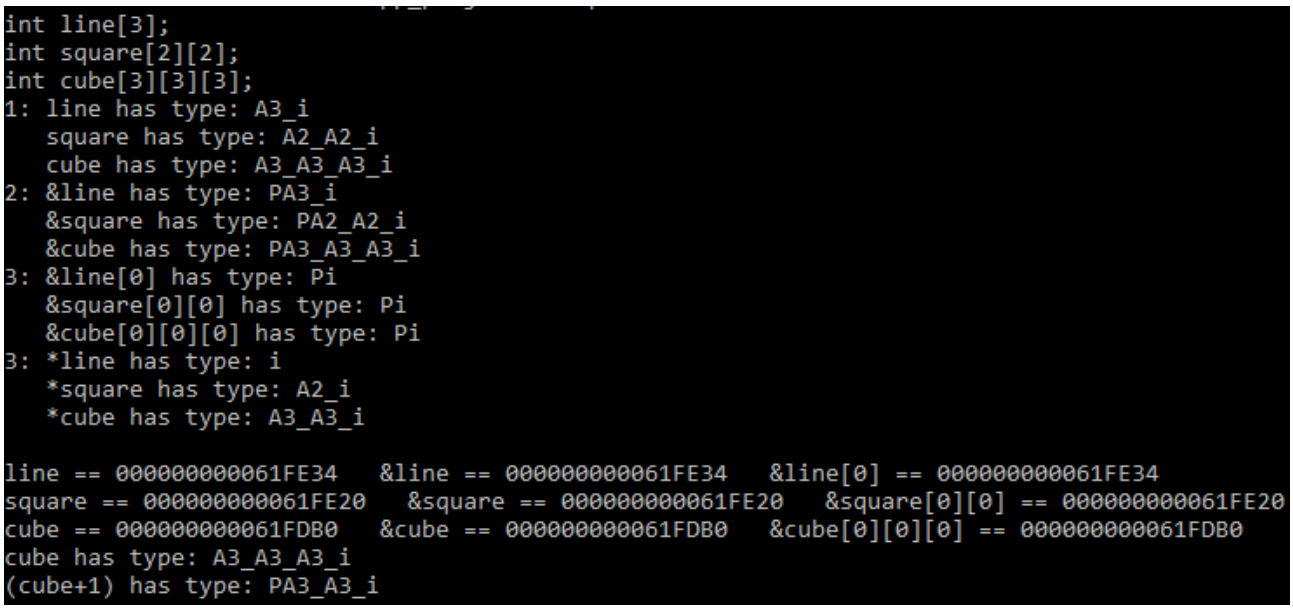
```
00000000004016AE    48:8D45 80          lea rax,qword ptr ss:[rbp-80]
00000000004016B2    48:83C0 24          add rax,24
00000000004016B6    48:89C2             mov rdx,rax
00000000004016B9    48:8D0D 44290000    lea rcx,qword ptr ds:[404004]
00000000004016C0    E8 0B150000         call <JMP.&printf>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

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;
}
```



Вывод программы:

Таааак. 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))).

Так я все-таки нашел время и силы разбираться дальше.
То что я считал в порядке бреда, оказалось не так далеко от истины. Нужно было всего-то почитать документацию к языку. Оказывается там есть целый раздел посвященный неявным преобразованиям. https://en.cppreference.com/w/cpp/language/implicit_conversion
И в нем рассказывается о том, что меня мучило.
Итак. Что же там написано касательно неявных преобразований массива?
А написано там примерно следующее.
Существует неявное преобразование массива к указателю на его первый элемент. Это преобразование выполняется тогда, когда массив появляется в контексте, где ожидается не массив, а указатель.

Да то, что я слышал, оказалось правдой. Я писал чуть выше «Где-то когда-то слышал (или статье читал), что компилятор при работе с массивом для многих операций приводит массив к указателю на первый элемент». Но теперь понятно для каких именно операций, а также понятно, что я делал не так в рассуждениях.
В операции *(cube + 1)* ожидается указатель. Значит компилятор произведет неявное преобразование к указателю на первый элемент. Я считал первым элементом — *cube[0][0][0]*. Однако в документации написано, что в данном случае первым элементом многомерного массива является массив, в котором на 1 измерение меньше. То есть cube представляется как массив состоящий из 3 массивов [3][3]. И получается первым элементом будет как раз двумерный массив [3][3].
Еще раз. Компилятор встретив операцию *(cube + 1)* неявно преобразует *cube* к указателю на массив[3][3] (к указателю на первый элемент).
Выглядеть это будет так:
(cube + 1) преобразуется в *((int(*)[3][3])(&cube[0][0][0]) + 1)*
Я даже подставил эту запись в компилятор вместо *(cube + 1)* и все прекрасно работает.
Еще одно замечание касательно *((int(*)[3][3])(&cube[0][0][0]) + 1)* — единица в данном случает означает отступить на 1 массив [3][3], то есть численно равна *sizeof(int[3][3])* (36 байт).
И напоследок замечание по поводу вот этих моих слов «*cube* и *&cube* равны по значениям! Но ведь *cube* имеет свой адрес (как мы уже видели в отладчике)!»
На самом деле *cube* и *&cube* не равны по значениям, хотя и printf говорит нам об этом. Все дело в том, что когда мы передаем cube в функцию printf в качестве аргумента, то это как раз то место, где ожидается указатель, а не массив, а значит будет неявное преобразование! В итоге в функцию передастся *(int(*)[3][3])(&cube[0][0][0])* (то есть указатель на первый элемент). В случае же с *&cube*, операция & вполне применима и к массиву, а значит неявного преобразования не требуется. В функцию передастся (*int(*)[3][3][3])(&cube[0][0][0])*. И вот у этих аргументов значения равны. (В примере с отладчиком это было бы значение rbp – 80).