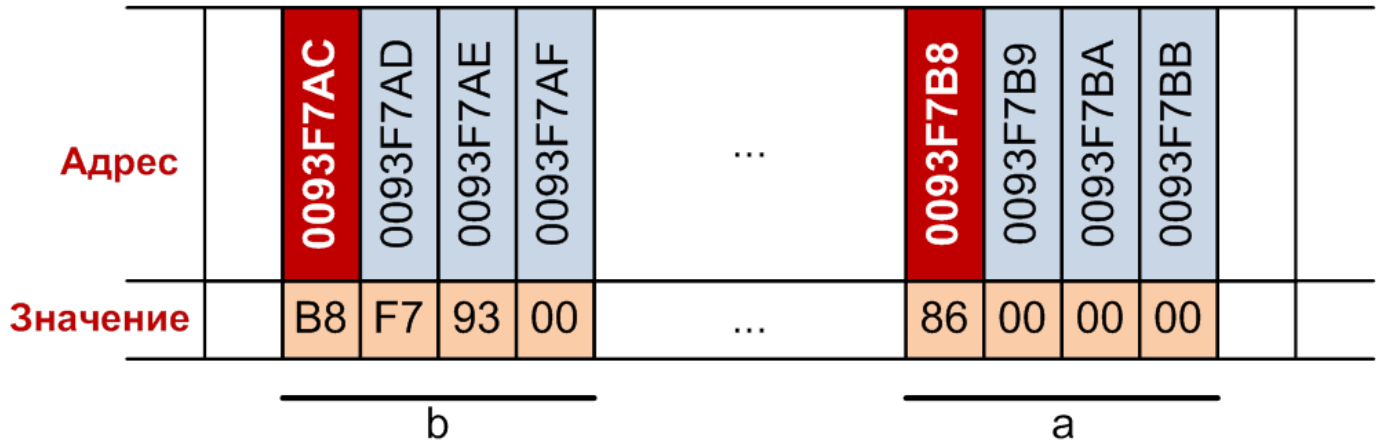
Создадим переменную типа uint8_t и назовем ее Counter. Пусть она будет равняться 142.

uint8_t Counter = 142;

Начнем с того, что же представляет из себя переменная в Си.



Переменная представляет из себя ячейку в памяти. Ячейка в памяти имеет адрес. Адрес этой ячейки представляет из себя число.

Т.е. Имя, которое мы дали переменной эквивалентно адресу ячейки в памяти.

Имен переменных на машинных кодах не существует.

И когда вы компилируете свой код, компилятор связывает имя переменной и ее адрес.

Т.е. за каждой переменной стоит какой-то адрес.

Как узнать адрес переменной:

Для этого в языке си есть такой оператор &(амперсанд).

Это унарный оператор, возвращающий адрес операнда в памяти.

Давайте рассмотрим пример:

```
uint8_t Counter = 142;
```

Есть у нас глобальная переменная Counter. Ее тип uint8_t.

Чтоб узнать адрес этой переменной, нужно выполнить операцию &Counter.

Итак. Давайте получим адрес переменной Counter.

```
printf("%p\r\n", &Counter);//p – вывод указателя
```

Консоль:

0x20000000

Вот мы получили число 0x20000000. Это число и есть адрес переменной Counter.

Раз адрес является числом, то его можно сохранить? Да.

Давайте создадим переменную A и сохраним в нее этот адрес.

Только какой тип будет у &Counter?

Т.к. переменная Counter имеет тип uint8_t, то тип адреса будет uint8_t*(только со звездочкой). Это будет называться адресным типом.

```
uint8_t* A = &Counter; //Переменная адресного типа
printf("%p\r\n", A);
```

Консоль:

0x20000000

Вроде бы все понятно, но дальше вы можете задуматься….Мы создали переменную адресного типа…А ведь у этой переменной тоже есть наверное свой адрес…

Да, так и есть…

Как нам его узнать?

Так же воспользоваться функцией взятия адреса переменной: &A;

А какой тогда будет тип? uint8_t**

```
uint8_t** B = &A;
printf("%p\r\n", B);
```

Давайте проверим.

Консоль:

0x20000000

0x20004ff0

Вот мы и узнали адрес переменной адресного типа A, который указывает на адрес переменной Counter.

Соответственно, у переменной адресного типа B, которая указывает на переменную адресного типа A, которая указывает на адрес переменной Counter тоже есть свой адрес.

```
uint8_t ***C = &B;
printf("%p\r\n", C);
```

Консоль:

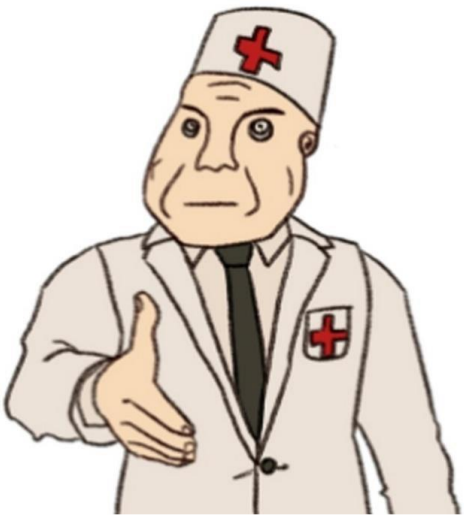
0x20000000

0x20004ff0

0x20004fec

Вот мы получили адрес переменной адресного типа B, которая указывает на переменную адресного типа A, которая указывает на адрес переменной.

Думаю дальше продолжать не стоит) Так можно идти до бесконечности и угодить в дурку.



Давайте теперь сделаем следующее. Вот у нас есть переменная Counter. Ее значение 142.

Как нам добраться до этого значения через переменные адресного типа A, B и C.

Делать мы это будем через операцию разыменования. Обозначается она *.

```
printf("Counter = %d\r\n", *A);
printf("Counter = %d\r\n", **B);
printf("Counter = %d\r\n", ***C);
```

Консоль:

0x20000000

0x20004ff0

0x20004fec

Counter = 142

Counter = 142

Counter = 142

Хорошо. Теперь рассмотрим массивы. Что из себя представляет массив данных?

Для ответа на этот вопрос создадим массив типа uint8_t, назовем его Array и выделим памяти на 10 элементов. Заполним его данными.

```
uint8_t Array[10] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09 };
```

Как нам вывести в консоль 0 элемент массива?

```
printf("Test:\r\n");
printf("%.2X\r\n", Array[0]);
```

Консоль:

Test:

00

А как через указатели?

```
printf("Test:\r\n");
printf("%.2X\r\n", *Array);
```

Консоль:

Test:

00

Либо с явным приведением типа:

```
printf("Test:\r\n");
printf("%.2X\r\n", *(uint8_t*)Array);
```

Консоль:

Test:

00

Тут “Array” является указателем на адрес 0 элемента массива.

Т.е. массивом в языке си является константный адрес и выделенная память.

Как добраться до следующих элементов массива?

Тут нам поможет адресная арифметика.

Доберемся до 1 элемента массива:

```
printf("Test:\r\n");
printf("%.2X\r\n", *((uint8_t*)(Array+1)));
```

Консоль:

Test:

01

Доберемся до 10 элемента массива:

```
printf("Test:\r\n");
printf("%.2X\r\n", *((uint8_t*)(Array+9)));
```

Консоль:

Test:

09

Двигаемся дальше.

Давайте попробуем собрать из двух элементов массива типа uint8_t одно значение переменной типа uint16_t.

Создадим переменную uint16_t A = 0x0000;

Представим переменную A, как побитовое сложение 2 элементов массива:

```
printf("Test:\r\n");
A = Array[1]<<8 | Array[2];
printf("A = %.4X\r\n", A);
```

Консоль:

Test:

A = 0102

А как нам сделать то же самое, но через указатели.

```
A = *((uint8_t*)(Array+1))<<8 | *((uint8_t*)(Array+2));
```

Консоль:

Test:

A = 0102

В данном случае мы использовали адресный тип uint8_t*.

А что будет, если мы попробуем брать адресный тип uint16_t*?

Давайте попробуем.

```
A = *(uint16_t*)Array;

printf("A = %.4X\r\n", A);
```

Консоль:

Test:

A = 0100

Что мы замечаем. Он берет 16 бит, это два рядом стоящих элемента массива, но берет он их задом наперед.

Давайте попробуем шагнуть вправо.

```
A = *((uint16_t*)(Array+1));
printf("A = %.4X\r\n", A);
```

Консоль:

Test:

A = 0201

Т.е. мы видим, что он шагает вправо на 8 бит, берет 16 бит данных, но берет их в обратной последовательности. И вот эта штука, часто путает, когда из какого-то массива хочется взять кусок памяти.

И что самое главное, скобки здесь играют очень большую роль!
Смотрите, стоит нам убрать вот эти скобки(которые я выделил зеленым), как результат изменится:

```
A = *((uint16_t*)Array+1);
printf("A = %.4X\r\n", A);
```

Консоль:

Test:

A = 0302

Теперь мы видим, что он шагает вправо уже на 16 бит и также берет 16 бит данных в обратной последовательности.

Но как нам можно исправить эту обратную последовательность?
Попробуем рассмотреть полученное 16 битное A, как две 8 битные переменные:

```
printf("A_1 = %.2X\r\n", *(uint8_t*) &A);
printf("A_2 = %.2X\r\n", *((uint8_t*) &A + 1));
```

Не забывайте про скобки, которые указывают, что мы будем разыменовывать, т.к. без них мы будем разыменовывать данные типа uint8_t по адресу A и прибавлять к этим данным единицу, а нам нужно адрес+1, поэтому скобки очень важны.

Консоль:

Test:

A = 0302

A_1 = 02

A_2 = 03

Соответственно, взять определенные данные из одного места в другое можно через тот же цикл for:
Напомню, мы хотим в 16 битное A положить 1 и 2 элемент массива по очереди, чтоб получилось 0x0102;

```
for (uint8_t i = 0; i < 2; i++) {
	*((uint8_t*) &A + i) = *((uint8_t*) Array + (2 - i));
}
printf("A = %.4X\r\n", A);
```

Консоль:

Test:

A = 0102

И вот у нас это получилось.

Как Вы видите, используя указатели, мы уже не так сильно привязаны к типам переменных. При помощи указателей можно представлять одни типы данных в другие.

К примеру, давайте представим float, как uint32_t.

Ранее, когда я еще не начал изучать тему указателей в си, я использовал union.

Мне всегда этот способ казался каким-то громоздким, но по-другому я не умел.

Продемонстрирую его и покажу, как можно сделать то же самое, но используя указатели.

Создадим переменные:

```
float pi = 3.14159f;
uint32_t pi_conversion = 0;

union {
	uint32_t result; //определения элементов
	float data;
} data_conversion; //имя объединения
```

Теперь, как я раньше выполнял преобразование:

```
data_conversion.data = pi;
pi_conversion = data_conversion.result;
printf("pi_conversion = %lX\r\n", pi_conversion);
```

Консоль:

Test:

pi_conversion = 40490FD0

Проверим. Да...Действительно. Преобразовалось верно.

Теперь сделаем то же самое, но используя всего одну строку и указатели:

```
pi_conversion = *(uint32_t*) &pi;
printf("pi_conversion = %lX\r\n", pi_conversion);
```

Консоль:

Test:

pi_conversion = 40490FD0

Вот мы получили то же самое) Прекрасно. А теперь обратно преобразуем данные типа uint32_t во float:

Создадим переменную типа float:

```
float test = 0.0f;
test = *(float*)&pi_conversion;
printf("pi = %f\r\n", test);
```

Консоль:

Test:

pi = 3.141590

Как Вы видите, любую информацию в памяти можно представить, как нужный нам тип данных. Главное знать адрес и конечно же учитывать последовательность байт данных.

Ну и чтоб закрепить наши умения, давайте разложим это же число float, как 4 переменные типа uint8_t;

У нас должны получиться переменные `40 49 0F D0;`

Переменная pi у нас есть

```
float pi = 3.14159f;
```

Создадим массив из 4 элементов:

```
uint8_t Array[4] = { 0, };
```

А далее, как обычно, используя цикл for:

```
for (uint8_t i = 0; i < 4; i++) {
	*((uint8_t*) Array + i) = *((uint8_t*) &pi + (3-i));
	printf("Data %d = %.2X\r\n", i, Array[i]);
	}
```

Консоль:

Test:

Data 0 = 40

Data 1 = 49

Data 2 = 0F

Data 3 = D0

Как Вы видите, используя указатели можно многие преобразования делать в разы быстрее.

Последний пример, который я бы хотел сегодня продемонстрировать, это как работать с указателями на переменные в функции.

Пример простой.

Создадим переменную

```
uint8_t X = 5;
```

Создадим простейшую функцию:

```
void function_test_1(uint8_t data) {
	printf("Data = %d\r\n", data);
	data = data + 5;
	printf("Data = %d\r\n", data);
}
```

Теперь, запишем такой код:

```
printf("Test:\r\n");
function_test_1(X);
printf("Data = %d\r\n", X);

printf("Data = %d\r\n", X);
```

Консоль:

Test:

Data = 5

Data = 10

Data = 5

Как видите, данная функция никак не повлияла на глобальную переменную. Она как была равна 5, так и осталась. Ведь в этой функции создается копия переменной, которую мы туда вставляем, и ее жизнь длится ровно столько, сколько эта функция выполняется.

Теперь, воспользуемся указателем и сделаем этот пример еще раз:

Создадим функцию:

```
void function_test_2(uint8_t *data) {
	printf("Data = %d\r\n", *data);
	*data = *data + 5;
	printf("Data = %d\r\n", *data);
}
```

Код:

```
printf("Test:\r\n");
//function_test_1(X);
function_test_2(&X);
printf("Data = %d\r\n", X);
```

Консоль:

Test:

Data = 5

Data = 10

Data = 10

Теперь мы не создавали локальную переменную, а работали напрямую с адресом переменной, которую туда вставили, тем самым изменили ее.