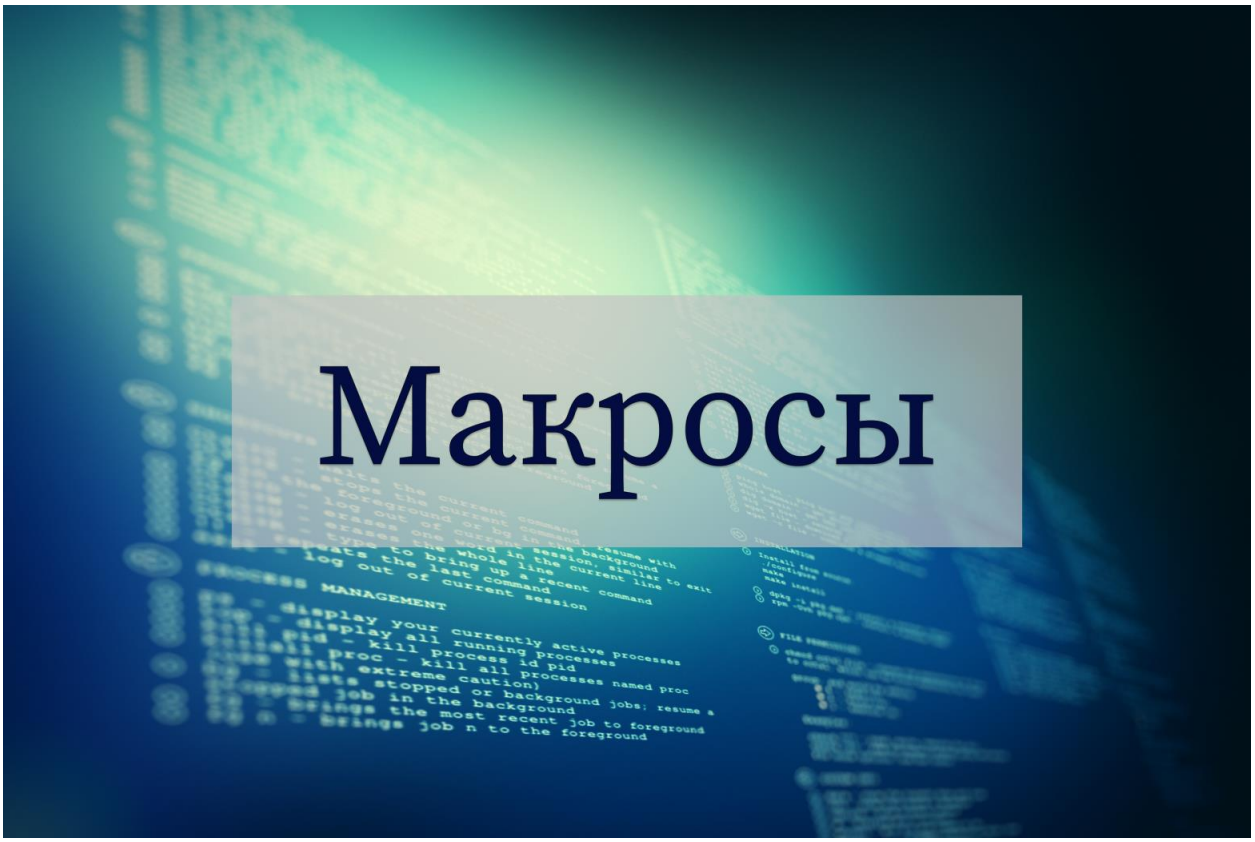
Макросы в C и C++

C++*C*
Из песочницы
Tutorial



Макросы - один из моих самых любимых инструментов в языках C и C++. Умные люди и умные книжки советуют по максимуму избегать использования макросов, по возможности заменяя их шаблонами, константами и inline-функциями, и на то есть веские основания. С помощью макросов можно создавать не только изящный код, но и плодить не менее изящные баги, которые потом будет очень сложно отловить и пофиксить. Но если соблюдать ряд несложных правил при работе с макросами, они становятся мощным оружием, которое не стреляет по твоим собственным коленям. Но сперва разберемся, что вообще такое макросы в C и C++?

Что есть макросы?

В языках C и C++ есть такой механизм, как *препроцессор*. Он обрабатывает исходный код программы ДО того, как она будет скомпилирована. У перпроцессора есть свои директивы, такие как *#include, #pragma, #if* и тд. Но нам интересна только директива *#define*.

В языке Си довольно распространенной практикой является объявление глобальных констант с помощью директивы #define:

```
#define PI 3.14159
```

А потом, на этапе препроцессинга, все использования PI будут заменены указанным значением:

```
double area = 2 * PI * r * r;
```

После препроцессинга, который по сути является банальной подстановкой, это выражение превратится в:

```
double area = 2 * 3.14159 * r * r;
```

PI - макрос, в самом простом его исполнении. Естественно, макросы в таком виде не работают как переменные. Им нельзя присваивать новое значение или использовать их адрес.

```
// Так нельзя:

PI = 3;       // после препроцессинга: 3.14159 = 3

int *x = &PI;    // после препроцессинга: int *x = &3.14159
```

О макросах важно понимать, что область видимости у них такая же, как у нестатических функций в языке Си, то есть они видны везде, куда их "заинклюдили". Однако в отличии от функций, объявление макроса можно отменить:

```
#undef PI
```

После этой строчки обращаться к PI будет уже нельзя.

Макросы с параметрами

Самое интересное начинается, когда у макросов появляются параметры. Параметры в макросах работают примерно так же, как аргументы функции.

Простой пример - макрос, который определяет больший из переданных ему параметров:

```
#define MAX(a, b) a >= b ? a : b
```

Макрос может состоять не только из одного выражения. Например макрос, который меняет значения двух переменных:

```
#define SWAP(type, a, b) type tmp = a; a = b; b = tmp;
```

Поскольку мы первым параметром передаем тип, данный макрос будет работать с переменными любого типа:

```
SWAP(int, num1, num2)

SWAP(float, num1, num2)
```

В подобных макросах, вместо передачи типа аргументов первым параметром, полезно использовать оператор *typeof* в языке C или *decltype в* C++. С их помощью можно удобно объявлять переменную *tmp* того же типа, что и переданные аргументы:

```
#define SWAP(a, b) decltype(a) tmp = a; a = b; b = tmp;
```

Макросы также можно записывать в несколько строк, но тогда каждая строка, кроме последней, должна заканчиваться символом '\':

```
#define SWAP(a, b) \

  decltype(a) tmp = a; \

  a = b; \

  b = tmp;
```

Параметр макроса можно превратить в строку, добавив перед ним знак '#':

```
#define PRINT_VAL(val) printf("Value of %s is %d", #val, val);

int x = 5;

PRINT_VAL(x)  // -> Value of x is 5
```

А еще параметр можно приклеить к чему-то еще, чтобы получился новый идентификатор. Для этого между параметром и тем, с чем мы его склеиваем, нужно поставить '##':

```
#define PRINT_VAL (number) printf("%d", value_##number);

int value_one = 10, value_two = 20;

PRINT_VAL(one)  // -> 10

PRINT_VAL(two)  // -> 20
```

Техника безопасности при работе с макросами

Есть несколько основных правил, которые нужно соблюдать при работе с макросами.

1. Параметрами макросов не должны быть выражения и вызовы функций.

Ранее я уже объявлял макрос MAX. Но что получится, если попытаться вызвать его вот так:

```
int x = 1, y = 5;

int max = MAX(++x, --y);
```

Со стороны все выглядит нормально, но вот что получится в результате макроподстановки:

```
int max = ++x >= --y ? ++x : --y;
```

В итоге переменная *max* будет равна не 4, как мы ожидали, а 3. Потом можно уйму времени потратить, отлавливая эту ошибку. Так что в качестве аргумента макроса нужно всегда передавать уже конечное значение, а не какое-то выражение или вызов функции. Иначе выражение или функция будут вычислены столько раз, сколько используется этот параметр в теле макроса.

2. Все аргументы макроса и сам макрос должны быть заключены в скобки.

Это правило я уже нарушил при написании макроса MAX. Что получится, если мы захотим использовать этот макрос в составе какого-то математического выражения?

```
int result = 5 + MAX(1, 4);
```

По логике, переменная result должна будет иметь значение 9, однако вот что мы получаем в результате макроподстановки:

```
int result = 5 + 1 > 4 ? 1 : 4;
```

И переменная result внезапно примет значение 1. Чтобы такого не происходило, макрос MAX должен быть объявлен следующим образом:

```
#define MAX(a, b) ((a) >= (b) ? (a) : (b))
```

В таком случае все действия произойдут в нужном порядке.

3. Многострочные макросы должны иметь свою область видимости.

Например у нас есть макрос, который вызывает две функции:

```
#define MACRO() doSomething(); \

  doSomethinElse();
```

А теперь попробуем использовать этот макрос в таком контексте:

```
if (some_condition) MACRO()
```

После макроподстановки мы увидим вот такую картину:

```
if (some_condition) doSomething();

doSomethinElse();
```

Нетрудно заметить, что под действие if попадет только первая функция, а вторая будет вызываться всегда. Именно для того, чтобы избежать подобных багов, у макросов должна быть объявлена своя область видимости. Для удобства в этих целях принято использовать цикл *do {} while (0); .*

```
#define MACRO() do { \

    doSomething(); \

    doSomethingElse(); \

  } while(0)
```

Поскольку в условии цикла стоит ноль, он отработает ровно один раз. Это делается, во первых, для того, чтобы у тела макроса появилась своя область видимости, ограниченная телом цикла, а во вторых, чтобы сделать вызов макроса более привычным, потому что теперь после MACRO() нужно будет ставить точку с запятой. Если бы мы просто ограничили тело макроса фигурными скобками, точку с запятой после его вызова поставить бы не получилось.

```
if (some_condition) MACRO();
```

Еще немного примеров

В языке Си при помощи макросов можно эффективно избавляться от дублирования кода. Банальный пример - объявим несколько функций сложения для работы с разными типами данных:

```
#define DEF_SUM(type) type sum_##type (type a, type b) { \

  type result = a + b; \

  return result; \

}
```

Теперь чтобы нагенерировать таких функций для нужных нам типов, нужно просто использовать пару раз этот макрос в глобальной зоне видимости:

```
DEF_SUM(int)

DEF_SUM(float)

DEF_SUM(double)



int main() {

  sum_int(1, 2);

  sum_float(2.4, 6,3);

  sum_double(1.43434, 2,546656);

}
```

Таким образом у нас получился аналог шаблонов из C++. Но стоит сразу обратить внимание, что данный способ не подойдет для типов, название которых состоит более чем из одного слова, например long long или unsigned short, потому что не получится нормально склеить название функции (*sum_##type*). Для этого сперва придется объявить для них новый тип, состоящий из одного слова.

В современном C++ можно спокойно обходиться без макросов вовсе, используя только шаблоны и inline-функции. Но в Си жить с макросами все же удобнее, чем без них. При грамотном использовании макросы позволяют избавиться от большого количества дублирования кода и сделать сам код более симпатичным и удобочитаемым.