

# Макросы в Си: как, когда и зачем?



Программисты Си, дойдя до определённого уровня квалификации, обязательно сталкиваются с одной из особенностей этого языка — [макросами](#). Почти во всех других языках аналога макросов нет. И это неспроста. Использование макросов может оказаться весьма небезопасным. В них скрывается ряд особенностей, специфика которых не всегда лежит на поверхности.

Прим. перев. Макросы в таких языках, как Lisp, Scala и Rust, во многом лишены тех проблем и недостатков, которые описаны в этой статье.

## Макросы и функции

При первом знакомстве макросы могут показаться обычными вызовами функций. Конечно, у них немного странный синтаксис, но они «ведут себя» как обычные функции. Тогда в чём разница?

Макрос можно условно назвать функцией обработки и замены программного кода: после сборки программы макросы заменяются макроопределениями. Вот как это выглядит:

```
#include <stdio.h>

#define SUM(x, y) (x + y)

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 10;
    int sum = SUM(a, b);
    printf("%d\n", sum);
}
```

Этот код преобразуется в следующий:

```
/* обработанный код опущен */

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 10;
    int sum = (a + b);
    printf("%d\n", sum);
}
```

При вызове же функции под неё выделяется новый [стековый кадр](#), и она выполняется самостоятельно и не зависит от места в коде, откуда была вызвана. Таким образом, переменные с

одинаковыми именами в разных функциях не вызовут ошибку, даже если вызывать одну функцию из другой.

```
#include <stdio.h>

void bar()
{
    int var = 10;
    printf("var in bar: %d\n", var);
}

void foo()
{
    int var = 5;
    printf("var in foo: %d\n", var);
    bar();
}

int main(int argc, char *argv[])
{
    foo();
}
```

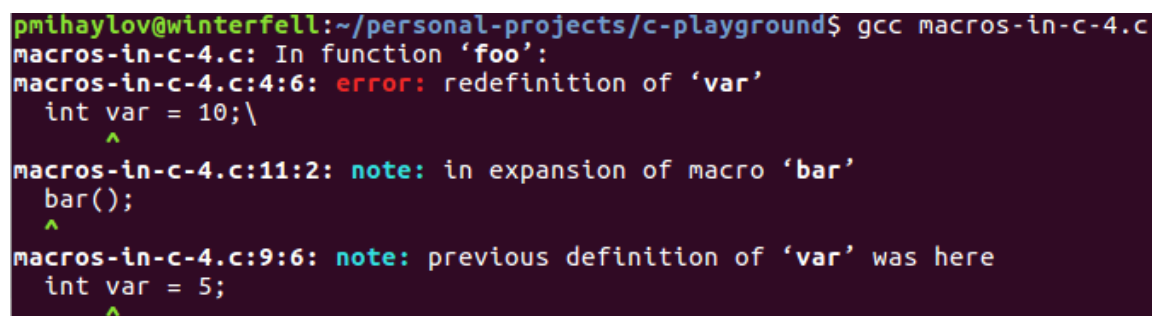
Но если попытаться проделать такой же трюк с помощью макроса, то во время компиляции будет выброшена ошибка, так как обе переменные в итоге будут находиться в одной функции:

```
#include <stdio.h>

#define bar() \
    int var = 10;\
    printf("var in bar: %d\n", var)

void foo()
{
    int var = 5;
    printf("var in foo: %d\n", var);
    bar();
}

int main(int argc, char *argv[])
{
    foo();
}
```



```
pmihaylov@winterfell:~/personal-projects/c-playground$ gcc macros-in-c-4.c
macros-in-c-4.c: In function 'foo':
macros-in-c-4.c:4:6: error: redefinition of 'var'
    int var = 10;\
    ^
macros-in-c-4.c:11:2: note: in expansion of macro 'bar'
    bar();
    ^
macros-in-c-4.c:9:6: note: previous definition of 'var' was here
    int var = 5;
    ^
```

### [Дополнение от редакции](#)

Один из способов решить эту проблему — поместить тело макроса в новую область видимости имён:

```
#include <stdio.h>
#define bar() { \
    int var = 10; \
    printf("var in bar: %d\n", var); \
}
```

```

void foo()
{
    int var = 5;
    printf("var in foo: %d\n", var);
    bar();
}

int main(int argc, char *argv[])
{
    if (condition)
        bar();
    else
        foo();
}

```

Но если просто обернуть его в блок, то может возникнуть такая проблема: блок, в который раскроется использование `bar`, и стоящая после него точка с запятой будут учтены как две разные команды, и в результате, встретив `else`, компилятор выдаст ошибку.

Простым решением было бы не ставить после использования такого макроса точку с запятой, но тогда программисту потребовалось бы помнить о необходимости ставить или не ставить точку с запятой для каждого макроса.

Обычно эту проблему решают с помощью такого трюка:

```

#include <stdio.h>
#define bar() do { \
    int var = 10; \
    printf("var in bar: %d\n", var); \
} while (0)

int main(int argc, char *argv[])
{
    if (condition)
        bar();
    else
        baz();
}

```

Такой цикл выполнится только один раз, но поскольку конструкция `do-while` в Си требует точки с запятой после условия, стоящая после макроса точка с запятой будет отнесена к нему, а не воспринята как отдельная команда.

Более того, функции выполняют проверку типов: если функция ожидает на входе строку, а получает число, будет выброшена ошибка (или, по крайней мере, предупреждение, в зависимости от компилятора). Макросы в то же время просто заменяют аргумент, который им передан.

И, наконец, макросы не подлежат отладке. В отладчике можно войти в функцию и пройтись по её коду, а вот с макросами такое не пройдёт. Поэтому, если макрос почему-то сбоит, единственный способ выявить проблему — переходить к его определению и разбираться уже там.

Прим. перев. Чтобы не переходить к определению каждого макроса, можно попросить компилятор раскрыть макросы — это можно сделать с помощью команды `gcc -E source.c`. Имейте в виду, что если вы включаете в свой код с помощью `#include` стандартные заголовочные файлы, после препроцессинга в коде может оказаться много тысяч строк, так что стоит перенаправить вывод компилятора в файл.

Тем не менее, можно выделить одно явное преимущество макросов перед функциями — производительность. Макрос быстрее, чем функция. Как уже упоминалось выше, под функцию

выделяются дополнительные ресурсы, которые можно сэкономить, если использовать макросы. Это преимущество может сыграть весомую роль в системах с ограниченными ресурсами (например, в очень старых микроконтроллерах). Но даже в современных системах программисты производят оптимизации, используя макросы для небольших процедур.

Прим. перев. Интересный подход к оптимизации использования ресурсов в программе на Си рассмотрен в [другой нашей статье](#).

В C99 и C++ существует альтернатива макросам — [встраиваемые \(inline\) функции](#). Если добавить ключевое слово `inline` перед функцией, компилятору будет дано указание включить тело функции в место её вызова (по аналогии с макросом). При этом встраиваемые функции могут быть отлажены, и у них есть проверка типов.

Однако ключевое слово `inline` — это просто подсказка для компилятора, а не строгое правило, и компилятор может проигнорировать эту подсказку. Чтобы этого не произошло, в gcc есть атрибут `always_inline`, который заставляет компилятор встроить функцию.

Встраиваемые функции — отличная штука, которая, как может показаться, делает использование макросов нецелесообразным. Однако это не совсем так.

## Когда использовать макросы в Си

### Передача аргументов по умолчанию

В C++ есть весьма удобный инструмент, которого нет в Си, — [аргументы по умолчанию](#):

```
#include <iostream>

using namespace std;

void printError(int errorCode, string msg = "No message")
{
    cerr << "Error code: " << errorCode << " (" << msg << ")\n";
}

int main(int argc, char *argv[])
{
    printError(9, "Bad alloc");
    printError(8);
}
```

В Си задача опциональных аргументов может быть решена с помощью макросов:

```
#include <iostream>

#define printErrord(errorCode) printError(errorCode, "No message")

void printError(int errorCode, char *msg)
{
    printf("Error code: %d (%s)\n", errorCode, msg);
}

int main(int argc, char *argv[])
{
    printError(9, "Bad alloc");
    printErrord(8);
}
```

Этот код довольно безопасен и не скрывает никаких подводных камней (их мы обсудим далее). Конечно, эту задачу можно также решить с помощью функции, но она достаточно тривиальна, чтобы нести накладные расходы при использовании функций. Макрос справляется с ней на ура.

## Использование отладочных строк

Некоторые компиляторы [предопределяют](#) макросы, которые нельзя использовать в функциях:

```
__FILE__, __LINE__, __func__.
```

Их можно включать в определения макросов, используемых для отладки:

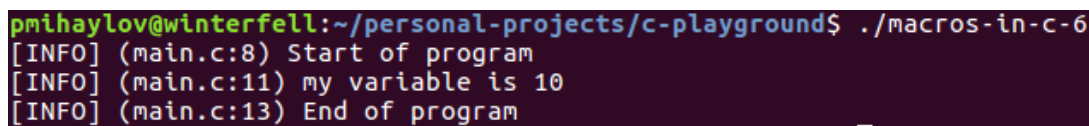
```
#include

#define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n", \
    __FILE__, __LINE__, ##__VA_ARGS__)

int main(int argc, char *argv[])
{
    log_info("Start of program");
    int var = 10;

    log_info("my variable is %d", var);

    log_info("End of program");
}
```



```
pmihaylov@winterfell:~/personal-projects/c-playground$ ./macros-in-c-6
[INFO] (main.c:8) Start of program
[INFO] (main.c:11) my variable is 10
[INFO] (main.c:13) End of program
```

В итоге получается интересный способ ведения логов.

## Модификация синтаксиса

Это очень мощная особенность макросов. Используя её, можно создать свой собственный синтаксис.

Например, в Си нет конструкции `foreach`. Но её можно создать через макрос:

```
#include <stdio.h>

struct ListNode;
typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    void *value;
} ListNode;

typedef struct List {
    int count;
    ListNode *first;
    ListNode *last;
} List;

#define LIST_FOREACH(curr, list) \
    ListNode *curr = list->first; \
    for (ListNode *_node = list->first; \
        _node != NULL; \
        curr = _node = _node->next)

int main(int argc, char *argv[])
```

```

{
    List *lst;

    /* Fill the linked list */

    LIST_FOREACH(curr, lst)
    {
        printf("%d\n", (int)curr->value);
    }
}

```

В этой вставке кода определена структура, содержащая [связный список](#). Предполагая, что его узлы заполнены, можно пройти по нему с помощью `LIST_FOREACH` так же, как и при использовании `foreach` в современных языках.

Эта техника действительно очень эффективна и при правильном использовании может дать довольно хорошие результаты.

## Другие типы макросов

Помимо макросов, которые подменяют функции, есть и другие очень полезные директивы препроцессора. Вот некоторые из наиболее часто используемых:

- `#include` — включить содержимое стороннего файла в текущий файл,
- `#ifdef` — задать условие для компиляции,
- `#define` — определить константу (и, конечно же, макрос).

`#ifdef` играет ключевую роль при создании заголовочных файлов. Использование этого макроса гарантирует, что заголовочный файл включён только один раз:

```

#ifndef MACROS_IN_C_H
#define MACROS_IN_C_H

/* Прототипы функций */

#endif // MACROS_IN_C_H

```

Стоит отметить, что основное предназначение `#ifdef` — условная компиляция блоков кода на основе некоторого условия. Например, вывод отладочной информации только в режиме отладки:

```

#include <stdio.h>

#ifdef DEBUG
    #define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n", \
        __FILE__, __LINE__, ##__VA_ARGS__)
#else
    #define log_info(M, ...)
#endif

int main(int argc, char *argv[])
{
    log_info("Start of program");

    int var = 10;
    printf("my variable is %d\n", var);

    log_info("End of program");
}

```

```
pmihaylov@winterfell:~/personal-projects/c-playground$ gcc -DDEBUG main.c
pmihaylov@winterfell:~/personal-projects/c-playground$ ./a.out
[INFO] (main.c:12) Start of program
my variable is 10
[INFO] (main.c:17) End of program
pmihaylov@winterfell:~/personal-projects/c-playground$ gcc main.c
pmihaylov@winterfell:~/personal-projects/c-playground$ ./a.out
my variable is 10
```

Прим. перев. С помощью `#ifdef` также довольно часто задаётся условие для компиляции на различных версиях ОС и архитектурах: `#ifdef _WIN32`, `#ifdef _WIN64`, `#ifdef __linux__` и так далее.

Как правило, для определения констант используется `#define`, но в некоторых проектах его заменяют на `const` и перечисления (`enum`). Однако при использовании любой из этих альтернатив есть свои преимущества и недостатки.

Использование ключевого слова `const`, в отличие от макроподстановки, позволяет произвести проверку типов данных. Но в Си это создаёт не совсем полноценные константы. Например, их нельзя использовать в операторе `switch-case` и для определения размера массива.

Прим. автора В C++ переменные, определённые ключевым словом `const`, являются полноценными константами (их можно использовать в приведённых выше случаях), и настоятельно рекомендуется использовать именно их, а не `#define`.

Перечисления в то же время — полноценные константы. Они могут использоваться в операторах `switch-case` и для определения размера массива. Однако их недостаток заключается в том, что в перечислениях можно использовать только целые числа. Их нельзя использовать для строковых констант и констант с плавающей запятой.

Вот почему использование `#define` — оптимальный вариант, если нужно достичь единообразия в определении различного рода констант.

Таким образом, у макросов есть свои преимущества. Но использовать их нужно весьма аккуратно.

## Подводные камни при использовании макросов

### Отсутствие скобок

Наиболее распространённая ошибка, которую допускают при использовании макросов, — отсутствие скобок вокруг аргументов в определениях макросов. Поскольку макросы подставляются непосредственно в код, это может вызвать неприятные побочные эффекты:

```
#include <stdio.h>

#define MULTIPLY(x) (x * 5)

int main(int argc, char *argv[])
{
    int x = 5;

    int result = MULTIPLY(x + 5);

    printf("RESULT: %d\n", result);
}
```

В этом примере выполняется вычисление `MULTIPLY(x + 5)` и ожидаемый результат — 50. Но в процессе подстановки произойдёт следующее преобразование:

```
MULTIPLY(x + 5) -> (x + 5 * 5)
```

Как несложно подсчитать, данное выражение выдаст не 50, а 30.

А вот как выполнить данную задачу правильно:

```
#include <stdio.h>

#define MULTIPLY(x) ((x) * 5)

int main(int argc, char *argv[])
{
    int x = 5;

    int result = MULTIPLY(x + 5);

    printf("RESULT: %d\n", result);
}
```

## Инкремент и декремент

Допустим, есть такой код:

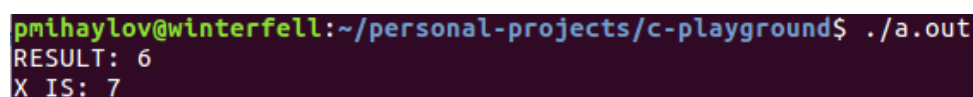
```
#include <stdio.h>

#define ABS(x) ((x) < 0 ? -(x) : (x))

int main(int argc, char *argv[])
{
    int x = 5;
    int result = ABS(x++);

    printf("RESULT: %d\n", result);
    printf("X IS: %d\n", x);
}
```

Здесь можно ожидать, что `x` будет увеличен на единицу и будет равен 6, а результат — 5. Но вот что получится в реальной жизни:



```
pmihaylov@winterfell:~/personal-projects/c-playground$ ./a.out
RESULT: 6
X IS: 7
```

Виновата всё та же макроподстановка: `ABS(x++) -> ((x++) < 0 ? -(x++) : (x++))`

Как видно, `x` увеличивается на единицу в первый раз при проверке и во второй раз при определении результата, что и приводит к соответствующим итогам.

## Передача вызовов функций

Использованием функции в коде никого не удивишь. Равно как и передачей результата одной функции в виде аргумента для другой. Часто это делается так:

```
#include <stdio.h>

int bar()
{
    return 10;
}

void foo(int num)
```



```

{
    printf("%d\n", num);
}

int main(int argc, char *argv[])
{
    foo(bar());
}

```

И в этой вставке кода всё в порядке. Но, когда это же производится с помощью макроса, можно столкнуться с серьёзными проблемами производительности. Допустим, есть вот этот код:

```

#include <stdio.h>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

int sum_chars(char *chars)
{
    if ((*chars) == '\0') return 0;

    return sum_chars(chars + 1) + (*chars);
}

int main(int argc, char *argv[])
{
    char *str1 = "Hello world";
    char *str2 = "Not so fast";

    int minCharSum = MIN(sum_chars(str1), sum_chars(str2));
    printf("MIN CHARS: %d\n", minCharSum);
}

```

Здесь определена рекурсивная функция `sum_chars`. Она вызывается один раз для первой строки (`str1`) и другой раз — для второй (`str2`). Но, если передать вызовы функций, как аргументы для макроса, будет выполнено три рекурсивных вызова вместо двух. Для больших структур данных это станет узким местом производительности. Особенно, если макрос используется внутри рекурсивной функции.

## Многострочные макросы

Программисты не всегда используют фигурные скобки вокруг единичных команд в циклах и условных операторах. Но, если произвести макроподстановку внутри этого блока, и этот макрос будет содержать несколько строк, это приведёт к весьма интересным результатам:

```

#include <stdio.h>

#define MODIFY(arr, index)\
    arr[index] *= 5;\
    index++;

int main(int argc, char *argv[])
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int i = 0;

    while (i < 5)
        MODIFY(arr, i);

    for (i = 0; i < 5; ++i)
    {
        printf("ELEMENT %d: %d\n", i, arr[i]);
    }
}

```

Во вставке кода выше нет фигурных скобок в первом цикле и, так как макрос заменяет одну строку несколькими, только первое выражение в макросе выполняется в цикле. Следовательно, это приведёт к бесконечному циклу, так как `i` никогда не увеличится.

Прим. перев. Эту проблему также можно решить с помощью упомянутого выше трюка с `do {} while (0)`.

Именно из-за таких особенностей многие стараются избегать использования макросов.

## Хорошая практика

Чтобы свести к минимуму проблемы, вызванные использованием макросов, хорошей практикой будет использование единого подхода для определения макросов в вашем коде. Каким будет этот подход, не имеет значения. Есть проекты, в которых все макроопределения объявлены в верхнем регистре. В некоторых проектах в начале имени макроса используют букву «m». Выберите себе любой подход, но этот подход должен быть таким, чтобы и вы, и другой программист, который будет работать с вашим кодом, сразу понимал, что имеет дело с макросами.

Прим. перев. Другие полезные практики оформления кода можно посмотреть в [нашей статье](#).

## Вывод

В большинстве случаев программисты предпочитают использовать функции, поскольку макросы скрывают некоторые серьёзные побочные эффекты. Тем не менее, использование макросов иногда более целесообразно. В этом случае нужно соблюдать правила их именования и учитывать специфику их использования, рассмотренную в статье.

Перевод статьи [«How to properly use macros in C»](#)