Десять применений define-ов в языке Си

Микоян Филипп

20 февраля 2017 г.

1 Краткое введение в макросы

1.1 Терминология

- **Препроцессор** - программа, обрабатывающая исходные файлы перед компиляцией.
- **Директива** - указание.
- **Макрос** - символьное имя, заменяемое препроцессором на последовательность символов, возможно зависящую от аргументов макроса. Макросы могут применяться для сокращения записи повторяющихся/мало отличающихся последовательностей инструкций.
- #**define** - директива препроцессору создать макрос, т.е. заменить все последующие употребления имени макроса на определённую последовательность символов.

1.2 Синтаксис

- **Макрос без аргументов**
  $\#define\ identifier\ token - stringopt$

Листинг 1: Пример применения макроса без аргументов

```
1  #include <stdio.h>
2
3  #define mew int
4
5  int main(void)
6  {
7      mew a = 5;
8      printf("%i", a/2); // 2 is to be printed
9      return 0;
10 }
```

Этот пример, вероятно, не совсем безопасного применения макроса иллюстрирует, как можно 'переименовать' тип целочисленной переменной.

- **Макрос с аргументами**
  $\#define\ identifier\ (identifieropt1,\ ...,\ identifieroptn)\ token\_sequence$

Листинг 2: Пример применения макроса с аргументами

```
#include <stdio.h>

#define sqr(x) ((x)*(x))

int main(void)
{
    int a = 5;
    printf("%i", sqr(a)); //25 is to be printed
    return 0;
}
```

В данном примере каждый *'x'* из выражения $((x)*(x))$ непосредственно заменяется на то, что подаётся в макрос в качестве аргумента, будь то строка, выражение, символ, идентификатор, число и т.д.

2 Macros ≈ Evil

Макросы следует использовать с осторожностью, придерживаясь следующих правил:

- В названиях макросов употреблять только заглавные буквы, использовать префиксы, содержащие имя проекта и/или названия, потенциально редко применяемые остальными программистами
- Использовать макросы только если это оправдано, всегда задаваться вопросом о возможности замены макроса функцией
- Описывать макросы с простой реализацией
- Описывать только те макросы, которые не требуют отладки
- Избегать макросов, неявно что-либо изменяющих
- В случае необходимости описания больших макросов/макросов со сложной реализацией, обязательно снабжать их комментариями

В случае соблюдения вышеописанных правил, единственным серьёзным недостатком макросов можно назвать разве что "разбухание"исходного файла, обработанного препроцессором, и, как следствие, более длительную компиляцию и увеличение размера скомпилированной программы.

3 Допустимые оправдания использования макросов

Макросы обладают несколькими достоинствами, которые, по моему мнению, могут являться единственным оправданием их использования. К ним относятся следующие особенности макросов(перечислены в порядке убывания важности):

- Позволяют избежать копипаста
- Подчастую могут увеличить читабельность кода
- Позволяют узнать *имя* переменной, переданной в макрос, а не только её значение
- Позволяют *слить* два токена в один
- Инвариантность вида макроса относительно типов передаваемых в него аргументов (см. п.4.3)
- Используются при создании *#include guard*-ов
- Вкупе с остальным функционалом препроцессора(в частности, include-ами) позволяет выносить некоторые описания в отдельные файлы, которые можно назвать конфигурационными
- Предописанные макросы бывают незаменимы
- При грамотном применении создают новый уровень абстракций(в Си позволяет следовать принципам Domain-driven design, имитировать Domain-specific language)

4 Типичные 'use cases'(примеры применения) макросов

4.1 'Пустые' макросы

4.1.1 'Флажки' в исходном коде

Бывает удобно использовать 'пустой' макрос, например, 'Govnocode', чтобы помечать в исходном коде те места, которые следует отрефакторить в первую очередь, и которые, в связи с близостью дедлайна, трудно написать хорошо сразу.

Листинг 3: Govnocode

```
#include <stdio.h>

#define Govnocode

const int CONST1 = 666;
const int CONST2 = 678;

int main(void)
{
    char c = '\0';
    while (!scanf("%c", &c));
    switch (c)
    {               Govnocode //Replace with macros
        case '^':
            {
                printf("%d", CONST1 ^ CONST2);
            }; break;
        case '+':
            {
                printf("%d", CONST1 + CONST2);
            }; break;
        case '-':
            {
                printf("%d", CONST1 - CONST2);
            }; break;
        case '*':
            {
                printf("%d", CONST1 * CONST2);
            }; break;
        case '/':
            {
                printf("%d", CONST1 / CONST2);
            }; break;
        default: return 1;
    }
    return 0;
}
```

4.1.2 #include guards

#include guard - конструкция, используемая с целью избежать "двойное подключение"директивой *#include*.

Листинг 4: #include guard

```
#ifndef SUPER_HEADER_INCLUDED
#define SUPER_HEARER_INCLUDED

struct SuperStruct
{
    int superValue;
    char superChar;
};

#endif
```

4.1.3 Условная компиляция

Условная компиляция - совокупность директив препроцессора, позволяющая компилировать или пропускать часть программы в зависимости от выполнения некоторого условия, в частности, от существования того или иного макроса. К директивам условной компиляции относятся **#ifdef**, **#ifndef**, **#else**, **#endif**

Листинг 5: Условная компиляция

```
#include <stdio.h>

//#define TEST_MODE
//#define SILENT_MODE

int main(void)
{
#ifndef SILENT_MODE

#ifdef TEST_MODE
    printf("Test mode");
#else  //TEST_MODE
    printf("Normal mode"); //This is to be printed
#endif //TEST_MODE

#endif //SILENT_MODE
    return 0;
}
```

Следует отметить, что т.к. директивы условной компиляции обычно пишут без ведущих пробелов, т.е. без отступов, следует снабжать каждую директиву **#else**, **#endif** комментарием, указывающим, какой блок условной компиляции даннная директива закрывает.

4.2 Именование констант

Зачастую начинающие программисты используют макросы для именования констант, к примеру:

Листинг 6: Именованная константа

```
1 #define MAX_STRING_LENGTH 256
```

Эта практика небезопасна, т.к. вместо числа '256' программист может случайно написать, к примеру, вызов функции, о чём впоследствии забыть. Это может привести к долгой, изнурительной и безрезультатной отладке, а потому вместо макросов в данном случае рекомендуется использовать enum. В случае, если несколько констант имеет смысл объединить в одну группу, следует поместить их в один enum.

Листинг 7: Enums

```
1 enum { MAX_STRING_LENGTH = 256 }; //Single constant
2
3 enum STKERRORS //Block of constants
4 {
5   STKERR_NULLPTR,
6   STKERR_NULLDATAPTR,
7   STKERR_NEGSIZE,
8 };
```

4.3 Макросы как функции

Вообще говоря, макросы всегда нужно стараться заменить функциями, т.к. их проще отлаживать и они более безопасны. Единственным обоснованием применения макросов вместо функций может являться то, что один и тот же макрос может принимать аргументы разных типов.

Листинг 8: Macro Function

```
#include <stdio.h>

//#define DUMB_SQR

#ifdef DUMB_SQR
double sqrD(double a)
{
    return a * a;
};

int sqrI(int a)
{
    return a * a;
};

char sqrC(char a)
{
    return a * a;
};
#else

#define sqr(a) ((a)*(a))

#endif

int main(void)
{
    printf("%i", sqr(5));
    return 0;
}
```

В данном подпункте следует особо отметить, что в макросах нужно ставить как можно больше скобок, учитывать все возможные значения аргументов. Это проиллюстрировано в следующем примере:

Листинг 9: Wrong mul

```
#include <stdio.h>

#define mul(a, b) a * b //Wrong 'mul' macro - not enough brackets.

const int CONST = 666;

int main(void)
{
    printf("%i", mul(CONST + 5, 3)); //681 is to be printed, 2013 expected
}
```

4.4 Макросы для получения имени переменной

Листинг 10: Получение имени переменной макросов

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_STRING_LEN = 256 };

struct Point
{
    double x;
    double y;
};

void pointDump_(struct Point* point, const char pointName[MAX_STRING_LEN])
{
    assert(point);

    printf("struct Point '%s' [%p]\n", pointName, point);
    printf("{\n"
           "\tx = %lg\n"
           "\ty = %lg\n"
           "}\n", point->x, point->y);
}
#define pointDump(pnt)                                  \
{                                                       \
    assert(strlen(#pnt) <= MAX_STRING_LENGTH);          \
    pointDump_(pnt, #pnt);                              \
}

int main(void)
{
    struct Point* mySuperPoint = (struct Point*)calloc(1, sizeof(*
        mySuperPoint));
    pointDump(mySuperPoint);
    return 0;
}
```

В данном примере (*#pnt*) - строковое представление переданного в макрос параметра.
Вывод программы Point dump.c:

```
struct Point 'mySuperPoint' [0x55fa8e58b010]
{
    x = 0
    y = 0
}
```

Обратите внимание на обратные слэши в конце каждой строки в описании макроса. Они используются для описания многострочных макросов. Их нужно ставить в конце строки(обратите внимание, что после них не должны стоять даже пробелы), которую необходимо перенести.

4.5 Макросы для склейки токенов

Листинг 11: Склейка токенов макросами

```
int compare_chicks(const void *chick1, const void *chick2)
{
    #define boobs(i)   (*((const int **)chick##i)) [0]
    #define waist(i)   (*((const int **)chick##i)) [1]
    #define hippies(i) (*((const int **)chick##i)) [2]
    #define sum(i)     (boobs(i) + waist(i) + hippies(i))

    int sum1 = sum(1);
    int sum2 = sum(2);

    int firstChickIsBest = (sum1 < sum2);

    if (sum1 == sum2)
    {
        firstChickIsBest = boobs(1) > boobs(2);
        if (boobs(1) == boobs(2))
        {
            firstChickIsBest = waist(1) < waist(2);
            if (waist(1) == waist(2))
            {
                firstChickIsBest = hippies(1) > hippies(2);
            }
        }
    }

    return !firstChickIsBest;

    #undef sum
    #undef boobs
    #undef waist
    #undef hippies
}
```

В данном примере особое внимание следует обратить на двойные решётки('##') в описаниях макросов. Если бы их не было, препроцессор воспринял бы последовательность символов *chicki* как идентификатор и напрямую заменял бы, скажем, *waist(0)* на *(*((const int **)chicki)) [1]*, т.е. это выражение даже не зависело бы от передаваемого параметра.
С помощью двойной решётки мы указываем компилятору, что эти две последовательности символов(*chick* и *i*) следует сначала проверить на необходимость замены на аргументы, а затем 'склеить'.
Также следует отметить присутствие директив препроцессору *#undef*. Эти директивы используются для ограничения области видимости макросов. Их обязательно следует ставить там, где кончается область видимости макроса.
Отмечу также, что чем шире область видимости макросов, тем 'привередливей' следует относиться к их названиям, и наоборот.

4.6 Макросы для замены однотипных case-ов и функций

Листинг 12: Switch and macros

```
#include <stdio.h>

const int CONST1 = 666;
const int CONST2 = 678;

int main(void)
{
    char c = '\0';
    while (!scanf("%c", &c));

#define caseOperator(operChar, oper)       \
case operChar:                             \
    {                                      \
        printf("%d", CONST1 oper CONST2); \
    }; break;

    switch (c)
    {
        caseOperator('^', ^);
        caseOperator('+', +);
        caseOperator('-', -);
        caseOperator('*', *);
        caseOperator('/', /);
        default: return 1;
    }

#undef caseOperator
    return 0;
}
```

Этот пример представляет собой переделанный Листинг 3 из пункта 4.1.1.

В данном листинге хорошо проиллюстрировано, как можно использовать макросы для устранения копипаста - переписывания/копирования участка кода в разные части программы. Метод состоит в следующем: создаётся макрос, целиком содержащий в себе этот участок кода. Если этот участок в зависимости от той части программы, в которую его нужно вставить, слегка изменяется, то эти изменения производятся путём введения аргументов макроса, отличающихся в разных частях программы.

Листинг 13: Functions and macros

```
#include <stdio.h>

#define sqrFunc(varType)          \
varType sqr_##varType(varType a); \
                                  \
varType sqr_##varType(varType a)  \
{                                 \
    return a * a;                 \
};

sqrFunc(double)
sqrFunc(int)
sqrFunc(char)

#undef sqrFunc

int main(void)
{
    printf("%i", sqr_int(5));
    return 0;
}
```

Данный пример представляет собой переделанный Листинг 8 из пункта 4.3.

Кроме приведённых примеров к данному пункту также можно отнести, например, применение макросов для описания похожих функций, отличающихся лишь наборов передаваемых в них параметров.

4.7 Макросы, описанные в отдельных(≈ конфигурационных) файлах

Листинг 14: Файл Enums.h, использующий макросы, описанные в Keywords.h

```
#ifndef ENUMS_H_INCLUDED
#define ENUMS_H_INCLUDED

enum LangKeyword
{
    #define LANG_KEYWORDS

    #define LANG_KEYWORD(keywd, type) LANGKWD_##keywd,
    #include "Keywords.h"
    #undef LANG_KEYWORD

    #undef LANG_KEYWORDS
    LANGKWD_KWDQT,
};


```

```
18 enum LangKeywordType
19 {
20     #define LANG_TYPES
21
22     #define LANG_TYPE(type) LANGKWT_##type,
23     #include "Keywords.h"
24     #undef LANG_TYPE
25
26     #undef LANG_TYPES
27 };
28
29 #endif
```

Листинг 15: Файл, использующий макросы, описанные в Keywords.h

```
1  #ifdef LANG_MATH_OPERATORS
2  //  LANG_MATH_OP( name, char, priority )
3  //+-------------------------------------------+
4      LANG_MATH_OP( ln     , 'l', 0             )
5  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  //--------------------------------------------
7  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8      LANG_MATH_OP( minus , '-', 3              )
9  //+-------------------------------------------+
10 #endif
11
12 #ifdef LANG_KEYWORDS
13 //+-------------------------------------------------+
14 //  LANG_KEYWORD( keyword,  keyword_type      )
15 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
16 //--------------------------------------------------
17 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
18     LANG_KEYWORD( while,     loop                 )
19 //+-------------------------------------------------+
20 #endif
21
22 #ifdef LANG_TYPES
23 //+-------------------------------------------+
24 //  LANG_TYPE( type              )
25 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
26 //--------------------------------------------
27 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
28     LANG_TYPE( function          )
29 //+-------------------------------------------+
30 #endif
```

4.8 Getter и setter, реализованные с помощью макросов

5 Опция -E

Отдельный заголовок хотелось бы выделить, чтобы обратить внимание читателя на опцию компилятора '-E', которая позволяет увидеть исходные файлы такими, какими они стали после обработки препроцессором, т.е. со всеми 'раскрытыми' '$\#define$-ами и $\#include$-ами.

Начинающему пользователю макросов настоятельно рекомендуется воспользоваться данной опцией и изучить возвращённые препроцессором файлы. Также бывает очень полезно воспользоваться данной опцией при неожиданном и необъяснимом поведении программы, в которой, возможно, построена чересчур сложная система макросов.