Ольга Кононова, Антон Москаль, Олег Плисс

Расширенный макропроцессор для языка C/C++

Содержание

Аннотация

Описываемый макропроцессор является в основном расширением стандартного макропроцессора и может использоваться как вместо него, так и в сочетании с ним. Строгости ради следует отметить, что имеется довольно много мелких несовместимостей со стандартным препроцессором. Как показал опыт его использования, эти несовместимости практически не вызывают проблем. Устранить их не представляется возможным по соображениям внутренней логики.

Основными расширениями являются:

- возможность описания многострочных макрокоманд, которых сами могут содержать операторы макропроцессора.
- возможность использовать в макрокомандах произвольное число аргументов и произвольные разделители между ними.
- возможность использовать различные типы скобок ([], ( ), { }, #(#))
- возможность перегрузки макроопределений по типу скобок и числу аргументов.
- циклы по спискам и присваивания значений переменным

Макропроцессор написан на C++ с использованием самого себя. В настоящее время существуют его реализации для MS-DOS, OS/2 и Linux.

1. Лексика

Макропроцессор является лексически ориентированным. Лексика в основном соответствует стандартной лексике *C*. Добавлены новые лексемы: "..", "@". в начале идентификаторов допускается использование символа $. Кроме того, разрешается обозначение символа перевода строки при помощи комбинации $\backslash n$. Таким образом:

#ifdef *__BC__* $\backslash n$ **#include** "stdio.h"$\backslash n$ **#endif**

по своему действию эквивалентно

#ifdef *__BC__*
    #include "stdio.h"
#endif

Данная возможность полезна в некоторых случаях для увеличения читаемости текста.

Кроме того, \ может быть использован для подавления распознавания макрооператоров. Так:

#include "xxx.xx"

будет выведен в выходной поток без изменений, но с удаленным символом \.

1.1. Основные конструкции

1.1.1. Разделители

Разделители используются для разделения формальных и фактических аргументов макросов.

<delimiter> ::=
, | ; | / | < | > | − | % | == | ! =
↑ | & | && | ∗ | + | = | <= | >= |
: | ? | / | . | .. | ! | | | || | + = | − = |
% = | ↑= | & = | | = | ∗ = | / =

1.1.2. Список формальных параметров

Данная конструкция используется в описаниях макросов и операторе # **with**.

<formal parameters list> ::=
    <formal parameters list$_1$> |
    <formal parameters list$_1$>;

Точка с запятой обозначает, что соответствующий список фактических параметров тоже должен завершаться точкой с запятой, которая входит в него (то есть заменяется при макроподстановке).

<formal parameters list$_1$> ::=
<empty> |
(<formal parameters list$_2$>) |
[<formal parameters list$_2$>] |
{<formal parameters list$_2$>} |
#(<formal parameters list$_2$>#)

<formal parameters list$_2$> ::=
<empty> |
<formal parameters list$_3$> |
<delimiter><formal parameters list$_3$>

Разделитель перед списком формальных параметров используется для задания разделителя в случае, если формальный параметр один (это может быть необходимо для правильного определения числа параметров, в особенности при использовании перегруженных макросов)

<formal parameters list$_3$> ::=
<formal parameters list$_4$> |
<formal parameters list$_4$><delimiter>...

Многоточие в конце списка обозначает, что последний параметр может ассоциироваться с произвольным (непустым) списком аргументов.

<formal parameters list$_4$> ::=
<formal parameter> |
<formal parameter><delimiter>
<formal parameters list$_4$>

Все разделители в списке должны быть различны.

1.1.3. Список фактических параметров

Список фактических параметров выглядит традиционно, но имеет следующие важные отличия:

Могут использоваться все четыре вида скобок. При выделении аргументов единым аргументом считаются не только конструкции заключенные в круглые скобки, но и заключенные в скобки других типов. Это является отклонением от семантики стандартного макропроцессора. Если операнд окружен скобками $\#(\ -\ \#)$, то они удаляются перед подстановкой этого операнда. Удаление производится только с самыми внешними скобками (и только в том случае, если они окружают весь аргумент - с аргумента вида $\#(a, b, c\#)d$ скобки удалены не будут. Скобки $\#(\ -\ \#)$ используются для предотвращения анализа внутренней структуры параметров при вызове макрокоманды.

Кроме того, в список фактических параметров может включаться завершающая точка с запятой.

2. Операторы макропроцессора

2.1. Оператор описания макроса (#macro)

<statement **#macro**> ::=
#macro<name><formal parameters list>
<macro body>
#endm

<macro body> – это текст, содержащий сбалансированное количество скобок # **macro** – #**endm**. Ни один из операторов, содержащихся в теле макроса в процессе обработки макроопределения не выполняется.

2.2. Оператор #define

Оператор #**define** имеет обычный синтаксис, за исключением того, что список формальных параметров может иметь общий вид. С точки зрения семантики этот оператор может рассматриваться как сокращенный вариант оператора #**macro**. Его использование может быть рекомендовано как средство для достижения совместимости отдельных фрагментов со стандартным макропроцессором и для сокращения записи простейших макросов.

Использование в нем других макрооператоров не рекомендуется, так как семантика таких макросов может оказаться неочевидной.

2.3. Операторы #ifdef, #ifndef, #if, #else, #elif, #endif, #error

Данные операторы имеют обычную семантику с тем отличием, что вычисления в арифметических выражениях в операторе #**if** производятся над строковым представлением (и возможно использование строковых функций и макросов).

2.4. Оператор #for

Оператор #**for** служит для перебора элементов списка и имеет вид:

<statement **#for**> ::=
#for<name><formal parameters list>
<loop body>
#endf |

#for<name>**#by**<delimiter><formal parameters list>
<loop body>
#endf

<name> является именем переменной цикла. <formal parameters list> разделяется разделителем (по умолчанию – “,”) на компоненты. Семантика оператора #**for** эквивалентна семантике описания макроса с одним параметром и последующим вызовом этого макроса для каждого элемента списка.

2.5. Оператор #undef

Этот оператор отличается от стандартного тем, что вместо одного идентификатора может быть задан список идентификаторов, разделенных запятой.

Еще одно отличие состоит в том, что операторы #**define** и #**macro** для одноименных макросов не замещают старое определение, а образуют стек. Оператор #**undef**, соответственно, удаляет последнее значение из стека, делая доступным предыдущее. Это очень удобно для организации локальных переменных и макро.

2.6. Оператор #setm

<statement #**setm**> ::=
#**setm** <name>
<new value>
#**ends**

Этот оператор устанавливает новое значение тела макро с именем <name> , которая должна быть уже существующей макро без параметров.

2.7. Оператор #set

<statement #**set**> ::= #**set**<name><new value>

Этот оператор похож на #**setm**, за исключением того, что новое значение не может содержать несколько строк. Действие оператора

#**set** <id> <expr>

может быть выражено следующим образом:

#**setm** <id>
<expr>\
#**ends**

2.8. Оператор #include

Единственным отличием оператора #**include** от стандартного является то, что в режиме -a при отсутствии указанного в операторе файла вместо выдачи диагностики об ошибке происходит вывод в выходной файл самого оператора #**include**. Эта опция применяется при использовании расширенного макрогенератора в паре со стандартным.

2.9. Оператор #while

<statement #**while**> ::=
#**while**<experssion>
<loop body>
#**endw**

Этот оператор выполняет <loop body> до тех пор, пока выполняется условие <experssion> .

2.10. Оператор #with

<statement #**with**> ::=
#**with** <formal parameters list>\
<actual parameters list>
<statment body>
#**end_with** |

#**with** <formal parameters list>\
<actual parameters list>
<statement body>
#**welse**
<welse – part>
#**end_with**

Оператор #**with** выполняет разбор <actual parameters list> в соответствии со <format parameters list> и подставляет их в <statment body> как в тело макроса. При несоответствии списка фактических параметров списку формальных параметров выполняется <welse – part> , а при ее отсутствии – выдается диагностика об ошибке.

2.11. Операторы #act, #deact

<statement #**act**> ::= #**act** <macro name>
<statement #**deact**> ::= #**deact** <macro name>

Оператор #**deact** запрещает подстановку макросов с именем указанным в нем до следующего оператора #**act**. Этот оператор в основном используется для эмуляции требуемой стандартом ANSI семантики оператора #**define**: если задан ключ -n (запрещающий рекурсивную подстановку #**define**-макросов), то тело каждой макрокоманды, определенной при помощи оператора #**define**, обрамляется этими директивами.

2.12. Операторы #on, #off

<statement #**on**> ::= #**on**
<statement #**off**> ::= #**off**

Оператор #**off** запрещает выдачу генерируемого текста в выходной поток, оператор #**on** – возобновляет ее.

2.13. Оператор #hide

<statement #**hide**> ::= #**hide**

Этот оператор может использоваться только внутри макровызова. Его использование подавляет выдачу строки, относящейся к этому макросу в трассе ошибки. Его использование целесообразно в полностью отлаженных макросах, которые не могут быть причиной ошибки ни при каких значениях входных параметров.

Ключ -f отменяет действие всех операторов #**hide**.

3. Правила макроподстановки

Подстановка макроса производится следующим образом: в теле макроса все вхождения формальных параметров заменяются на фактические, если формальный параметр предварялся символом #, то он заключается в кавычки в соответствии с лексическими соглашениями языка *C*, параметр, отделенный от лексемы (или другого формального параметра) символами ##, конкатенируется с ним. После выполнения подстановки на место вызова макрокоманды (вместе с фактическими параметрами) подставляется результат подстановки.

В отличие от стандартного макропроцессора, в фактических параметрах макросов макроподстановка не производится. При обработке текста, предназначенного для обработки стандартным макропроцессором, это приводит к отличиям только в случае использования операций # и ## с такими параметрами. В остальных случаях эти подстановки выполняются при повторном просмотре результата макроподстановки. Для вынуждения немедленной подстановки в списке фактических параметров служит операция @.

3.1. Принудительная однократная подстановка макросов (@)

Символ @ в списке фактических параметров, за которым следует вызов макроса, обозначает: что необходимо немедленно подставить этот макрос. Сам символ @ при этом удаляется. При это следует иметь в виду, что при повторном сканировании текста подстановки производится будут только в том случае, если они также предваряются символом @.

3.2. Правила выбора перегруженных макросов

Когда в тексте встречается имя активного макроса, то производится поиск первого макро с типом скобки, таким же как открывающая скобка сразу за вызовом макроса. Если такого не нашлось (или если за именем макро идет не скобка), то производится поиск макроса без параметров.

Если макрос с соответствующим типом скобок был найден, то список фактических параметров разбирается в соответствии с разделителем этого макроса. После этого производится поиск макроса с данным именем, типом скобок, разделителем и числом параметров. Первый (в порядке, обратном порядку их определения) подошедший макрос вызывается.

3.3. Правила конкатенации строк

Если в результате применения оператора # к параметру макрокоманды полученная строка *непосредственно* соседствует с другой строкой, то производится удаление пары подряд стоящих кавычек и эти строки объединяются. Это не важно при использовании препроцессора с трансляторами с языка *C*, поскольку *C* сам "склеивает" такие строки, но может быть необходимо в других случаях (например при генерации "*.MD*" файла для *GCC*). Чтобы подавить этот эффект, достаточно вставить пробел между строкой и оператором *#id*.

Таким образом в результате трансляции

```
#with (x) (456)
        "123"#x"789"
        "123 – #x "789"
#end_with
```

будет получен текст:

```
"123456789"
"123 – "456 – "789"
```

4. Встроенные макросы

В макропроцессоре реализован набор встроенных макрокоманд, позволяющий выполнять достаточно сложные операции по преобразованию текстов.

4.1. Макрокоманда $\$cmp$

Макрокоманда $\$cmp\,(S1, S2)$ осуществляет сравнение своих аргументом. Результатом этого макровызова будет

- 0, если $S1 = S2$
- 1, если $S1 > S2$
- -1, если $S1 < S2$

4.2. Макрокоманда $\$len$

Результатом макрокоманды $\$len\,(arg)$ является число, равное длине ее аргумента в литерах.

4.3. Макрокоманда $\$upper$

Результатом макрокоманды $\$upper\,(arg)$ является строка arg, в которой все буквы нижнего регистра заменены на соответствующие буквы верхнего регистра,

4.4. Макрокоманда $\$lower$

Результатом макрокоманды $\$lower\,(arg)$ является строка arg, в которой все буквы верхнего регистра заменены на соответствующие буквы нижнего регистра,

4.5. Макрокоманда $\$find$

Данная макрокоманда имеет два аргумента. Результатом выполнения макровызова $\$find\,(str, pat)$ является номер позиции первого вхождения строки pat в строку str (нумерация позиций начинается с 0). Если pat не входит в str, то значением макроса $\$find$ является -1.

4.6. Макрокоманда $\$char$

Аргументом макрокоманды $\$char\,(code)$ является число. Результатом макрокоманды является литера, с кодом равным этому числу. Следует соблюдать осторожность в использовании этого макровызова: например появление в тексте управляющих символов, например литер с кодами 0 и 026 $(Ctrl/Z)$ приведет к труднопредсказуемым эффектам.

4.7. Макрокоманда $\$ascii$

Данная макрокоманда имеет вид $\$ascii\,(arg\left[, pos\right])$. Ее результатом является ASCII-код символа с номером pos из строки arg. Если pos опущен, то выдается код первого символа строки arg.

4.8. Макрокоманда $\$substr$

Макрокоманда $\$substr\,(str, pos, len)$ вырезает из аргумента str подстроку, начинающуюся с позиции pos (нумерация позиций начинается с 0) с длиной len. Например, результатом макровызова $\$substr\,(123456789, 3, 2)$ будет строка 45.

4.9. Макрокоманда $repeat

Макрокоманда $*repeat* имеет два аргумента. Ее результатом является второй аргумент, повторенный столько раз, сколько указано первым операндом. Например результатом макровызова $*repeat*$(3, 239, +)$ будет строка $239, +239, +239, +$

4.10. Макрокоманда $press

Макрокоманда $*press*(<arg>) удаляет из начала и конца своего аргумента все пробелы, табуляции и переводы строки. Эта макрокоманда очень часто необходима для обеспечения корректной работы макроопределений (особенно тех, которые допускают многострочные аргументы)

4.11. Макрокоманда $uniqid

данная макрокоманда при каждом своем вызове порождает новый идентификатор вида $\_\_ ID\_nnnn$, где $nnnn$ - уникальный номер из четырех символов.

4.12. Макрокоманда $eval

Макрокоманда $*eval* служит для вычисления значения своего аргумента. Она имеет единственный аргумент, который должен быть правильно построенным арифметическим выражением. Как правило она используется в препроцессорных циклах и других местах, где нежелательно образование слишком длинного выражения. Например, результатом макрорасширения цикла

#define *sum* 0
#for *i* $(1, 2, 3, 4, 5)$
    #set *sum* $(sum) + (i)$
#endf
sum

будет текст $(((((0) + (1)) + (2)) + (3)) + (4)) + (5)$, что, во-первых, громоздко, во-вторых, может быть неприемлемо либо по соображениям сложности (длина такого выражения в более сложной конструкции может расти экспоненциально), либо если полученный результат необходимо использовать именно как число (например, как суффикс идентификатора). С использованием функции $*eval* это может быть переписано следующим образом:

#define *sum* 0
#for *i* $(1, 2, 3, 4, 5)$
    #set *sum* $*eval*((sum) + (i))$
#endf
sum

и результатом будет 15. Отметим, что того же результата можно было добиться и использовав $*eval* в последней строке ($*eval*$(*sum*)), но это несколько хуже, так как промежуточный результат может иметь большую длину (впрочем, в данном примере это различие несущественно).

Также следует обратить внимание на то, что использование символа @ для вынуждения подстановки значения макро *sum* не требуется. В аргументе макро $*eval* макроподстановка производится автоматически [1].

[1] на самом деле эту макроподстановку делает процедура, которая реализует вычисление выражения, эта процедура является в точности той же, которая используется для вычисления выражений в операторах # **if** и #**while**

4.13. Макрокоманда $*write*

Макрокоманда $*write* (<fname>, <str>) создает новый файл с именем <fname> (оно должно быть заключено в кавычки) и записывает в него строку <str>.

4.14. Макрокоманда $*append*

Макрокоманда $*append* (<fname>, <str>) добавляет строку <str> к файлу с именем <fname> (и создает файл, если он не существует).

4.15. Макрокоманда $*hex*

Макрокоманда $*hex* (<expr>) вычисляет значение выражения <expr> при помощи функции $*eval* и возвращает его шестнадцатиричное представление. Префикс $0x$ не присоединяется и цифры $a - f$ представляются буквами нижнего регистра.

4.16. Макрокоманда $*HEX*

Макрокоманда $*HEX* (<expr>) тождественна предыдущей макрокоманде с тем отличием, что для представления числа используются буквы верхнего регистра.

4.17. Прочие макроопределения

- $\_\_STDC\_\_$ – значение данного макроса всегда равно 1
- $\_\_LINE\_\_$ – значением данного макроса является номер текущей строчки
- $\_\_FILE\_\_$ – значением данного макроса является имя текущего файла

5. Некоторые приемы использования макропроцессора:

5.1. Разбор сложных параметров

Довольно часто оказывается удобным реализовать более сложный синтаксис для параметров макро. Это обычно уменьшает число скобок и значительно повышает читаемость текста. Здесь мы приведем пример макроса, который позволяет записывать описания переменных в стиле языка *Pascal*:

```
#macro DclVar [dcls; ...];
        #for dcl #by ; (dcls)
                #with (ids : type) (dcl)
                        #for id (ids)
                                type id;
                        #endf
                #end_with
        #endf
#endm
```

В результате макровызова $DclVar\,[a, b, c : \underline{int};\ c1, c2 : \underline{char}]$; будет сгенерирован текст:

```
int     a;
int     b;
int     c;
char    c1;
char    c2;
```

5.2. Использование локальных макропеременных

Часто бывает необходимо использовать внутри макроса локальные переменные (например, для накопления результата арифметических вычислений. Теоретически можно описать переменную глобально и использовать ее, но при использовании рекурсивных макроопределений это может быть неприемлемо. Приводим пример макроса, который вычисляет сумму своих аргументов:

```
#macro sum (list, ...)
        #define result 0
        #for item (list)
                #set result $eval (result + item)
        #endf
        result
        #undef result
#endm
```

Следует обратить внимание на то, что # **undef** *result* стоит после использования значения result.

5.3. Макрокоманды с несколькими списками параметров

Довольно часто возникает желание описать макрокоманду, которая будет способна выбрать несколько подряд идущих списков параметров. Например, чтобы получить синтаксис наподобие оператора while языка *C*, где условие записывается в круглых скобках, а тело – в фигурных скобках сразу же после условия. К сожалению синтаксис <format parameters list> не позволяет это сделать напрямую. Однако есть способ добиться аналогичного эффекта. Для его демонстрации мы опишем макрос для организации цикла по всем элементам некоторого списка. Предполагается, что имя типа списка *List* [2]:

[2] При использовании транслятора GCC в данной ситуации очень удобно использовать оператор typeof)

```
#macro forAll (var = list_head)
        #macro __forAll {body; ...}
                #undef __forAll
                {
                        List * var = (list_head);
                        while (var ≠ NULL)
                        {
                                {body}
                                var = var → next;
                        }
                }
        #endm
        __forAll
#endm
```

Вызов этого макроса выглядит следующим образом:

```
int sum = 0;
forAll (p = list)
{
        sum += p → value;
}
```

в результате будет получен текст

```
int sum = 0
{
        List * p = (list);
        while (p ≠ NULL)
        {
                { sum += p → list; }
                p = p → next;
        }
}
```

Следует отметить использованный прием: определение локального макроса, который *начинает* свою работу с того, что отменяет свое определение (это корректно, так как в момент обработки оператора #**undef** тело макроса уже подставлено и дальше он не нужен.

5.4. Описание макросов, имитирующих блочную структуру

Способ, использованный для описания оператора forAll, приведенный выше имеет несколько существенных недостатков: Он связан с использованием длинного аргумента макрокоманды, что в сочетании с тем, что макропроцессор не способен выдавать точную позицию ошибки внутри параметров макрокоманд (он всегда указывает на начало параметра в исходном тексте) крайне затрудняет работу. Кроме этого, данный способ жестко фиксирует структуру оператора (например смоделировать подобным образом if_-then-else с необязательным оператором else уже затруднительно. Поэтому часто бывает удобен другой способ решения той же задачи:

```
#macro forAll (var = list_head)
    #with (break_label) (@$uniqid)
        {
            List * var = (list_head);
            while (var ≠ NULL)
            {
                {
        #macro break;
            goto break_label;
        #endm
        #macro endForAll
            #undef endForAll, break
                    }
                var = var → next;
                }
        break_label :;
            }
        #endm
    #end_with
#endm
```

Тогда его использование будет выглядеть так:

```
int sum = 0;
forAll (p = list)
    if (p → value < 0) break;
    sum += p → value;
endForAll
```

Следует отметить, что это макро будет нормально работать и при вложенных его использованиях. По поводу макрокоманды break следует отметить то, что этот макрос будет нормально работать с учетом возможной вложенности forAll (и даже в случае, когда он аналогичным образом определяется в других макросах того же типа), но при использовании внутри forAll обычного цикла языка C этот оператор перекроет возможность использования оператора break языка C.

5.5. Использование оператора #with для запоминания значений

Довольно типичным приемом при начале работы с макропроцессором является использование оператора #**define** для запоминания константных значений во всех случаях. Это довольно неудобно, так как переменные, определенные оператором # **define** являются макрокомандами и подчиняются соответствующим правилам.

Например они не будут автоматически подставляться в параметрах других макросов и т.п., невозможно их автоматическое использование в операторах # и ## и т.п.

По нашему опыту более предпочтительным способом для временного запоминания таких значений является использование оператора # **with** во всех случаях, когда это возможно:

```
#with (ARG) (@$upper (arg))
    int ARG;
    char ARG##_name[] = #ARG;
#end_with
```

получая в результате:

$$\underline{int}\ ARG;$$
$$\underline{char}\ ARG\_name[] = "ARG";$$

Отметим дополнительное достоинство использования оператора # **with** в данной ситуации: при использовании #**define** для конкатенации и взятия в кавычки пришлось бы использовать специальные макросы и оператор @:

$$\#\textbf{define}\ UARG\ @\$upper\,(arg)$$
$$\quad \underline{int}\ UARG;$$
$$\quad \underline{char}\ \$cat\,(@UARG, \_name) = \$quote\,(@UARG);$$
$$\#\textbf{undef}\ UARG$$

В этом примере предполагается, что макросы $\$cat$ и $\$quote$ описаны следующим образом:

$$\#\textbf{define}\ \$cat(x, y)\ x\#\#y$$
$$\#\textbf{define}\ \$quote(x)\ \#x$$

Следует также отметить, что оказалось необходимым переименовать параметр ARG в $UARG$, так как в противном случае при раскрытии строчки $\underline{int}\ ARG;$ возникла бы бесконечная рекурсия при подстановке макроса ARG (эта ошибка оказалась неожиданной даже для автора, когда он проверял этот пример). Разумеется, этот эффект возникает только в том случае, если не запрещены рекурсивные подстановки #**define**-макросов (ключ -n), но, тем не менее, это демонстрирует еще один потенциальный источник проблем, возникающих при использовании #**define**.

Действительно необходимым использование #**define** является при наличии надобности определить глобальные константы или в случаях, когда к этому значению будут применяться операторы # **setm** и #**set**.

Авторы не пришли к окончательному мнению следует ли использовать оператор # **with** также и для заключения в кавычки какого-либо текста. Единственным возражением против этого является некоторая громоздкость этой конструкции по сравнению с использованием специально описанной для этой цели макрокоманды (наподобие $\$cat$).

6. Работа с макропроцессором

6.1. Запуск макропроцессора

Запуск макропроцессора осуществляется командой

```
PPC <ключи> <имя входного файла>.cpp ...
```

6.2. Ключи макропроцессора

Ключи должны предваряться либо символом − либо символом /. В качестве ключей могут использоваться следующие последовательности:

- -$iPATH$ − пути поиска файлов, включаемых при помощи # **include**
- -$l0$ − не выводить директиву # **line**
- -$l1$ − выводить директивы # **line** с привязкой к вызову макро, породившей текст
- -$l2$ − выводить директивы # **line** с привязкой к строкам макро, породившей текст
- -n − запрет на использование рекурсивных # **define**'s
- -f − выводить при ошибках полную трассу (включая в нее вызовы "hidden" макро)
- -d<id>$\Big[= $<val>$\Big]$ − определить идентификатор <id> как имеющий значение <val>
- -r<file name> − задать корневой файл
- -o<file name pattern> − образец для имени выходного файла (по умолчанию – ".i")
- -?, -h − вывести краткую справку по ключам
- -a − при отсутствии файла, указанного в операторе # **include**, вместо выдачи диагностики об ошибке вывести этот оператор в выходной поток
- -m − ключ, предназначенный для обработки и генерации machine description file для компилятора GCC. При включении этого флага препроцессор начинает обрабатывать комментарии в стиле '.MD' файла: строки, начинающиеся с ';;' игнорируются. Комментарии в стиле $C/C++$ также остаются допустимыми.

6.2.1. Корневой файл

Если задан ключ -r<file name>, то макропроцессор начинает обрабатывать не тот файл, который был ему указан как входной файл, а файл, указанный в качестве значения ключа - r. При этом в стандартную переменную $\_\_MAINFILE\_\_$ заносится имя файла, подлежащего обработке. Как правило в корневом файле выполняются некоторые определения, после чего следует оператор **#include** $\_\_MAINFILE\_\_$, после которого иногда следуют какие-либо проверки корректного завершения основного файла.

6.2.2. Образец для имени выходного файла

Значением параметра -o в общем случае является произвольная спецификация файла, часть полей которой (путь, имя, тип) может отсутствовать. При работе опущенные части заменяются на соответствующие места из спецификации входного файла и полученное имя используется как имя выходного файла. Как правило задается либо только тип файла (например умолчание: ".i"), либо полная спецификация файла.

7. Несовместимости и известные ошибки

7.1. Несовместимости

Данный список не претендует на то, чтобы быть исчерпывающим описанием несовместимостей.

- Макровызовы в параметрах других макровызовов не раскрываются перед вызовом (но, естественно, раскрываются в теле макроса после его подстановки, если этому не препятствуют какие-либо причины (например взятие аргумента макроса в кавычки)
- При выборке очередного параметра макрокоманды учитываются все скобки ($[], (), \{\}$). При этом конструкция $(a[b,),c],e)$ рассматривается как один параметр (то есть первая закрывающая ")" игнорируется, так как заключена в скобки другого типа
- Макрооператоры (начинающиеся с #) могут начинаться с любой позиции строки (а не только с начала). При этом обычно предшествующий началу макрооператора символ перевода строки не рассматривается как часть макрокоманды и не удаляется при замене на тело макро.
- Определение макро при помощи # **define** не отменяет предыдущее определение того же макро, а помещает его в стек, последующий оператор # **undef** восстанавливает действие старого определения. Старое определение может также продолжать быть доступно в результате перегрузки (если его список параметров может быть отличен от списка параметров нового макро)
- При заданном ключе -n макросы, определенные при помощи оператора # **define** не могут быть вызваны рекурсивно

7.2. Ошибки, недоработки и неожиданные эффекты

7.2.1. Размер буфера

В настоящее время макропроцессор содержит только одно количественное ограничение: длина внутреннего текстового буфера (примерно равная суммарной длине тела макроса и его аргументов) не может превышать 8KB для версии real-mode MS-DOS и 64KB для всех прочих версий. На практике, помехой является только первое из двух ограничений. 64KB ограничение до сих пор превышалось только в результате ошибок (наиболее типичная - незакрытая скобка в списке параметров, другой типичный пример - бесконечная рекурсия при подстановке макроса).

7.2.2. Диагностика ошибок при подстановке параметров

При подстановке параметра макроса в нем не содержится детальное описание его привязки к исходному тексту, поэтому при ошибке, причина которой находится внутри параметра, в трассе вызовов будет указано место начала параметра.

7.2.3. Проблема при конкатенации аргументов, вложенных в различные макросы

При необходимости осуществить конкатенацию нескольких параметров, принадлежащих разным макрокомандам возникает проблема, связанная с тем, что при раскрытии тела внешнего макроса производится конкатенация, результат которой уже не является именем параметра вложенного макро. Пример: в результате обработки

```
#with (a) (A)
        #with (b) (B)
                a##b
        #end_with
#end_with
```

после раскрытия внешнего оператора # **with** будет получен следующий текст:

```
#with (b) (B)
      Ab
#end_with
```

А в результате его раскрытия будет получено Ab. Правильным вариантом в такой ситуации будет удвоение оператора ##:

```
#with (a) (A)
    #with (b) (B)
          a## ##b
    #end_with
#end_with
```

тогда после раскрытия внешнего оператора # **with** будет получено:

```
#with (b) (B)
      A##b
#end_with
```

и результатом его раскрытия будет AB.

7.2.4. Перевод строки в конце макроса

Результатом раскрытия конструкции

```
#macro M
      XXX
#endm
```

будет $XXX\backslash n$, так как $\backslash n$, предшествующий оператору # **endm** не рассматривается как его часть. В некоторых ситуациях это нежелательно. В такой ситуации следует писать # **endm** сразу после последней строки текста:

```
#macro M
     XXX#endm
```

Или, более удобочитаемая запись того-же самого:

```
#macro M
      XXX\
#endm
```