КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 09 — Директивы препроцессора

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.10.20

Оглавление

Препроцессор

Компилятор может *условным образом* обрабатывать либо пропускать обработку разделов исходных файлов, включать другие исходные файлы и выполнять замену макросов.

Эти возможности называются *предварительной обработкой* (*preprocessing*), потому что они выполняются до процесса, собственно, трансляции т.н. единицы трансляции, которая представляет собой результат процесса предварительной обработки.

Синтаксис

входной-файл-препроцессора:
    [группа]
группа:
    член-группы
    группа член-группы
член-группы:
    if-секция
    управляющая-строка
    строка-текста
    # не-директива

if-секция

if-секция:
    if-группа [elif-группы] [else-группа] endif-строка
if-группа:
    # if *константное-выражение EOL [группа]*
    # ifdef *идентификатор EOL [группа]*
    # ifndef *идентификатор EOL [группа]*
elif-группы:
    elif-группа
    elif-группы elif-группа
elif-группа:
    # elif *константное-выражение EOL [группа]*
else-группа:
    # else *EOL [группа]*
endif-строка:
    # endif *EOL*

Управляющая-строка

управляющая-строка:

 # include *pp-лексемы EOL*
 # define *id список-замен EOL*
 # define *id lpar [id-список]* **)** *список-замен EOL*
 # define *id lpar* **...** **)** *список-замен EOL*
 # define *id lpar id-список* **,** **...** **)** *список-замен EOL*
 # undef *id EOL*
 # line *pp-лексемы EOL*
 # error *[pp-лексемы] EOL*
 # pragma *[pp-лексемы] EOL*
*EOL*

id:

идентификатор

lpar:

(-символ, перед которым нет пробела

список-замен:

[pp-лексемы]

pp-лексемы:

лексема-препроцессора
pp- лексемы лексема-препроцессора

EOL:

конец строки

Две категории лексем — лексемы языка и лексемы препроцессора

Лексемы языка (token)

- ключевое (зарезервированное) слово (keyword);
- идентификатор (identifier);
- константа (constant);
- строковый литерал (string-literal);
- разделитель (punctuator); // скобки, точка, запятая, пробельные символы, ...

Лексемы препроцессора (preprocessing-token)

- имя заголовка (header-name);
- идентификатор (identifier);
- числа препроцессора (pp-number);
- символьная константа (character-constant);
- строковый литерал (string-literal);
- разделитель (punctuator);
- одиночные непробельные символы, которые не совпадают лексически с другими категориями лексем препроцессора[1].

В процессе препроцессирования некоторые лексемы препроцессора преобразуются в лексемы языка.

1 Дополнительная категория, метки, используются внутри на этапе 4 трансляции и не встречается в исходных файлах.

Директива препроцессора состоит из последовательности pp-лексем, удовлетворяющей следующим ограничениям:

1) первая лексема в последовательности является pp- лексемой **#**, которая является либо первым символом в исходном файле, либо перед ним может быть пробельный материал, не содержащий символов новой строки, либо следует за пробельным промежутком, содержащим хотя бы один символ новой строки (пустые строки).

2) последняя лексема в последовательности является первым символом новой строки, который появляется после первой лексемы в последовательности. Поэтому директивы препроцессора обычно называются «строками».

3) символ новой строки в любом случае завершает директиву препроцессора.

4) никакая другая строка кроме директив препроцессора не может начинаться с лексемы **#**.

5) единственными пробельными символами, которые могут появляться между pp-лексемами в директиве (между **#** и завершающим символом новой строки), являются пробел и горизонтальная табуляция (включая пробелы, которыми были заменены комментарии или, возможно, другие пробельные символы).

6) pp-лексемы, встречающиеся в директиве препроцессора, не подлежат макрорасширению.

Пример

```
#define EMPTY
EMPTY # include <file.h>
```

последовательность pp-лексем во второй строке не является директивой препроцессора, поскольку она не начинается с символа **#**, даже если это и будет иметь место после замены макроса **EMPTY**.

Ограничения

Каждая *лексема препроцессора*, преобразованная в *лексему языка*, должна иметь лексическую форму ключевого слова, идентификатора, константы, строкового литерала или разделителя.

Это значит, что можно определить в качестве **{** и **}** что-нибудь другое, например, **BEGIN** и **END**, как это в паскале и прочей Модуле:

```
#include <stdio.h>
int main(void) {

  int a = 123;
  printf("a(in file scope): %i\n", a);

#define BEGIN {
#define END   }

  BEGIN                                 // {
    int a = 999;                        //   int a = 999;
    printf("a(in BEGIN-END): %i\n", a); //   printf("a(in BEG-END): %i\n",
a);
  END                                   // }

}
$ gcc -o test -W -Wall test.c
$ ./test
a(in file scope): 123
a(in BEGIN-END) : 999
```

Лексемы препроцессора могут быть разделены пробелами, которые состоят из комментариев (`/*... */`, `//`) или символов пробела (пробел, символ горизонтальной табуляции, символ новой строки, символ вертикальной табуляции и символ перевода формата), или из того и другого.

Пробелы могут появляться в лексеме препроцессора только как часть имени заголовка или между символами кавычек в символьной константе или строковом литерале.

Условное включение

Директивы препроцессора в форме

 # if *константное-выражение EOL [группа]*
 # elif *константное-выражение EOL [группа]*

проверяют, является ли управляющее *константное-выражение* ненулевым. Выражения, управляющее условным включением, должны быть целочисленными константными выражениями, а также могут содержать унарные операторные выражения в форме:

defined *идентификатор*

или

defined (*идентификатор* **)**

Эти операторные выражения вычисляются как 1, если *идентификатор* в данный момент определен как имя макроса (то есть, либо он предопределен, либо является объектом директивы препроцессора **#define** без промежуточной директивы **#undef** с тем же идентификатором), или как 0, если он таковым не является.

Поскольку управляющее константное выражение вычисляется до фазы, собственно, трансляции, все идентификаторы либо являются именами макросов, либо ими не являются – просто еще нет ни ключевых слов, ни констант перечислений и прочего.

Перед вычислением выражения, вызовы макросов в списке pp-лексем, которые станут управляющим константным выражением, заменяются, как в обычном тексте.

Если в результате такого процесса замены генерируется лексема **defined** или использование унарного оператора **defined** не соответствует ни одной из двух вышеуказанных форм до замены макроса, поведение не определено.

Сформированные таким образом лексемы составляют управляющее константное выражение, которое вычисляется по правилам вычисления выражений в языке C.

В процессе данного преобразования и вычисления все целочисленные типы со знаком и все целочисленные типы без знака действуют так, как если бы они имели представление, идентичное типам **intmax_t** и **uintmax_t**, определенные в заголовке **<stdint.h>**.

Директивы препроцессора в форме

 # ifdef *идентификатор EOL [группа]*
 # ifndef *идентификатор EOL [группа]*

проверяют, определен ли *идентификатор* в данный момент как имя макроса.

Их условия эквивалентны **#if defined** *идентификатор* и **#if !defined** *идентификатор*, соответственно.

Проверяется условие каждой директивы.

Если значение равно **false** (ноль), группа, которой она управляет, пропускается.

Обрабатывается только первая группа, чье управляющее условие вычисляется как истинное (ненулевое).

Если ни одно из условий не получает значения **true** и существует директива **#else**, обрабатывается группа, контролируемая **#else**.

При отсутствии директивы **#else** все группы до **#endif** пропускаются.

В любом месте исходного файла могут появляться комментарии, в том числе в директиве препроцессора.

Включение исходных файлов

Директива препроцессора в форме

```
# include <h-последовательность-символов> EOL
```

ищет в последовательности мест, определенных реализацией, *заголовок*, уникально идентифицируемый последовательностью, указанной между разделителями **<** и **>**, и выполняет замену данной директивы всем содержимым заголовка.

Способы указания мест поиска, и/или что собой представляет заголовок, определяется реализацией.

Директива препроцессора в форме

```
# include "q-последовательность-символов" EOL
```

заставляет выполнить замену данной директивы всем содержимым *исходного файла*, идентифицируемого указанной последовательностью между **"**-разделителями.

Названный исходный файл ищется способом, определяемым реализацией.

Если данный способ поиска не поддерживается или если поиск не удался, директива обрабатывается так, как будто она читается

```
# include <h-последовательность-символов> EOL
```

с последовательностью символов, содержащейся между **"** (включая символы **>**, если таковые имеются), из исходной директивы.

Директива препроцессора в форме

```
# include pp-лексемы EOL
```

которая не соответствует одной из двух предыдущих форм, допускается.

Лексемы после включения в директиву обрабатываются так же, как и в обычном тексте – каждый идентификатор, определенный в данный момент, как имя макроса, заменяется своим списком замены pp-лексем.

Директива, получающаяся после всех замен, должна соответствовать одной из двух предыдущих форм.

Метод, с помощью которого последовательность лексем, находящихся между `<` и `>`, или парой символов `"` объединяются в одно имя заголовка, определяется реализацией.

Директива препроцессора **#include** может появиться в исходном файле, который был прочитан вследствие обработки другой директивы **#include** из другого файла, вплоть до максимального предела вложенности, определенного реализацией.

Пример 1

Наиболее распространенные варианты использования директив препроцессора **#include**:

```
#include <stdio.h>
#include "myprog.h"
```

Пример 2 иллюстрирует макрозамены директивы **#include**:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
// and so on
#else
#define INCFILE "versN.h"
#endif
#include INCFILE
```

Макрозамены

 # define *id* *список-замен EOL*
 # define *id* *lpar [id-список]* **)** *список-замен EOL*
 # define *id* *lpar* **...** **)** *список-замен EOL*
 # define *id* *lpar id-список* **,** **...** **)** *список-замен EOL*

Идентификатор, следующий сразу за **define**, называется *макроименем* (именем макроса).

Для имен макросов существует одно пространство имен.

Любые пробельные символы, предшествующие или следующие за списком замены pp-лексем, не считаются частью списка замены ни для одной из форм макросов.

Если pp-лексема **#**, за которым следует идентификатор, встречается в точке, в которой лексически может начинаться директива препроцессора, идентификатор замене макросами не подлежит.

Объектоподобный макрос

Директива препроцессора в форме

```
# define идентификатор список-замен EOL
```

определяет *объектоподобный макрос*, который вызывает замену каждого последующего экземпляра макроимени[2] *идентификатор* списком замены pp-лексем, из которых состоит оставшаяся часть директивы.

Список замены затем повторно (рекурсивно) сканируется для получения дополнительных имен макросов.

Идентичность списков замены — два списка замены идентичны в том и только в том случае, если pp-лексемы в обоих имеют одинаковое число, порядок, орфографию и шпаций (пробельных промежутков), при этом все шпации считаются идентичными.

Идентификатор, определенный в данный момент как объектоподобный макрос может переопределяться другой директивой препроцессора **#define**, если второе определение является определением объектоподобного макроса и два списка замены идентичны.

Между идентификатором и списком замены в определении объектоподобного макроса должен быть пробел.

---

2 Все символьные константы и строковые литералы никогда не сканируются на наличие имен или параметров макросов, поскольку к моменту макрозамены они являются pp-лексемами, подобные идентификаторам, а не последовательностями, возможно содержащими подпоследовательности.

Функционально-подобный макрос

Директива препроцессора в форме

 # define *id lpar [список-идентификаторов]* **)** *список-замен EOL*
 # define *id lpar* **...** **)** *список-замен EOL*
 # define *id lpar список-идентификаторов* **,** **...** **)** *список-замен EOL*

определяет ***функционально-подобный макрос*** с параметрами, использование которого синтаксически аналогично вызову функции.

Параметры задаются необязательным *списком идентификаторов*, область действия которого простирается от их объявления в списке идентификаторов до символа новой строки, который завершает директиву препроцессора **#define**.

За каждым последующим экземпляром функционально-подобного имени макроса в качестве следующей pp-лексемы следует символ **(**, за которым следует последовательность pp-лексем, которая заменяется списком замены из определения (вызов макроса).

Заменяемая последовательность pp-лексем завершается согласованной pp-лексемой **)**.

Если в списке идентификаторов в определении макроса есть **...**, то конечные аргументы, включая любые разделительные pp-лексемы «запятая», объединяются в один элемент – переменные аргументы. Количество аргументов, объединенных таким образом, таково, что после слияния число аргументов становится на один больше, чем количество параметров в определении макроса (исключая **...**).

Идентификатор, в данный момент определяемый как функционально-подобный макрос, может быть переопределен другой директивой препроцессора **#define**, если второе определение является функционально-подобным макроопределением, которое имеет одинаковое число и написание параметров, при этом два списка замены должны быть идентичны.

Если список идентификаторов в определении макроса не заканчивается многоточием, число аргументов (включая те аргументы, которые не содержат pp-лексемы) при вызове функционально-подобного макроса должно равняться числу параметров в определении макроса. В противном случае в вызове должно быть больше аргументов, чем параметров в определении макроса (исключая **...**).

Также должна существовать pp-лексема **)**, завершающая вызов.

Подстановка аргументов

После того, как аргументы для вызова функционально-подобного макроса идентифицированы, происходит подстановка аргументов.

Операторы директивы #define

Оператор преобразования в строку **#**

В результате его выполнения соответствующий аргумент заключается в двойные кавычки.

Оператор преобразует *параметры макроса* в строковые литералы без расширения определения параметра. Он используется только с макросами, принимающими аргументы.

Если он стоит перед формальным параметром в определении макроса, то фактический аргумент, передаваемый вызовом макроса, заключается в кавычки и обрабатывается как строковый литерал. Далее этим строковым литералом заменяется каждое сочетание строкового оператора с формальным параметром, которое встречается в определении макроса.

```
#define mkstr(a) # a
mkstr(#include <stdio.h>) --> "#include <stdio.h>"
```

Оператор вставки лексемы **##**

Оператор маркера[3] (**##**) иногда называют оператором слияния или объединения — он выполняет конкатенацию лексем, используемых в качестве фактических аргументов, для создания других лексем.

Может использоваться как в объектном, так и в функциональном макросах. Поскольку предназначен объединять отдельные лексемы в одну, поэтому не может быть первым или последним маркером в определении макроса.

Если перед формальным параметром в определении макроса или после него находится оператор **##**, этот формальный параметр сразу же заменяется фактическим аргументом, который не разворачивается. Результирующая лексема должна быть допустимой.

Сгенерированная в результате лексема сканируется далее на предмет возможной замены (если она представляет имя макроса).

3 placeholder

Пример

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y";
```

Макрорасширение на разных этапах производит:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

Другими словами, расширение **hash_hash** создает новую лексему, состоящий из двух смежных знаков **#**, но эта новая лексема не является оператором **##**.

Пример

Есть опасные случаи, когда неясно, является ли замена вложенной или нет.

Для примера даны следующие макроопределения:

```
#define f(a) a*g
#define g(a) f(a)
```

вызов

```
f(2)(9)
```

может расширяться либо в

```
2*f(9)
```

либо в

```
2*9*g
```

Программы, строго соответствующие стандарту, не ДОЛЖНЫ использовать такие конструкции.

Область действия макроопределений

Определение макроса длится (независимо от блочной структуры) до тех пор, пока не встретится соответствующая директива **#undef**.

Если **#undef** не встретится, определение макроса длится до конца модуля.

Директива препроцессора в форме

```
# undef идентификатор EOL
```

приводит к тому, что указанный идентификатор больше не является определенным как имя макроса и игнорируется.

Пример 1

Самое простое использование **#define** — определить «явную константу»

```
#define TABSIZE 100
int table[TABSIZE];
```

Пример 2 определяет функционально-подобный макрос, значение которого является максимальным из его аргументов.

Он может работать с любыми совместимыми типами аргументов и генерировать встроенный код без дополнительных затрат на вызов функции.

У него есть недостатки — он генерирует больше кода, чем функция, если вызывается несколько раз. Также невозможно получить его адрес, так как у него его нет.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Скобки гарантируют, что аргументы и полученное выражение будут сцеплены правильно.

Пример 3 Иллюстрация правил переопределения и пересмотра

```
#define x          3
#define f(a)       f(x * (a))
#undef  x
#define x          2
#define g          f
#define z          z[0]
#define h          g(~
#define m(a)       a(w)
#define w          0,1
#define t(a)       a
#define p()        int
#define q(x)       x
#define r(x,y)     x ## y
#define str(x)     # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

В результате будет сгенерировано следующее

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

Пример 4

Иллюстрация правил создания символьных строковых литералов и конкатенации лексем

```
#define str(s)        # s
#define xstr(s)       str(s)
#define debug(s, t)   printf("x" # s "= %d, x" # t "= %s", \
                                 x ## s, x ## t)
#define INCFILE(n)    vers ## n
#define glue(a, b)    a ## b
#define xglue(a, b)   glue(a, b)
#define HIGHLOW       "hello"
#define LOW           LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

приведет к следующему

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",s);
#include "vers2.h"                                  (после замены макроса)
"hello";
"hello" ", world"
```

или, после объединения символьных строковых литералов,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers2.h"                                    (после замены макроса)
"hello";
"hello, world"
```

Пробелы вокруг **#** и **##** в макроопределениях не являются обязательными

Пример 5 — Иллюстрирация применения правил для маркеров меток

```
#define t(x,y,z) x ## y ## z
int j[] = {
    t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
    t(10,,), t(,11,), t(,,12), t(,,)
};
```

после расширения получим

```
int j[] = { 123, 45, 67, 89, 10, 11, 12, };
```

Управление строками

Номер текущей исходной строки равен количеству прочитанных символов новой строки, встретившихся при обработке исходного файла, плюс единица.

Директива препроцессора в форме

```
# line числовая-последовательность EOL
```

заставляет компилятор вести себя так, как будто следующая последовательность исходных строк начинается со строки, которая имеет номер, заданный последовательностью цифр (интерпретируется как десятичное целое число). Последовательность цифр не должна быть ни равна нулю, ни быть числом больше 2147483647.

Директива препроцессора в форме

```
# line числовая-последовательность " [s-последовательность] " EOL
```

устанавливает предполагаемый номер строки и изменяет предполагаемое имя исходного файла на содержимое строкового литерала символов.

Директива препроцессора в форме

```
# line pp-лексемы EOL
```

которая не соответствует ни одной из двух предыдущих форм, *допускается*.

В этом случае pp-лексемы после **`line`** в директиве обрабатываются так же, как и в обычном тексте (каждый идентификатор, определенный в данный момент как имя макроса, заменяется своим списком замены pp-лексем).

Директива, получающаяся после всех замен, должна соответствовать одной из двух предыдущих форм.

Директива error

Директива препроцессора в форме

```
# error [pp-лексемы] EOL
```

приводит к выдаче диагностического сообщения, включающего указанную последовательность pp-лексем.

Директива pragma

Директива препроцессора в форме

```
# pragma [pp-лексемы] EOL
```

где pp-лексема, следущая сразу же за **pragma** в директиве не является **STDC**, (до того, как будет выполнена замена макроса) заставляет компилятор вести себя определенным для данной конкретной реализации образом.

Любая такая **pragma**, которая не распознается компилятором, игнорируется.

Если за **pragma** (до того, как будет выполнена замена макроса) следует pp-лексема **STDC**, то макрозамена не выполняется, а директива должна иметь одну из следующих форм:

#pragma STDC FP_CONTRACT *on-off-switch*
#pragma STDC FENV_ACCESS *on-off-switch*
#pragma STDC CX_LIMITED_RANGE on-off-switch

on-off-switch: one of **ON OFF DEFAULT**

Пустая директива

Директива препроцессора в форме

*новая-строка*

эффекта не имеет

Предопределенные макроимена

Обязательные макросы

__DATE__ — дата трансляции (препроцессинга) файла в форме «**Mmm dd yyyy**», где названия месяцев совпадают с названиями месяцев, сгенерированными функцией **asctime**, а первый символ **dd** — символ пробела, если значение меньше **10**.

Если дата трансляции недоступна, будет указана некоторая другая действительная дата, определенная реализацией.

__TIME__ — время трансляции (препроцессинга) — строковый литерал в форме «**hh:mm:ss**», как оно генерируется функцией **asctime**. Если время трансляции недоступно, будет указана некоторая другая действительная дата, определенная реализацией.

__FILE__ — предполагаемое имя текущего исходного файла (символьная строка-литерал).

__LINE__ — предполагаемый номер (в текущем исходном файле) текущей исходной строки (целочисленная константа).

__STDC__ — целочисленная константа, равная 1, предназначенная для указания совместимшой реализации.

__STDC_HOSTED__ — целочисленная константа, равная 1, если реализация является реализацией, размещенной в системе, или целочисленная константа, равная 0, если это не так.

__STDC_VERSION__ — целочисленная константа **201ymmL**.

Значения предопределенных макросов за исключением **__FILE__** и **__LINE__**), остаются неизменными на протяжении всей единицы трансляции.

Ни одно из этих макроимен, ни определенный идентификатор, не должны использоваться в директивах **#define** или **#undef**.

Любые другие предопределенные макроимена начинаются с начального подчеркивания, за которым следует заглавная буква или второе подчеркивание.