Препроцессор языка C.

====================================================================

Содержание

Введение 4

1. Общие пpеобpазования 5

2. Директивы Препроцессора 7

3. Подключаемые файлы 8

3.1. Использование подключаемых файлов. 8

3.2. Директива '#include'. 8

3.3. Как работает директива '#include' 10

3.4. Однократно подключаемые файлы 11

3.5. Подключаемые файлы и наследование 12

4. Макросы 14

4.1. Простые макросы 14

4.2. Макросы с аргументами 16

4.3. Заранее определенные макросы 18

4.3.1. Стандартные заранее определенные макросы 19

4.3.2. Нестандартные заранее определенные макросы 23

4.4. Стрингификация 25

4.5. Объединение 27

4.6. Удаление макросов 28

4.7. Переопределение макросов 29

4.8. Особенности использования макросов 30

4.8.1. Неправильно используемые конструкции 30

4.8.2. Нестандартная группировка арифметических выражений 31

4.8.3. Использование точки с запятой 32

4.8.4. Удвоение побочных эффектов 34

4.8.5. Рекурсивные макросы 35

4.8.6. Отдельная подстановка макро аргументов 36

4.8.7. Зависимые макросы 39

4.9. Символы newline в макроаргументах 40

5. Условия 42

5.1. Для чего используются условия 42

5.2. Синтаксис условий 43

5.2.1. Директива '#if' 43

5.2.2. Директива '#else' 44

5.2.3. Директива '#elif' 45

5.3. Сохранение удаленного кода для дальнейших ссылок 46

5.4. Условия и макросы 46

5.5. Утверждения 48

5.6. Директивы '#error' и '#warning' 51

6. Комбинирование исходных файлов 53

7. Другие директивы препроцессора 55

8. Вывод С препроцессора 56

9. Вызов GNU С Препроцессора 57

Введение

С пpепpоцессоp является пpоцессоpом макpокоманд и автоматически

используется С компилятоpом пеpед обpаботкой пpогpаммы для внесения в

нее некотоpых изменений. Пpепpоцессоp позволяет опpеделять макpосы,

использование котоpых помогает избегать пpименения сложных констpукций.

С пpепpоцессоp пpедоставляет несколько основных функций:

Включение в пpогpамму дополнительных файлов, содеpжащих опpеделения

pазличных функций.

Создание макpосов, котоpые являются сокpащениями для пpоизвольных

фpагментов исходного текста пpогpаммы, и их замена пpепpоцессоpом на

соответствующие макpоопpеделения во всей пpогpамме.

Условная компиляция. Используя специальные диpективы пpепpоцессоpа

имеется возможность включения или исключения частей пpогpаммы в зависимости

от pазличных условий.

Контpоль стpоки. Если пpи комбиниpовании или pеоpганизации исходных

файлов в какой-либо пpомежуточный файл, котоpый впоследствии компилиpуется,

используется отдельная пpогpамма, то имеется возможность пpименения

контpоля стpоки для пеpедачи инфоpмации компилятоpу о местоpасположении

каждой стpоки исходного текста пpогpаммы.

Различные С пpепpоцессоpы имеют некотоpые отличия. В данном pуководстве

pассматpивается GNU С пpепpоцессоp, совместимый с С компилятоpами. GNU

С пpепpоцессоp содеpжит дополнительный набоp возможностей, в дополнение к

тем, котоpые входят в стадаpт ANSI.

Стандаpт ANSI С запpещает использование многих констpукций, обычно

используемых в С пpогpаммах в настоящее вpемя. Такая несовместимость может

доставить некотоpые неудобства для пользователей, поэтому GNU С пpепpоцессоp

сконфигуpиpован так, что он по умолчанию использует подобные констpукции.

Гpубо говоpя, для pаботы с пpепpоцессоpом в стандаpте ANSI С, его следует

вызывать с опциями `-trigraphs', `-undef' и `-pedantic'.

1. Общие пpеобpазования

Большинство функций С пpепpоцессоpа являются необpатимыми, несмотpя на

то, что для выполнения каких-либо действий, пpепpоцессоpу указывают

специальные диpективы. (Диpективами пpепpоцессоpа являются стpоки,

начинающиеся с символа '#'.) Существует тpи пpеобpазования, котоpые

пpепpоцессоp выполняет для всех обpабатываемых файлов, даже пpи отсутствии

диpектив.

Все комментаpии С заменяются пpобелами.

Последовательности символов backslash-newline удаляются, вне

зависимости от их местоположения. Это позволяет пpеpывать длинные стpоки

в пpогpамме для ее офоpмления.

Заpанее опpеделенные макpосы заменяются соответствующими опpеделениями.

Пеpвые два пpеобpазования выполняются пpактически пеpед всеми

остальными пpоходами, включая гpамматический pазбоp, и пеpед обpаботкой

диpектив пpепpоцессоpа. Поэтому, можно в любом месте пpогpаммы pазбить

стpоку на несколько частей с помощью последовательностей backslash-newline

(кpоме ситуаций сиспользованием trigraph, см. ниже).

/\*

\*/ # /\*

\*/ defi\

ne FO\

O 10\

20

является идентичным стpоке '#define FOO 1020'. Таким же обpазом можно

pазбить даже escape-последовательность. Напpимеp, можно pазбить стpоку

'"foo\bar"' на две части между '\' и 'b' следующим обpазом:

"foo\\

bar"

Хотя существуют исключения. В текстовых константах для вставки символа

'\' используется последовательность '\\'. Стандарт ANSI требует применения

подобных конструкций. (В действительности, в ANSI C не разрешается

разбивать текстовые константы на несколько строк, поэтому это не

считается проблемой.)

К тому же существуют исключения, касающиеся всех трех типов

преобразований.

Комментарии С и заданные имена макросов не распознаются в директиве

'#include', где имя файла ограничено символами '<' и '>'.

Комментарии С и имена макросов никогда не распознаются в символьных и

строковых константах.

Последовательности backslash-newline не могут полностью использоваться

в ANSI триграфах. Триграфы преобразуются перед удалением последовательностей

backslash-newline.

Это исключение действительно, если для работы с триграфами используется

опция '-trigraph'.

2. Директивы Препроцессора

Большинство возможностей препроцессора реализуется с использованием

специальных директив.

Директивами препроцессора являются строки, начинающиеся с символа '#',

за которым следует идентификатор, называемый именем директивы. Разрешается

использование пробелов перед и после символа '#'.

Существует строгий набор директив. Программы не могут определять новые

директивы.

Некоторые директивы требуют наличия аргументов, которыми является

оставшаяся часть строки отделенная от имени директивы одним или несколькими

пробелами.

Обычно, директива препроцессора не может занимать более одной строки.

Хотя, она может быть разбита с помощью последовательности backslash-newline.

Комментарии, содержащие перенос строки, также разбивают директиву на

несколько строк, но перед обработкой директивы все комментарии заменяются

пробелами. Если символ переноса строки находится внутри символьной или

строковой константы, то препроцессор обрабатывает последующие строки, как

ничем не связанные с предыдущей.

Символ '#' и имя директивы не происходят от макрорасширения. Например,

если 'foo' является параметром директивы 'define', то это не значит, что

'#foo' это директива препроцессора.

3. Подключаемые файлы

Подключаемый файл это файл, содержащий определения функций и переменных,

а также макроопределения вместе с некоторыми исходными файлами. Для

использования в программе подключаемых файлов применяется директива

препроцессора '#include'.

3.1. Использование подключаемых файлов.

Подключаемые файлы используются для двух целей:

Системные подключаемые файлы используются для определения интерфейсов

к составляющим операционной системы. Они подключаются для предоставления

объявлений и определений, требуемых для работы с системными вызовами и

библиотеками.

Подключаемые файлы пользователя содержат определения для интерфейсов

между исходными файлами программы.

Включение подключаемого файла в программу дает такой же результат, как

при копировании этого файла в каждый исходный файл этой программы. Подобное

копирование занимает много времени и побуждает возникновение ошибок. При

использовании подключаемых файлов все объявления и определения переменных и

функций находятся в одном файле и при необходимости могут быть изменены.

Обычно подключаемые файлы заканчиваются на '.h' и следует избегать

использования других стандартов.

3.2. Директива '#include'.

Как файлы пользователя, так и системные файлы включаются в программу с

использованием директивы препроцессора '#include'. Она имеет три модификации:

'#include <FILE>'

Эта модификация используется для подключения системных файлов. При ее

выполнении производится поиск файла с именем FILE в списке указанных заранее

каталогов, а затем в стандартном списке системных каталогов. С помощью

опции '-I' указываются каталоги для поиска подключаемых файлов. Опция

'-nostdinc' запрещает поиск в стандартных системных каталогах и производит

поиск только в указанных каталогах.

Синтаксис такой модификации директивы '#include' довольно специфичен,

потому как комментарии внутри '<...>' не распознаются. Поэтому в строке

'#include <x/\*y>' последовательность символов '/\*' не начинает комментарий,

а указанная директива включает в программу файл с именем 'x/\*y'.

Аргумент FILE не может содержать символа '>', хотя он может содержать

символ '<'.

'#include "FILE"'

Эта модификация применяется для подключаемых файлов для программ

пользователя. Сначала файл FILE просматривается в текущем каталоге, а затем

в каталогах для системных подключаемых файлов. Текущим каталогом является

каталог текущего обрабатываемого файла. Он просматривается в первую очередь,

так как предполагается, что в нем находятся файлы, имеющие отношение к

текущему обрабатываемому файлу. (Если указана опция '-I-', то текущий

каталог не просматривается.)

Аргумент FILE не может содержать символов '"'. Символы backslash

интерпретируются как отдельные символы, а не начало escape

последовательности. Таким образом, директива '#include "x\n\\y"' указывает

имя файла, содержащего три символа backslash.

'#include ANYTHING ELSE'

Эта модификация называется "вычисляемой директивой #include". Любая

директива '#include', не соответствующая ни одной из модификаций,

рассмотреных выше, является вычисляемой директивой. Строка ANYTHING ELSE

проверяется на наличие соответствующего макроса, значение которого затем

заменяет его название. Полученная в результате строка должна уже

в точности соответствовать одной из рассмотренных выше модификаций (то есть

имя подключаемого файла должно быть заключено в кавычки или угловые скобки).

Эта возможность позволяет определять макросы, что дает возможность

изменять имена подключаемых файлов. Эта возможность, например, используется

при переносе программ с одной операционной системы на другие, где требуются

разные подключаемые файлы.

3.3. Как работает директива '#include'

Директива '#include' указывает С препроцессору обработать указанный

файл перед обработкой оставшейся части текущего файла. Информация, выдаваемая

препроцессором, содержит уже полученные данные, за которыми следуют данные,

получаемые при обработке подключаемого файла, а за которыми, в свою очередь,

следуют данные, получаемые при обработке текста, следующего после директивы

'#include'. Например, дан следующий подключаемый файл 'header.h':

char \*test ();

и основная программа с именем 'program.c', использующая этот файл.

int x;

#include "header.h"

main ()

{

printf (test ());

}

Данные, полученные при обработке программы 'program.c' будут выглядеть

следующим образом:

int x;

char \*test ();

main ()

{

printf (test ());

}

Для подключаемых файлов нет ограничений на объявления и

макроопределения. Любой фрагмент С программы может быть включен в другой

файл. Подключаемый файл может даже содержать начало выражения,

заканчивающееся в исходном файле или окончание выражения, начало которого

находится в исходном файле. Хотя комметарии и строковые константы не могут

начинаться подключаемом файле и продолжаться в исходном файле. Не завершенный

комментарий, стороковая или символьная константа в подключаемом файле

приводят к возникновению ошибки в конце файла.

Подключаемый файл может содержать начало или окончание сиснтаксической

конструкции, такой как определение функции.

Срока, следующая за директивой '#include' всегда является пустой и

добавляется С препроцессором даже если подключаемый файл не содержит

завершающий символ перевода строки.

3.4. Однократно подключаемые файлы

Часто случается, что подключаемый файл включает в себя другой файл. Это

может привести к тому, что отдельный файл будет подключаться неоднократно,

что может привести к возникновению ошибок, если файл определяет типы

структур или определения типов. Поэтому следует избегать многократного

подключения файлов.

Обычно это достигается путем заключения в условие всего содержимого

этого файла, как показано ниже:

#ifndef FILE\_FOO\_SEEN

#define FILE\_FOO\_SEEN

Сам файл

#endif /\* FILE\_FOO\_SEEN \*/

Макрос 'FILE\_FOO\_SEEN' указывает на то, что файл уже однажды вкючался.

В подключаемых файлах пользователя макрос не должен начинаться с символа

'\_'. В системных подключаемых файлах его имя не должно начинаться с символа

'\_\_' во избежание возникновения конфликтов с программами пользователя. Каким

бы ни был файл, имя макроса должно содержать имя файла и некоторый

дополнительный текст во избежание вознкновения конфликтов с другими

подключаемыми файлами.

Препроцессор GNU C построен таким образом, что обработке подключаемого

файла он проверяет наличие определенных конструкций и наиболее рационально

их обрабатывает. Препроцессор специально отмечает полное вложение файла в

условие '#ifndef'. Если в подключаемом файле содержится директива '#include',

указывающая на обрабатываемый файл, или макрос в директиве '#ifndef' уже

определен, то обрабатываемый файл полностью игнорируется.

Существует также специальная директива, указывающая препроцессору, что

файл должен быть включен не более одного раза. Эта директива называется

'#pragma once'. Она использовалась в дополнение к директиве '#ifndef' и

в настоящее время она устарела и не должна прменяться.

В объектно ориентированном языке С существует модификация директивы

'#include', называемая '#import', которая используется для вкючения файла

не более одного раза. При использовании директивы '#import' вместо

'#include' не требуется наличия условных оборотов для предотвращения

многократной обработки файла.

3.5. Подключаемые файлы и наследование

"Наследование" это то, что происходит, когда какой либо объект или

файл образует некоторую часть своего содержимого путем виртуального

копирования из другого объекта или файла. В случае подключаемых С файлов

наследование означает, что один файл включает другой файл, а затем заменяет

или добавляет что-либо.

Если наследуемый подключаемый файл и основной подключаемый файл имеют

различные имена, то такое наследование называется прямым. При этом

используется конструкция '#include "BASE"' в наследуемом файле.

Иногда необходимо чтобы у наследуемого и основного подключаемого файла

были одинаковые имена.

Например, предположим, что прикладная программа использует системный

подключаемый файл 'sys/signal.h', но версия файла '/usr/include/sys/signal.h'

на данной системе выполняет того, что требуется в прикладной программе.

Будет удобнее определить локальную версию, возможно с именем

'/usr/local/include/sys/signal.h' для замены или добавления к версии,

поставляемой с системой.

Это можно выполнить с применением опции '-I.' при компиляции, а также

созданием файла 'sys/signal.h' который выполняет требуемые программе функции.

Но сделать так, чтобы этот файл включал стандартный файл 'sys/signal.h' не

так просто. При включении строки '#include <sys/signal.h>' в этот файл

произойдет подключение новой версии файла, а не стандартной системной версии.

Это приведет к рекурсии и ошибке при компиляции.

При использовании директивы `#include </usr/include/sys/signal.h>'

нужный файл будет найден, но этот способ является не эфективным, так как

содержит полный путь к системному файлу. Это может отразиться на содержании

системы, так как это означает, что любые изменения местоположения системных

файлов потребуют дополнительных изменений где-либо еще.

Более эффективным решением этой проблемы является применение директивы

'#include\_next', которая используется для подключения следующего файла с

таким же именем. Эта директива функционирует также как и директива '#include'

за исключением поиска требуемого файла. Она начинает поиск списка каталогов

подключаемых файлов после каталога, где был найден текущий файл.

Предположим была указана опция '-I /usr/local/include', а список

каталогов для поиска включает '/usr/include'. Также предположим, что оба

каталога содержат файл с именем 'sys/signal.h'. Директива

'#include <sys/signal.h>' найдет нужный файл под каталогом

'/usr/local/include'. Если этот файл содержит строку

'#include\_next <sys/signal.h>', то поиск будет возобновлен после предыдущего

каталога и будет найден файл в каталоге '/usr/include'.

4. Макросы

Макрос это тип сокращения, который можно заранее определить и

использовать в дальнейшем. Существует довольно много возможностей, связанных

с использованием макросов в С препроцессоре.

4.1. Простые макросы

"Простой макрос" это тип сокращения. Это идентификатор, который

используется для представления фрагмента кода.

Перед использованием макроса его необходимо определить с помощью

директивы '#define', за которой следует название макроса и фрагмент кода,

который будет идентифицировать этот макрос. Например,

#define BUFFER\_SIZE 1020

определяет макрос с именем 'BUFFER\_SIZE', которому соответствует текст

'1024'. Если где-либо после этой директивы встретится выражение в следующей

форме:

foo = (char \*) xmalloc (BUFFER\_SIZE);

то С препроцессор определит и заменит макрос 'BUFFER\_SIZE' на его значение и

в результате получится

foo = (char \*) xmalloc (1020);

Использование прописных букв в названиях макросов является стандартным

соглашением и повышает читабельность программ.

Обычно, макроопределением должна быть отдельная строка, как и при

использовании всех директив препроцессора. (Длинное макроопределение можно

разбить на несколько строк с применением последовательности

backslash-newline.) Хотя существует одно исключение: символы перевода

строки могут быть вкючены в макроопределение если они находятся в строковой

или символьной константе, потому как макроопределение не может содержать

каких-либо специальных символов. Макроопределение автоматически дополняется

соответствующим специальным символом, который завершает строчную или

символьную константу. Комментарии в макроопределениях могут содержать символы

перевода строки, так как это ни на что не влияет, потому как все комментарии

полностью заменяются пробелами вне зависимости от того, что они содержат.

В отличие от выше сказанного, не существует никаких ограничений на

значение макроса. Скобки не обязательно должны закрываться. Тело макроса не

обязательно должно содержать правильный С код.

Препроцессор С обрабатывает программу последовательно, поэтому

макроопределения вступают в силу только в местах, где они используются.

Поэтому, после обработки следующих данных С препроцессором

foo = X;

#define X 4

bar = X;

получится такой результат

foo = X;

bar = 4;

После подстановки препроцессором имени макроса, тело макроопределения

добавляется к началу оставшихся вводимых данных и происходит проверка на

продолжение вызовов макросов. Поэтому тело макроса может содержать ссылки

на другие макросы. Например, после выполнения

#define BUFSIZE 1020

#define TABLESIZE BUFSIZE

значением макроса 'TABLESIZE' станет в результате значение '1020'.

Это не является тем же, что и определение макроса 'TABLESIZE' равным

значению '1020'. Директива '#define' для макроса 'TABLESIZE' использует в

точности те данные, которые были указаны в ее теле и заменяет макрос

'BUFSIZE' на его значение.

4.2. Макросы с аргументами

Значение простого макроса всегда одно и то же при каждом его

использовании. Макросы могут быть более гибкими, если они принимают

аргументы. Аргументами являются фрагменты кода, которые прилагаются при

каждом использовании макроса. Эти фрагменты включаются в расширение макроса

в соответствии с указаниями в макроопределении.

Для определения макроса, использующего аргументы, применяется директива

'#define' со списком имен аргументов в скобках после имени макроса. Именами

аргументов могут быть любые правильные С идентификаторы, разделенные запятыми

и, возможно, пробелами. Открывающаяся скобка должна следовать сразу же после

имени макроса без каких-либо пробелов.

Например, для вычисления минимального значения из двух заданных можно

использовать следующий макрос:

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

Для применения макроса с аргументами нужно указать имя макроса, за

которым следует список аргументов, заключенных в скобки и разделенных

запятыми. Количество принимаемых аргументов должно соответствовать количеству

указываемых. Например, макрос 'min' можно использовать так: 'min (1, 2)' или

'min (x + 28, \*p)'.

Значение макроса зависит от используемых аргументов. Каждое имя

аргумента во всем макроопределении заменяется на значения соответствующих

указанных аргументов. При использовании макроса 'min', рассмотренного ранее,

следующим образом:

min (1, 2)

будет получен следующий результат:

((1) < (2) ? (1) : (2))

где '1' заменяет 'X', а '2' заменяет 'Y'.

При указании аргументов, скобки должны закрываться, а запятая не должна

завершать аргумент. Однако, не существует каких либо ограничений на

использование квадратных или угловых скобок. Например

macro (array[x = y, x + 1])

передает макросу 'macro' два аргумента: 'array[x = y' и 'x + 1]'.

После подстановки указанных аргументов в тело макроса, полученный в

результате текст добавляется к началу оставшихся данных и производится

проверка на наличие других вызовов макросов. Поэтому указываемые аргументы

могут содержать ссылки к другим макросам как с аргументами, так и без, а

также к тому же макросу. Тело макроса также может включать ссылки к другим

макросам. Например, макрос 'min (min (a, b), c)' заменяется следующим

текстом:

((((a) < (b) ? (a) : (b))) < (c)

? (((a) < (b) ? (a) : (b)))

: (c))

(Срока разбита на три для ясности и в действительности она не разбивается.)

Если макрос 'foo' принимает один аргумент и нужно передать ему пустой

аргумент, то в скобках следует указать по крайней мере один пробел:

'foo ( )'. Если пробел не указывать, а макрос 'foo' требует один аргумент,

то произойдет ошибка. Для вызова макроса, не принимающего аргументы, можно

использовать конструкцию 'foo0()' как рассмотрено ниже:

#define foo0() ...

Если используется имя макроса, за которым не следует открывающаяся

скобка (после удаления всех следующих пробелов, символов табуляции и

комментариев), то это не является вызовом макроса и препроцессор не изменяет

текст программы. Поэтому возможно использование макроса, переменной и функции

с одним именем и в каждом случае можно изменять, когда нужно применить макрос

(если за именем следует список аргументов), а когда - переменную или функцию

(если список аргументов отстутствует).

Подобное двойственное использование одного имени может привести к

осложнениям и его следует избегать, за исключением случаев, когда оба

значения являются синонимами, то есть когда под одним именем определена

функция и макрос и оба выполняют одинаковые действия. Можно рассматривать

это имя как имя функции. Использование имени не для ссылки (например, для

получения адреса) приведет к вызову функции, в то время как ссылка приведет

к замене имени на значение макроса и в результате будет получен более

эффективный но идентичный код. Например, используется функция с именем

'min' в том же исходном файле, где определен макрос с тем же именем.

Если написать '&min' без списка аргументов, то это приведет к вызову функции.

Если же написать 'min (x, bb)' со списком аргументов, то вместо этого будет

произведена замена на значение соответствующего макроса. Если использовать

конструкцию '(min) (a, bb)', где за именем 'min' не следует открывающаяся

скобка, то будет произведен вызов функции 'min'.

Нельзя определять простой макрос и макрос с аргументами с одним именем.

В определении макроса с аргументами список аргументов должен следовать

сразу после имени макроса без пробелов. Если после имени макроса стоит

пробел, то макрос определяется без аргументов, а остальная часть строки

становится значением макроса. Причиной этому является то, что довольно

часто определяются макросы без аргументов. Определение макросов подобным

образом позволяет выполнять такие операции как

#define FOO(x) - 1 / (x)

(где определяется макрос 'FOO', принимающий один аргумент и добавляет минус

к числу, обратному аргументу) или

#define BAR (x) - 1 / (x)

(где определяется макрос 'BAR' без аргументов и имеющий постоянное значение

'(x) - 1 / (x)').

4.3. Заранее определенные макросы

Некоторые простые макросы являются заранее определенными. Их можно

применять без предварительного определения. Они разделяются на два класса:

стандартные макросы и системно-зависимые макросы.

4.3.1. Стандартные заранее определенные макросы

Стандартные заранее определенные макросы могут применяться вне

зависимости от используемой платформы или операционной системы на которой

функционирует GNU C. Их имена начинаются и заканчиваются двойным символом

подчеркивания. Все макросы в следующем списке до '\_\_GNUC\_\_' являются

стандартизированными ANSI C. Остальные макросы являются расширениями GNU C.

'\_\_FILE\_\_'

Этот макрос заменяется на имя текущего исходного файла в форме строковой

константы С. Возвращаемым именем является одно из указанных в директиве

'#include' или имя основного исходного файла.

'\_\_LINE\_\_'

Этот макрос заменяется на номер текущей строки в форме десятичной целой

константы. В то время как он называется заранее определенным макросом, его

значение меняется динамически.

Этот макрос и макрос '\_\_FILE\_\_' используются при генерировании сообщения

об ошибке для вывода несоответствия, определенного программой. Сообщение

может содержать номер строки исходного файла где была обнаружена ошибка.

Например,

fprintf (stderr, "Internal error: "

"negative string length "

"%d at %s, line %d.",

length, \_\_FILE\_\_, \_\_LINE\_\_);

Директива '#include' изменяет значения макросов '\_\_FILE\_\_' и '\_\_LINE\_\_'

на соответствующие исходному файлу. В конце этого файла, если это был

подключаемый файл, значения '\_\_FILE\_\_' и '\_\_LINE\_\_' становятся теми, какими

они были до директивы '#include' (только значение '\_\_LINE\_\_' увеличивается

на единицу, так как затем обрабатывается строка, следующая за директивой

'#include').

Значения '\_\_FILE\_\_' и '\_\_LINE\_\_' изменяются при использовании директивы

'#line'.

'\_\_DATE\_\_'

Этот макрос заменяется на строчную константу, которая указывает дату

запуска препроцессора. Эта константа содержит одинадцать символов и

выглядит примерно так '"Jan 29 1987"' или '"Apr 1 1905"'.

'\_\_TIME\_\_'

Этот макрос заменяется на строковую константу, которая указывает время

запуска препроцессора. Константа содержит восемь символов и выглядит

примерно так: '"23:59:01:'.

'\_\_STDC\_\_'

Этот макрос заменяется на константу со значением 1 для указания, что

это С стандарта ANSI.

'\_\_STDC\_VERSION\_\_'

Этот макрос заменяется на номер версии стандарта С, длинной целой

константой в форме 'YYYYMML', где YYYY и MM год и месяц выхода версии

стандарта. Это указывает на версию стандарта С, к которой относится

препроцессор.

'\_\_GNUC\_\_'

Этот макрос определен тогда и только тогда, когда используется GNU C.

Он определен только тогда используется полный GNU C компилятор. Если

вызвать препроцессор отдельно, то этот макрос будет не определен. Его

значение указывает на основной номер версии GNU CC ('1' для версии 1 GNU CC,

которая уже является устаревшей, и '2' для версии 2).

'\_\_GNUC\_MINOR\_\_'

Этот макрос содержит дополнительный номер версии компилятора. Он может

быть использован при работе с отличительными возможностями различных выпусков

компилятора.

'\_\_GNUG\_\_'

Компилятор GNU C определяет этот макрос если компилируемым языком

является С++.

'\_\_cplusplus'

Стандарт ANSI для С++ раньше требовал определения этой переменной.

Хотя ее наличие больше не требуется, в GNU C++ она все еще определяется, как

и в других известных компиляторах С++. Этот макрос может быть использован

для определения каким компилятором был скомпилирован заголовок (С или С++).

'\_\_STRICT\_ANSI\_\_'

Этот макрос определяется тогда и только тогда, когда при вызове GNU C

указывается опция '-ansi'. Он определяется как пустая строка.

'\_\_BASE\_FILE\_\_'

Этот макрос заменяется на имя основного исходного файла в форме

строковой константы С. Это исходный файл, указываемый в качестве параметра

при вызове компилятора С.

'\_\_INCLUDE\_LEVEL\_\_'

Этот макрос заменяется на десятичную целую константу, которая указывает

на уровень вложенности подключаемых файлов. Его значение увеличивается на

единицу при обработке директивы '#include' и уменьшается на единицу при

завершении обработки каждого файла. Начальное значение для файлов,

указываемых в командной строке при вызове компилятора является равным нулю.

'\_\_VERSION\_\_'

Этот макрос заменяется сторокой, указывающей номер версии GNU C.

Обычно это последовательность десятичных чисел, разделенных точками.

Например '"2.6.0"'.

'\_\_OPTIMIZE\_\_'

Этот макрос определяется в оптимизирующих компиляторах. Если но

определен, то это приводит к созданию в подключаемых файлах GNU

альтернативных макроопределений для некоторых функций из системных библиотек.

Проверка или использование значения этого макроса не имеет особого смысла,

до тех пор, пока не будет полной уверенности в том, что программы будут

выполняться с таким же эффектом.

'\_\_CHAR\_UNSIGNED\_\_'

Этот макрос определяется тогда и только тогда, когда тип данных 'char'

является беззнаковым. Он реализован для правильного функционирования

подключаемого файла 'limit.h'. Не следует использовать этот макрос. Вместо

этого можно использовать стандартные макросы, определенные в файле 'limit.h'.

Препроцессор использует этот макрос для определения необходимости в

добавлении знакового бита в больших восьмеричных символьных константах.

'\_\_REGISTER\_PREFIX\_\_'

Этот макрос заменяется на сроку, описывающую префикс, добавляемый к

обозначению регистров процессора в ассемблерном коде. Он может использоваться

для написания ассемблерного кода, функционирующего в различных оболочках.

Например, в оболочке 'm68k-aout' производится замена на строку '""', а

в оболочке 'm68k-coff' макрос заменяется на строку '"%"'.

'\_\_USER\_LABEL\_PREFIX\_\_'

Этот макрос заменяется на строку, описывающую префикс, добавляемый к

меткам пользователя в ассемблерном коде. Он может использоваться для

написания ассемблерного кода, функционирующего в различных оболочках.

Например, в оболочке 'm68k-out' он заменяется на строку '" "', а в оболочке

'm68k-coff' - на строку '""'.

4.3.2. Нестандартные заранее определенные макросы

Обычно С препроцессор имеет несколько заранее определенных макросов,

значения которых различаются в зависимости от используемой платформы и

операционной системы. В данном руководстве не представляется возможным

рассмотреть все макросы. Здесь описаны только наиболее типичные из них.

Для просмотра значений заранее определенных макросов можно воспользоваться

командой 'cpp -dM'.

Некоторые нестандартные заранее определенные макросы более или менее

подробно описывают тип используемой операционной системы. Например,

'unix'

Этот макрос обычно определен на всех системах Unix.

'BSD'

Этот макрос определен на последних версиях системы Berkley Unix

(возможно только в версии 4.3).

Другие макросы описывают тип центрального процесора. Например,

'vax'

Определен на Vax компьютерах.

'mc68000'

Определен на большинстве компьютеров, использующих процессор Motorola

68000, 68010 или 68020.

'm68k'

Также определен на большинстве компьютеров с процессором 68000, 68010

или 68020. Хотя некоторые разработчики используют 'mc68000', а некоторые -

'm68k'. Некоторые заранее определяют оба макроса.

'M68020'

Определен на некоторых системах с процессором 68020 в дополнение к

макросам 'mc68000' и 'm68k', которые являются менее специфичными.

'\_AM29K'

'\_AM29000'

Определены на компьютерах с процессорами из семейства AMD 29000.

'ns32000'

Определен на компьютерах, использующих процессоры серии National

Semiconductor 32000.

Другие нестандартные макросы описывают изготовителей компьютерных

систем. Например,

'sun'

Определен на всех моделях компьютеров Sun.

'pyr'

Определен на всех моделях компьютеров Pyramid.

'sequent'

Определен на всех моделях компьютеров Sequent.

Эти заранее определенные символы являются не только нестандартными, но

они к тому же не соответствуют стандарту ANSI, потому что их имена не

начинаются с символа подчеркивания. Поэтому опция '-ansi' запрещает

определение этих символов.

Это приводит к тому, что опция '-ansi' становится бесполезной, так как

большое количество программ зависит от нестандартных заранее определенных

символов. Даже системные подключамые файлы проверяют их значения и

генерируют неправильные объявления в случае если требуемые имена не

определены. В результате очень мало программ компилируется с опцией '-ansi'.

Что же нужно сделать в ANSI C программе для того, чтобы проверить тип

используемого компьютера?

Для этой цели GNU C предоставляет параллельную серию символов, имена

которых состоят из обычных символов с добавлением строки '\_\_' с начала и

с конца. Таким образом символ '\_\_vax\_\_' используется на системах Vax, и

так далее.

Набор нестандартных заранее определенных символов в GNU C препроцессоре

изменяется (при компиляции самого компилятора) с помощью макроса

'CPP\_PREDEFINES', которым является строка, состоящая из опций '-D',

разделенных пробелами. Например, на системе Sun 3 используется следующее

макроопределение:

#define CPP\_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"

Этот макрос обычно указывается в файле 'tm.h'.

4.4. Стрингификация

"Стрингификация" означает преобразование фрагмента кода в строковую

константу, которая содержит текст этого фрагмента кода. Например, в результате

стрингификации 'foo (z)' получается '"foo (z)"'.

В С препроцессоре, стрингификация является опцией, используемой при

замене аргументов в макросе макроопределением. При появлении имени аргумента

в теле макроопределения, символ '#' перед именем аргумента указывает на

стрингификацию соответствующего аргумента при его подстановке в этом месте

макроопределения. Этот же аргумент может быть заменен в другом месте

макроопределения без его стрингификации, если перед именем аргумента нет

символа '#'.

Вот пример макроопределения с использованием стрингификации:

#define WARN\_IF(EXP) \

do { if (EXP) \

fprintf (stderr, "Warning: " #EXP "\n"); } \

while (0)

Здесь аргумент 'EXP' заменяется один раз обычным образом (в конструкции

'if'), а другой - с использованием стрингификации (аргумет функции

'fprintf'). Конструкция 'do' и 'while (0)' является реализацией макроса

'WARN\_IF (ARG);'.

Возможности срингификации ограничены до преобразования одного макро

аргумента в одну строковую константу: не существует методов комбинирования

аргумента с другим текстом и посследующей стрингификации полученных данных.

Хотя рассмотренный выше пример показывает как может быть достигнут подобный

результат в стандартном ANSI C с использованием возможности объединения

смежных строковых констант в одну. Препроцессор стрингифицирует реальное

значение 'EXP' в отдельную строковую константу и в результате получается

следующий текст:

do { if (x == 0) \

fprintf (stderr, "Warning: " "x == 0" "\n"); } \

while (0)

но С компилятор обнаруживает три строковые константы, расположенные друг

за другом и объединяет их в одну:

do { if (x == 0) \

fprintf (stderr, "Warning: x == 0\n"); } \

while (0)

Стрингификация в С является не только заключением требуемого текста в

кавычки. Необходимо помещать символ backslash перед каждым дополнительным

символом кавычки, а также перед каждым символом backslash в символьной или

строковой константе для получения строковой константы в стандарте С. Поэтому

при стрингификации значения 'p = "foo\n";' в результате получится строка

'"p = \"foo\\n\";"'. Однако символы backslash, не принадлежащие символьной

или строковой константе, не дублируются: значение '\n' стрингифицируется в

'"\n"'.

Пробелы (включая комментарии), находящиеся в тексте, обрабатываются

в соответствии с установленными правилами. Все предшествующие и последующие

пробелы игнорируются. Любые последовательности пробелов в середине текста

в результате обработки заменяются на отдельный пробел.

4.5. Объединение

"Объединение" означает соединение двух строковых констант в одну. При

работе с макросами, это означает объединение двух лексических единиц в одну

более длинную. Один аргумент макроса может быть объединен с другим аргументом

или с каким-либо текстом. Полученное значение может быть именем функции,

переменной или типа, а также ключевым словом С. Оно даже может быть именем

другого макроса.

При определении макроса, проверяется наличие операторов '##' в его

теле. При вызове макроса и после подстановки аргументов все операторы '##',

а также все пробелы рядом с ними (включая пробелы, принадлежащие аргументам)

удаляются. В результате производится объединение синтаксических конструкций

с обоих сторон оператора '##'.

Рассмотрим С программу, интерпретирующую указываемые команды. Для этого

должна существовать таблица команд, возможно массив из структур, описанный

следующим образом:

struct command

{

char \*name;

void (\*function) ();

};

struct command commands[] =

{

{ "quit", quit\_command},

{ "help", help\_command},

...

};

Более удобным будет не указывать имя каждой команды дважды: один раз

в строковой константе, второй - в имени функции. Макрос, принимающий в

качестве аргумента имя команды позволяет избежать это. Строковая константа

может быть создана с помощью стрингификации, а имя функции - путем

объединения аргумента со строкой '\_command'. Ниже показано как это сделать:

#define COMMAND(NAME) { #NAME, NAME ## \_command }

struct command commands[] =

{

COMMAND (quit),

COMMAND (help),

...

};

Обычным объединением является объединение двух имен (или имени и какого

либо числового значения) в одно. Также возможно объединение двух числовых

значений (или числового значения и имени) в одно. Операторы, состоящие из

нескольких символов (такие как '+='), также могут быть получены с помощью

объединения. В некоторых случаях возможно объединение строковых констант.

Однако, два текстовых значения, не образующих вместе правильной лексической

конструкции, не могут быть объединены. Например, объединение с одной стороны

символа 'x', а с другой - '+' является бессмысленным с точки зрения

формирования лексических конструкций С. В стандарте ANSI указано, что

подобный тип объединения не определен, хотя препроцессор GNU C их определяет.

В данном случае он помещает вместе символы 'x' и '+' вместе без каких либо

побочных эффектов.

Следует заметить, что препроцессор С преобразует все комментарии в

пробелы перед обработкой макросов. Поэтому нельзя создать комментарий

путем объединения '/' и '\*' так как последовательность символов '/\*' не

является лексической конструкцией. Также можно использовать комментарии в

макроопределениях после строки '##' или в объединяемых аргументах, так как

сначала комментарии заменяются на пробелы, а при объединении эти пробелы

игнорируются.

4.6. Удаление макросов

"Удалить" макрос означает отменить его определение. Это производится с

помощью директивы '#undef', за которой следует имя макроса.

Как и определение, удаление макросов появляется в определенном месте

исходного файла и вступает в силу с этого места. Например,

#define FOO 4

x = FOO;

#undef FOO

x = FOO;

заменяется на

x = 4;

x = FOO;

В этом примере значение 'FOO' должно быть лучше переменной или

функцией, чем макросом, для получения после подстановки правильного С кода.

Директива '#undef' используется в такой же форме и для отмены

макроопределений с аргументами или без них. Применение этой директивы к

неопределенному макросу не дает никакого эффекта.

4.7. Переопределение макросов

"Переопределение" макроса означает определение (с помощью директивы

'#include') имени, которое уже было определено как макрос.

Переопределение явялется простым, если новое определение явно

идентично старому. Иногда не требуется специально выполнять простое

переопределение, хотя оно производится автоматически, если подключаемый

файл вкючается более одного раза, поэтому оно выполняется без какого-либо

эффекта.

Нетривиальные переопределения рассматриваются как возможная ошибка,

поэтому в таких случаях препроцессор выдает предупреждающее сообщение.

Однако, это иногда помогает при изменении определения макроса во время

предварительной компиляции. Появление предупреждающего сообщения можно

запретить путем предварительного уничтожения макроса с помощью директивы

'#undef'.

Для простого переопределения новое определение долно точно совпадать

с предыдущим значением за исключением двух случаев:

В начале и в конце определения могут быть добавлены или удалены пробелы.

Пробелы можно изменять в середине определения (но не в середине

строки). Однако они не могут быть полностью удалены, а также не могут быть

вставлены туда, где их не было вообще.

4.8. Особенности использования макросов

В этом разделе рассматриваются некоторые специальные правила работы,

связанные с макросами и макроподстановками, а также указываются отдельные

случаи, которые следует иметь в виду.

4.8.1. Неправильно используемые конструкции

При вызове макроса с аргументами, они подставляются в тело макроса, а

затем просматриваются полученные после подстановки данные вместе с оставшейся

частью исходного файла на предмет дополнительных макро вызовов.

Возможно объединение макро вызова, исходящего частично от тела макроса

и частично - от аргументов. Например,

#define double(x) (2\*(x))

#define call\_with\_1(x) x(1)

здесь строка 'call\_with\_1 (double)' будет заменена на '(2\*(1))'.

Макроопределения не обязательно должны иметь закрывающиеся скобки. Путем

использования не закрывающейся скобки в теле макроса возможно создание

макро вызова, начинающегося в теле макроса и заканчивающегося вне его.

Например,

#define strange(file) fprintf (file, "%s %d",

...

strange(stderr) p, 35)

В результате обработки этого странного примера получится строка

'fprintf (stderr, "%s %d", p, 35)'.

4.8.2. Нестандартная группировка арифметических выражений

Во большинстве примеров макроопределений, рассмотренных выше, каждое

имя макроаргумента заключено в скобки. В дополнение к этому, другая пара

скобок используется для заключения в них всего макроопределения. Далее

описано, почему лучше всего следует писать макросы таким образом.

Допустим, существует следующее макроопределение:

#define ceil\_div(x, y) (x + y - 1) / y

которое используется для деления с округлением. Затем предположим, что он

используется следующим образом:

a = ceil\_div (b & c, sizeof (int));

В результате эта строка заменяется на

a = (b & c + sizeof (int) - 1) / sizeof (int);

которая не выполняет требуемой задачи. Правила приоритета операторов С

позволяют написать следующую сроку:

a = (b & (c + sizeof (int) - 1)) / sizeof (int);

но требуется

a = ((b & c) + sizeof (int) - 1)) / sizeof (int);

Если определить макрос следующим образом:

#define ceil\_div(x, y) ((x) + (y) - 1) / (y)

то будет получен желаемый результат.

Однако, нестандартная группировка может привести к другому результату.

Рассмотрим выражение 'sizeof ceil\_div(1, 2)'. Здесь используется выражение

С, вычисляющее размер типа данных 'ceil\_div(1, 2)', но в действительности

производятся совсем иные действия. В данном случае указанная срока заменяется

на следующую:

sizeof ((1) + (2) - 1) / (2)

Здесь определяется размер типа целого значения и делится пополам.

Правила приоритета помещают операцию деления вне поля действия операции

'sizeof', в то время как должен определяться размер всего выражения.

Заключение в скобки всего макроопределения позволяет избежать подобных

проблем. Далее дан правильный пример определения макроса 'ceil\_div'.

#define ceil\_div(x, y) (((x) + (y) - 1) / (y))

4.8.3. Использование точки с запятой

Иногда требуется определять макросы, используемые в составных

конструкциях. Рассмотрим следующий макрос, который использует указатель

(аргумент 'p' указывает его местоположение):

#define SKIP\_SPACES (p, limit) \

{ register char \*lim = (limit); \

while (p != lim) { \

if (\*p++ != ' ') { \

p--; break; }}}

Здесь последовательность backslash-newline используется для разбиения

макроопределения на несколько строк, поскольку оно должно быть на одной

строке.

Вызов этого макроса может выглядеть так: 'SKIP\_SPACES (p, lim)'. Грубо

говоря, при его вызове он заменяется на составную конструкцию, которая

является полностью законченной и нет необходимости в использовании точки с

запятой для ее завершения. Но вызов этого макроса выглядит как вызов функции.

Поэтому удобнее будет вызывать этот макрос следующим образом:

'SKIP\_SPACES (p, lim);'

Но это может привести к некоторым трудностям при использовании его перед

выражением 'else', так как точка с запятой является пустым выражением.

Рассмотрим такой пример:

if (\*p != 0)

SKIP\_SPACES (p, lim);

else ...

Использование двух выражений (составной конструкции и пустого выражения)

между условием 'if' и конструкцией 'else' создает неправильный С код.

Определение макроса 'SKIP\_SPACES' может быть изменено для устранения

этого недостатка с использованием конструкции 'do ... while'.

#define SKIP\_SPACES (p, limit) \

do { register char \*lim = (limit); \

while (p != lim) { \

if (\*p++ != ' ') { \

p--; break; }}} \

while (0)

Теперь макрос 'SKIP\_SPACES (p, lim);' заменяется на

do {...} while (0);

что является одним выражением.

4.8.4. Удвоение побочных эффектов

Во многих С программах определяется макрос 'min' для вычисления

минимума:

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

При вызове этого макроса вместе с аргументом, содержащим побочный

эффект, следующим образом:

next = min (x + y, foo (z));

он заменяется на строку

next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));

где значение 'x + y' подставляется вместо 'X', а 'foo (z)' - вместо 'Y'.

Функция 'foo' используется в этой конструкции только один раз, в то

время как выражение 'foo (z)' используется дважды в макроподстановке. В

результате функция 'foo' может быть вызвана дважды при выполнении выражения.

Если в макросе имеются побочные эффекты или для вычисления значений

аргументов требуется много времени, результат может быть неожиданным. В

данном случае макрос 'min' является ненадежным.

Наилучшим решением этой проблемы является определение макроса 'min'

таким образом, что значение 'foo (z)' будет вычисляться только один раз. В

языке С нет стандартных средств для выполнения подобных задач, но с

использованием расширений GNU C это может быть выполнено следующим образом:

#define min(X, Y) \

({ typeof (X) \_\_x = (X), \_\_y = (Y); \

(\_\_x < \_\_y) ? \_\_x : \_\_y; })

Если не использовать расширения GNU C, то единственным решением будет

осторожное применение макроса 'min'. Например, для вычисления значения

'foo (z)', можно сохранить его в переменной, а затем использовать ее значение

при вызова макроса:

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

...

{

int tem = foo (z);

next = min (x + y, tem);

}

(здесь предполагается, что функция 'foo' возвращает значение типа 'int').

4.8.5. Рекурсивные макросы

"Рекурсивные" макросы - это макросы, в определении которых используется

имя самого макроса. Стандарт ANSI C не рассатривает рекурсивный вызов

макроса как вызов. Он поступает на вывод препроцессора без изменений.

Рассмотрим пример:

#define foo (4 + foo)

где 'foo' также является переменной в программе.

Следуя обычным правилам, каждая ссылка на 'foo' заменяется на значение

'(4 + foo)', затем это значение просматривается еще раз и заменяется на

'(4 + (4 + foo))' и так далее, пока это не приведет к ошибке (memory full)

препроцессора.

Однако, правило об использовании рекурсивных макросов завершит этот

процесс после получения результата '(4 + foo)'. Поэтому этот макрос может

использоваться для прибавления 4 к значению переменной 'foo'.

В большинстве случаев не следует опираться на эту возможность. При

чтении исходных текстов может возникнуть путаница между тем, какое значение

является переменной, а какое - вызовом макроса.

Также используется специальное правило для "косвенной" рекурсии. Здесь

имеется в виду случай, когда макрос X заменяется на значение 'y', которое

является макросом и заменяется на значение 'x'. В результате ссылка на

макрос 'x' является косвенной и происходит от подстановки макроса 'x', таким

образом, это является рекурсией и далее не обрабатывается. Поэтому после

обработки

#define x (4 + y)

#define y (2 \* x)

'x' заменяется на '(4 + (2 \* x))'.

Но предположим, что 'y' используется где-либо еще и не в определении

макроса 'x'. Поэтому использование значения 'x' в подстановке макроса 'y'

не является рекурсией. Таким образом, производится подстановка. Однако,

подстановка 'x' содержит ссылку на 'y', а это является косвенной рекурсией.

В результате 'y' заменяется на '(2 \* (4 + y))'.

Неизвестно где такие возможности могут быть использованы, но это

определено стандартом ANSI C.

4.8.6. Отдельная подстановка макро аргументов

Ранее было объяснено, что макроподстановка, включая подставленные

значения аргументов, заново просматривается на предмет наличия новых макро

вызовов.

Что же происходит на самом деле, является довольно тонким моментом.

Сначала значения каждого аргумента проверяются на наличие макро вызовов.

Затем полученные значения подставляются в тело макроса и полученная макро

подстановка проверяется еще раз на наличие новых макросов.

В результате значения макроаргументов проверяются дважды.

В большинстве случаев это не дает никакого эффекта. Если аргумент

содержит какие-либо макро вызовы, то они обрабатываются при первом проходе.

Полученное значение не содержит макро вызовов и при втором проходе оно не

изменяется. Если же аргументы будут подставлены так, как они были указаны,

то при втором проходе, в случае наличия макро вызовов, будет произведена

макроподстановка.

Рекурсивный макрос один раз подставляется при первом проходе, а второй

раз - при втором. Не подставляемые рекурсивные элементы при выполнении

первого прохода отдельно помечаются и поэтому они не обрабатываются при

втором.

Первый проход не выполняется, если аргумент образован путем

стрингификации или объединения. Поэтому

#define str(s) #s

#define foo 4

str (foo)

заменяется на '"foo"'.

При стрингификации и объединении аргумент используется в таком виде, в

каком он был указан без последующего просмотра его значения. Этот же аргумент

может быть просмотрен, если он указан где-либо еще без использования

стрингификации или объединения.

#define str(s) #s lose(s)

#define foo 4

str (foo)

заменяется на '"foo" lose(4)'.

Возникает вопрос: для чего используется два прохода для просмотра

макроса и почему бы не использовать один для повышения скорости работы

препроцессора. В действительности, здесь есть некоторая разница и она

может быть видна в трех отдельных случаях:

При однородных вызовах макросов.

При использовании макросов, вызывающих другие макросы, которые

используют стрингификацию или объединение.

При использовании макросов, содержащих открытые запятые.

Макро вызовы называются "однородными", если аргумент этого макроса

содержит вызов этого же макроса. Например, 'f' это макрос, принимающий

один аргумент, а 'f (f (1))' является однородной парой вызовов макроса 'f'.

Требуемая подстановка производится путем подстановки значения 'f (1)' и

его замены на определение 'f'. Дополнительный проход приводит к желаемому

результату. Без его выполнения значение 'f (1)' будет заменено как

аргумент и во втором проходе оно не будет заменено, так как будет является

рекурсивным элементом. Таким образом, применение второго прохода

предотвращает нежелательный побочный эффект правила о рекурсивных макросах.

Но применение второго прохода приводит к некоторым осложнениям в

отдельных случаях при вызовах однородных макросов. Рассмотрим пример:

#define foo a,b

#define bar(x) lose(x)

#define lose(x) (1 + (x))

bar(foo)

Требуется преобразовать значение 'bar(foo)' в '(1 + (foo))', которое

затем должно быть преобразовано в '(1 + (a,b))'. Но вместо этого,

'bar (foo)' заменяется на 'lose(a,b)' что в результате приводит к ошибке,

так как 'lose' принимает только один аргумент. В данном случае эта проблема

решается путем использования скобок для предотвращения неоднородности

арифметических операций:

#define foo (a,b)

#define bar(x) lose((x))

Проблема становится сложнее, если аргументы макроса не являются

выражениями, например, когда они являются конструкциями. Тогда использование

скобок неприменимо, так как это может привести к неправильному С коду:

#define foo { int a, b; ... }

В GNU C запятые можно закрыть с помощью '({...})', что преобразует

составную конструкцию в выражение:

#define foo ({ int a, b; ... })

Или можно переписать макроопределение без использования таких запятых:

#define foo { int a; int b; ... }

Существует также еще один случай, когда применяется второй проход. Его

можно использовать для подстановки аргумента с его последующей

стрингификацией при использовании двухуровневых макросов. Добавим макрос

'xstr' к рассмотренному выше примеру:

#define xstr(s) str(s)

#define str(s) #s

#define foo 4

xstr (foo)

Здесь значение 'xstr' заменяется на '"4"', а не на '"foo"'. Причиной

этому служит то, что аргумент макроса 'xstr' заменяется при первом проходе

(так как он не использует стрингификацию или объединение аргумента). В

результате первого прохода формируется аргумент макроса 'str'. Он использует

свой аргумент без предварительного просмотра, так как здесь используется

стрингификация.

4.8.7. Зависимые макросы

"Зависимым" макросом называется макрос, тело которого содержит ссылку

на другой макрос. Это довольно часто используется. Например,

#define BUFSIZE 1020

#define TABLESIZE BUFSIZE

Это не является определением макроса 'TABLESIZE' со значением '1020'.

Директива '#define' для макроса 'TABLESIZE' использует в точности тело

указанного макроса, в данном случае это 'BUFSIZE'.

Подстановка значения 'TABLESIZE' производится только при использовании

этого макроса.

При изменении значения 'BUFSIZE' в каком-либо месте программы ее

выполнение меняется. Макрос 'TABLESIZE', определенный как было описано выше,

всегда заменяется с использованием значения макроса 'BUFSIZE':

#define BUFSIZE 1020

#define TABLESIZE BUFSIZE

#undef BUFSIZE

#define BUFSIZE 37

Теперь значение 'TABLESIZE' заменяется (в две стадии) на '37'.

4.9. Символы newline в макроаргументах

При обычной обработке макросов все символы newline в макроаргументах

используются при макроподстановке. Это означает, что если некоторые аргументы

подставляются более одного раза или вообще не подставляются, то символы

newline могут дублироваться. Если подстановка состоит из нескольких

конструкций, то в результате порядок строк этих конструкций будет нарушен.

Это может привести к неправильным значениям номеров строк в сообщениях об

ошибках или при работе с отладчиком.

При работе GNU C препроцессора в режиме ANSI C, им контролируется

многократное использование одного аргумента. При первом его использовании

подставляются все символы newline, а при последующем использовании эти

символы игнорируются. Но даже при работе в таком режиме может возникнуть

ошибочная нумерация строк если аргументы используются не в надлежащем

порядке или вообще не используются.

Рассмотрим пример:

#define ignore\_second\_arg(a,b,c) a; c

ignore\_second\_arg (foo (),

ignored (),

syntax error);

Синтаксическая ошибка со ссылкой на четвертую строку получается в

результате обработки строки 'syntax error', хотя ошибочное выражение

находится в пятой строке.

5. Условия

В работе препроцессора "условием" называется директива, при выполнении

которой часть программы игнорируется во время компиляции после проверки

некоторых условий. В С препроцессоре в условии могут использоваться как

арифметические выражения, так и имена определенных макросов.

Условие в С препроцессоре в некоторых аспектах имеет сходство с

конструкцией 'if' языка С, но важно понимать их отличия. Условие в

конструкции 'if' проверяется при выполнении программы. Ее целью служит

изменение хода программы в зависимости от обрабатываемых данных. Условие в

препроцессоре проверяется при компиляции программы. Оно используется для

включения в программу различных частей кода в зависимости от условий,

установленных при ее компиляции.

5.1. Для чего используются условия

Существует три основных причины для применения условий.

Для выполнения программы на различных платформах может потребоваться

разный исходный код. В некоторых случаях, программа, написанная для одной

операционной системы, будет некорректно работать в другой операционной

системе. В подобных ситуациях недостаточно уклонения от выполнения

ненужных процедур. Если программа их содержит, то часто случается, что

невозможно скомпоновать и запустить программу. При использовании условной

компиляции, неиспользуемый код может быть исключен из программы.

Иногда требуется скомпилировать один исходный файл в две разные

программы. Случается, что различия в программах заключаются в том, что

одна из них выполняет постоянную, занимающую много времени, обработку данных

или выдает значения этих данных для отладки, в то время как другая не

делает этого.

Применение условий, где проверка выдает заведомо ложный результат,

используется для исключения кода из программы, который может являться

одним из видов комментариев для ссылки на него в будующем.

В большинстве простых программ, предназначенных для выполнения только

на одном компьютере, условия обычно не используются.

5.2. Синтаксис условий

Условие в С препроцессоре начинается с директивы условия: '#if',

'#ifdef' или '#ifndef'. Далее рассматривается только директива '#if'.

5.2.1. Директива '#if'

Простейшая форма использования директивы '#if' рассмотрена ниже.

#if EXPRESSION

CONTROLLED TEXT

#endif /\* EXPRESSION \*/

Комментарий, следующий за директивой '#endif' не является обязательным,

но помогает при написании и чтении программы. Такие комментарии всегда

следует использовать, за исключением небольших конструкций. В

действительности, текст в строке после директивы '#endif' игнорируется

GNU C препроцессором, но стандарт ANSI C требует применения комментариев.

Выражение EXPRESSION является С выражением типа integer, что

представляет собой сильное ограничение. Оно может содержать:

Целые константы, которые рассматриваются как тип 'long' или 'unsigned

long'.

Символьные константы, которые интерпретируются в соответствии с набором

символов и в зависимости от компьютера и операционной системы, на которой

установлен препроцессор. Для таких констант GNU С препроцессор использует

тип данных 'char'. Поэтому являются ли коды некоторых символов отрицательными

значениями, определяется С компилятором, который использовался для

компиляции препроцессора. Если тип 'char' является знаковым, то символы,

значения которых достаточно велики для установки знакового бита,

рассматриваются как отрицательные значения. В противном случае тип 'char'

является всегда положительным значением.

Арифметические операции сложения, вычитания, умножения, деления,

операции с битами, сдвиги, сравнения, а также логические операции ('&&' и

'||').

Идентификаторы, не являющиеся макросами и рассматриваемые как нулевое

значение.

Макро вызовы. Перед вычислением значения выражения сначала производится

макроподстановка.

Слдует заметить, что не допускается использовать операторы 'sizeof' и

значения типа 'enum'. Все значения типа 'enum', также как и все

идентификаторы, не являющиеся макро вызовами, рассматриваются как нулевое

значение.

Текст, находящийся внутри условной конструкции, может включать

директивы препроцессора, которые обрабатываются при выполнении требуемых

условий. Текст может также содержать и другие условные конструкции. Однако

директивы '#if' и '#endif' должны образовывать единую конструкцию.

5.2.2. Директива '#else'

Директива '#else' может использоваться в условной конструкции для

предоставления альтернативного кода программы втом случае, если условие

ложно. Вот как это выглядит:

#if EXPRESSION

TEXT-IF-TRUE

#else /\* Not EXPRESSION \*/

TEXT-IF-FALSE

#endif /\* Not EXPRESSION \*/

Если значение EXPRESSION является ненулевым и используется код

TEXT-IF-TRUE, то директива '#else' рассматривается как ложное условие и

код TEXT-IF-FALSE игнорируется. И наоборот, если условие '#if' - ложно,

то включается код TEXT-IF-FALSE.

5.2.3. Директива '#elif'

Обычное применение однородных условий связано с проверкой более чем

двух возможных вариантов. Например:

#if X == 1

...

#else /\* X != 1 \*/

#if X == 2

...

#else /\* X != 2 \*/

...

#endif /\* X != 2 \*/

#endif /\* X != 1 \*/

Дополнительная директива '#elif' позволяет это сократить как рассмотрено

ниже.

#if X == 1

...

#elif X == 2

...

#else /\* X != 2 and X != 1\*/

...

#endif /\* X != 2 and X != 1\*/

Директива '#elif' означает "иначе если" ("else if"). Также как и

'#else', она помещается в середину конструкции '#if'-'#endif' и подразделяет

ее. Ей не требуется наличия собственной директивы '#endif'. Также как и

'#if', директива '#elif' включает в себя тестируемое выражение.

Текст, следующий за директивой '#elif' включается только в том случае,

когда значение изходящей директивы '#if' - ложно, а условие '#elif' - верно.

В одной конструкции '#if'-'#endif' может использоваться более чем одна

директива '#elif'. Текст после директивы '#elif' включается только в том

случае, когда условие '#elif' - верно и находится после условия '#if' или

предшествующего '#elif', значения которых - ложь. '#else' является

эквивалентом директивы '#elif 1', а '#else' может следовать после

любого количества директив '#elif', в то время как '#elif' не может

следовать за '#else'.

5.3. Сохранение удаленного кода для дальнейших ссылок

Если часть программы была перемещена или удалена, но есть необходимость

в сохранении старого кода в качестве комментария для дальнейших ссылок к

нему, то простейший способ реализации этого заключается в использовании

конструкции '#if 0'-'#endif', внутри которой находится этот код. Это

рациональнее применения обычных комментариев, так как это не всегда

помогает, если этот код также содержит комментарии.

Такая конструкция в любом случае будет безошибочной, даже если

заключенный в нее текст также содержит условия (полные конструкции '#if'-

'#endif').

Однако не следует применять такую конструкцию, если комментируемый

текст не является С кодом. Для этого используются обычные С комментарии.

Директива '#if 0' должна состоять из правильных лексем.

5.4. Условия и макросы

Условия часто используются вместе с макросами или утверждениями, так

как они являются единственными выражениями, чьи значения могут варьироваться

при компиляции. Директива '#if', не использующая макросы или утверждения,

является эквивалентом директиве '#if 1' или '#if 0'.

Например, рассмотрим условие, проверяющее выражение 'BUFSIZE == 1020',

где 'BUFSIZE' является макросом.

#if BUFSIZE == 1020

printf ("Large buffers!\n");

#endif /\* BUFSIZE is large \*/

При программировании часто требуется определить размер переменной или

тип данных в директиве '#if', но препроцессор не обрабатыват такие операторы

как 'sizeof' или ключевые слова как 'int'.

В директиве '#if' применяется специальный оператор 'defined',

используемый для проверки соответствия указанного имени существующему

макросу. В любом случае, значением выражения 'defined NAME' или

'defined (NAME)' является 1, если в данном месте программы определен макрос

с именем NAME, в противном случае значением выражения будет 0. Для оператора

'defined' имеет значение не определение макроса, а то что оно есть.

Рассмотрим пример:

#if defined (vax) || defined (ns16000)

Здесь значением выражения будет истина, если как имя 'vax', так и

'ns16000' определены как макросы. То же самое можно выполнить с помощью

утверждений:

#if #cpu (vax) || #cpu (ns16000)

Если макрос был определен, а затем уничтожен с помощью директивы

'#undef', то последующее применение оператора 'defined' возвратит значение

0, так как это имя больше не определено. Если же макрос заново определен

директивой '#define', то оператор 'defined' возвратит значение 1.

Условия, проверяющие определение одного имени довольно часто

используются, поэтому для этой цели существует две дополнительные условные

директивы.

'#ifdef NAME'

что является эквивалентом '#if defined (NAME)'.

'#ifndef NAME'

что является эквивалентом '#if ! defined (NAME)'.

Макроопределения могут меняться при разных процессах компиляции по

некоторым причинам.

Некоторые макросы являются заранее определенными, в зависимости от

типа используемого компьютера. Например, на компьютерах Vax, имя 'vax'

является заранее определенным макросом. На других компьютерах оно не

определено.

Большое количество макросов определяется системными подключаемыми

файлами. На различных системах и компьютерах определяются разные макросы с

разными значениями. Очень часто полезно проверять эти макросы условными

конструкциями во избежание использования аппаратных возможностей на

компьютере, где они не реализованы.

Макросы являются простым способом настройки пользователями программы

для различных систем или приложений. Например, макрос 'BUFSIZE' может быть

определен в конфигурационном файле программы, который вкючается в качестве

подключаемого файла в каждый исходный файл. Можно использовать макрос

'BUFSIZE' в условии препроцессора для генерации кода, зависящего от

выбранной конфигурации.

Макросы могут определяться или уничтожаться с помощью опций

препроцессора '-D' и '-U' при компиляции программы. Можно сделать так, что

один и тот же исходный файл будет скомпилирован в две различные программы

путем определения нужного макроса, использования условий для проверки

значения этого макроса и передачи значения макроса через опции компилятора.

Утверждения обычно являются заранее определенными, но они также могут

быть определены с помощью директив или опций препроцессора.

5.5. Утверждения

"Утверждения" являются более систематической альтернативой макросам

при создании условий на проверку типа компьютера или системы, используемой

при компиляции программы. Утверждения обычно определены заранее, хотя они

могут быть также определены директивами препроцессора или с помощью опций.

Обычно макросы не классифицируются каким-либо образом по их определению.

Они могут указывать на архитектуру модели компьютера, отдельную модель

компьютера, на операционную систему, ее версию или на специфические

возможности конфигурации. Все это может сочетаться в одном макросе. В

отличие от макросов, утверждения состоят из четко поставленного вопроса и

ответа на него. Вопрос обычно называется "утверждением". Утверждение

выглядит следующим образом:

#PREDICATE (ANSWER)

Для имени PREDICATE следует использовать правильно сформированный

идентификатор. Значением ANSWER может быть любая последоватльность слов.

Здесь все символы являются значимыми, за исключением пробелов, расположенных

в начале и в конце ANSWER. Различия в пробелах в середине значения

игнорируются. Не разрешается использовать символ ')' в значении ANSWER.

Далее приведен пример условия, проверяющего является ли ответ ANSWER

утверждением PREDICATE:

#if #PREDICATE (ANSWER)

Для одного утверждения может существовать несколько ответов. Если ответ

упущен при определении утверждения, то следует проверять, существует ли

у данного утверждения какой-нибудь ответ:

#if #PREDICATE

Большинство проверяемых утверждений являются заранее определенными.

GNU C предоставляет три заранее определенных утверждения: 'system', 'cpu'

и 'machine'. Утверждение 'system' используется для описания типа операционной

системы, 'cpu' - для описания архитектуры компьютера, а 'machine'

предоставляет дополнительную информацию о компьютере. Например, в системе

GNU будут верны следующие утверждения:

#system (gnu)

#system (mach)

#system (mach 3)

#system (mach 3.SUBVERSION)

#system (hurd)

#system (hurd VERSION)

а также возможно и другие. Альтернативные утверждения с более или менее

подробной информацией о версии системы помогут получить ответ на вопрос о

типе операционной системы.

В системе Unix существует утверждение '#system (unix)', а возможно

одно из следующих: `#system (aix)', `#system (bsd)', `#system (hpux)',

`#system (lynx)', `#system (mach)', `#system (posix)', `#system (svr3)',

`#system (svr4)', или `#system (xpg4)' вероятно с последующей информацией

о версии системы.

Другие значения для 'system' это '#system (mvs)' и '#system (vms)'.

Многие Unix С компиляторы предоставляют только один ответ на

утверждение 'system': '#system (unix)', если они вообще используют

утверждения.

Утверждение, ответ которого сосотоит из нескольких слов сильно

отличается от утверждений с ответом из одного слова. Например, утверждение

'system (mach 3.0)' не означает, что 'system (3.0)' - верно. Это также не

всегда означает, что 'system (mach)' тоже верно, но в GNU C последнее

утверждение может быть использовано.

В настоящий момент возможные значения утверждений для 'cpu' являются

'#cpu (a29k)', `#cpu (alpha)', `#cpu (arm)', `#cpu (clipper)', `#cpu

(convex)', `#cpu (elxsi)', `#cpu (tron)', `#cpu (h8300)', `#cpu

(i370)', `#cpu (i386)', `#cpu (i860)', `#cpu (i960)', `#cpu (m68k)',

`#cpu (m88k)', `#cpu (mips)', `#cpu (ns32k)', `#cpu (hppa)', `#cpu

(pyr)', `#cpu (ibm032)', `#cpu (rs6000)', `#cpu (sh)', `#cpu (sparc)',

`#cpu (spur)', `#cpu (tahoe)', `#cpu (vax)', `#cpu (we32000)'.

В С программе можно создавать свои утверждения с помощью директивы

'#assert' следующим образом:

#assert PREDICATE (ANSWER)

(следует заметить отсутствие симола '#' перед PREDICATE.)

При каждом выполнении этой директивы создается новый правильный ответ

для PREDICATE. При утверждении одного ответа предыдущие значения остаются в

силе. Единственный способ удалить утверждение - использовать директиву

'#unassert'. Эта директива имеет такой же систаксис как и '#assert'. Можно

удалить все утверждения для PREDICATE следующим образом:

#unassert PREDICATE

Также имеется возможность добавления или удаления утверждений с

помощью опций при вызове 'gcc' или 'cpp'.

5.6. Директивы '#error' и '#warning'

Директива '#error' вынуждает препроцессор сделать отчет о фатальной

ошибке. Все что следует после '#error' используется для сообщения.

Директива '#error' в теле условия, проверяющего комбинацию параметров,

не до конца поддерживаемых программой, используется для сообщения о

возможной ошибке. Например, если известно, что программа не совсем корректно

выполняется на системе Vax, то можно написать:

#ifdef \_\_vax\_\_

#error Won't work on Vaxen. See comments at get\_last\_object.

#endif

Если имеется несколько конфигурационных параметров, которые должны

быть указаны соответствующим образом при установке, можно использовать

условия для определения несоответствия и выдать сообщение об ошибке.

Например,

#if HASH\_TABLE\_SIZE % 2 == 0 || HASH\_TABLE\_SIZE % 3 == 0 \

|| HASH\_TABLE\_SIZE % 5 == 0

#error HASH\_TABLE\_SIZE should not be divisible by a small prime

#endif

Директива '#warning' аналогична директиве '#error', но приводит к тому,

что препроцессор выдает предупреждающее сообщение и продолжает обработку.

Все что, что следует после '#warning' используется для сообщения.

Эту директиву можно использовать в устаревших подключаемых файлах с

указанием на новую версию файла.

6. Комбинирование исходных файлов

Одна из основных задач С препроцессора - это передача информации

компилятору о месторасположении различных частей программы.

Код программы может формироваться из нескольких исходных файлов при

использовании директивы '#include'. Применение как директивы '#include',

так и условий с макросами приводит к изменению основного исходного файла.

Следует принимать во внимание значение нумерации строк С компилятором

(при сообщениях об ошибках) и отладчиком (например, GDB).

В С препроцессоре существует директива, позволяющая контролировать эту

возможность. Это может пригодиться в случае, если файлом ввода препроцессора

является файл вывода другой программы, такой как 'bison', которая

обрабатывает другой файл, являющийся основным исходным файлом. При подобной

обработке нумерация строк теряется.

Для устранения этого недостатка используется директива '#line', которая

позволяет указать номер строки реального исходного файла вместе с его именем.

Директива '#line' применяется в трех модификациях:

'#line LINENUM'

Здесь LINENUM это десятичная целая константа, указывающая, что следующая

строка является строкой исходного файла с номером LINENUM.

'#line LINENUM FILENAME'

Здесь LINENUM это десятичная целая константа, а FILENAME - строковая

константа, указывающая, что следующая строка является строкой исходного

файла с именем FILENAME, а ее номер - LINENUM. Значение FILENAME должно

быть заключено в двойные кавычки.

'#line ANYTHING ELSE'

Значение ANYTHING ELSE проверяется на наличие макро вызовов, которые

затем подставляются. Результатом должна быть десятичная целая константа,

за которой может следовать стороковая константа, как это рассмотрено выше.

Директива '#line' изменяет значения заранее определенных макросов

'\_\_FILE\_\_' и '\_\_LINE\_\_'.

Вывод препроцессора (который затем перенапрвляется в компилятор)

содержит директивы, подобные '#line', только они начинаются с символа '#',

в отличие от '#line', за которыми следует номер строки и имя файла.

7. Другие директивы препроцессора

В этом разделе рассматриваются три дополнительные директивы. Они

описываются для полноты и не являются часто используемыми.

"Пустая директива" состоит из символа '#', за которым следует символ

новой строки, причем между ними могут быть только пробелы (включая

комментарии). Пустая директива рассматривается как директива препроцессора и

не влияет на общий вывод. Основным значением этой директивы является то,

при обработке строки, содержащей символ '#', на вывод ничего не передается,

в отличие от выводимой строки, содержащей '#'.

Стандарт ANSI указывает, что директива '#pragma' имеет произвольное

значение, устанавливаемое при разработке. В препроцессоре GNU C она

не используется, за исключением значения '#pragma once'. Однако, она

остается при выводе препроцессора и может быть использована в процессе

компиляции.

Директива '#ident' используется для совместимости с некоторыми

системами. За ней следует строка с текстом. На некоторых системах этот

текст копируется в отдельное место объектного файла. Но в большинстве систем

он просто игнорируется и применение этой директивы не дает никакого эффекта.

В действительности, эта директива является единственной директивой,

используемой в подключаемых файлах и функционирует на тех системах,

которые ее поддерживают.

8. Вывод С препроцессора

Вывод С препроцессора выглядит примерно так же, как и ввод, только

все строки с директивами заменяются на пустые и комментарии заменяются

пробелами. Пробелы в середине строки не меняются. Однако пробел вставляется

после большинства макроподстановок.

Имя исходного файла и информация о номере строки передается в строках

следующей формы:

# LINENUM FILENAME FLAGS

которые вставляются в середину вводимого файла (но не в строчную или

символьную константу). Появление такой строки означает, что следующая

строка содержится в файле FILENAME и имеет порядковый номер LINENUM.

После имени файла следует нуль или более флагов, значения которых

могут быть '1', '2', '3' или '4'. Если флагов более одного, то они

разделяются пробелами. Далее следует описание этих флагов.

'1'

Указывает на начало нового файла.

'2'

Указывает на возврат в файл (после включения другого файла).

'3'

Указывает на то, что следующий текст исходит из системного

подключаемого файла, поэтому должны использоваться специальные

предупреждающие сообщения.

'4'

Указывает на то, что следующий текст должен рассматриваться как С код.

9. Вызов GNU С Препроцессора

Обычно нет необходимости вызывать С препроцессор отдельно: он вызывается

автоматически С компилятором. Однако иногда требуется запустить препроцессор

без компилятора.

С препроцессор запрашивает два имени файла в качестве аргументов. Один

из них является входным файлом, а другой - выходным. Препроцессор считыват

входной файл вместе с другими файлами, указанными директивой '#include'.

Получаемые при обработке данные поступают в выходной файл.

Как входным так и выходным файлами может быть параметр '-'. Если он

указан вместо имени входного файла, то все данные считываются со стандартного

ввода, если же он указан вместо выходного файла, то все выходные данные

поступают на стандартый вывод. Также если имя выходного файла или оба имени

файла опущены, то для ввода и вывода вместо отсутствующих имен файлов

используется стандартный ввод и вывод.

Далее следует список опций, используемых С препроцессором. Эти опции

также могут быть указаны при компиляции программ, так как они автоматически

передаются препроцессору при вызове компилятора.

'-P'

Предотвращает генерацию строк, начинающихся с символа '#', вместе с

номером строки. Это может быть полезно при обработке препроцессором

какого-либо текста, не являющегося С программой.

'-C'

Не удаляет комментарии и передает их в выходной файл. Комментарии в

аргументах макросов поступают на вывод перед подстановкой макросов.

'-traditional'

Иммитирует работу более ранней версии препроцессора, что

противопоставляется стандарту ANSI C.

При макроподстановках игнорируется наличие символов двойной

и одинарной кавычки. Имя макроаргумента заменяется на его значение, даже

если он является строчной или символьной константой.

Значение макроса может завершаться в середине строковой или символьной

константы. Константа завершается в тексте программы после макроподстановки.

Конец строки завершает строковую или символьную константу без сообщения

об ошибке.

Комментарии заменяются пустым значением. (В ANSI C комментарии

заменяются пробелом.)

Не производится обработка чисел. Значение '1.0e+4' рассматривается как

три значения: '1.0e', '+' и '4'.

Не запрещается использование макросов в их макроопределениях. Поэтому

использование любого рекурсивно заданного макроса неизбежно приводит к

ошибке.

Символ '#' не имеет какого-либо специального значения в

макроопределении.

Текст программы, находящийся в конце значения макроса может выполняться

вместе с текстом, находящимся после вызова макроса, образуя единую

конструкцию, что невозможно в ANSI C.

Символ '\' в макроаргументе указывает на синтаксический смысл следующего

символа.

'-trigraphs'

Обработка последовательностей trigraph стандарта ANSI. Это

последовательности, состоящие из трех символов и начинающиеся со строки '??'.

Они определены стандартом ANSI C для обозначения отдельных символов.

Например, значением '??/' является '\', поэтому значением ''??/n'' будет

символьная константа, вкючающая символ перевода строки. GNU C препроцессор

не поддерживает все программы в стандарте ANSI C, пока не указана опция

'-trigraphs'.

'-pedantic'

Выдаются предупреждающие сообщения в соответствии со стандартом ANSI C,

например в случаях, если какой-либо другой текст кроме комментариев стоит

после директив '#else' или '#endif'.

'-pedantic-errors'

То же, что и '-pedantic', только выдаются сообщения об ошибках, вместо

предупреждений.

'-Wtrigraphs'

Выдается предупреждение, если обнаруживается trigraph

последовательность (предполагается, что указана опция '-trigraphs').

'-Wcomment'

Выдается предупреждение, если обнаруживается последовательность '/\*'

внутри комментария.

'-Wall'

То же, что и '-Wtrigraphs' и '-Wcomment' (но не '-Wtraditional').

'-Wtraditional'

Выдается предупреждение, если встречаются отдельные конструкции,

имеющие различия в обычном и в ANSI C.

'-I DIRECTORY'

Имя каталога DIRECTORY добавляется к началу списка каталогов, где

производится поиск подключаемых файлов. Это может быть использовано для

вставки нестандартного подключаемого файла со стандартным именем, потому

как эти каталоги просматриваются перед системными каталогами с подключаемыми

файлами. При использовании более одной опции '-I' указанные каталоги

просматриваются слева направо, затем просматриваются системные каталоги.

'-I-'

Все каталоги, указанные опцией '-I' до указания опции '-I-'

просматриваются только при выполнении директивы '#include "FILE"' и не

используются директивой '#include <FILE>'.

Если после опции '-I-' указаны дополнительные каталоги опцией '-I', то

эти каталоги используются всеми директивами '#include'.

При использовании опции '-I-', текущий каталог не просматривается в

первую очередь директивой '#include "FILE"'. Поэтому текущий каталог

просматривается только в случае его указания опцией '-I.'. Если указать обе

опции '-I-' и '-I.' то можно проследить, какие каталоги просматриваются до

текущего, а какие - после.

'-nostdinc'

Стандартные системные каталоги не используются для поиска подключаемых

файлов. Используются только каталоги, указанные опциями '-I' (а также

текущий каталог, если он указан).

'-nostdinc++'

Не производится поиск подключаемых файлов в стандартных каталогах С++,

а для поиска используются остальные стандартные каталоги. (Эта опция

применяется при построении библиотеки libg++.)

'-D NAME'

Определяется макрос с именем NAME и значением '1'.

'-D NAME=DEFINITION'

Определяется макрос с именем NAME и значением DEFINITION. Не существует

никаких ограничений на значение DEFINITION, но если препроцессор вызывается

из оболочки или подобной программы, то следует использовать специальный

синтаксис для предотвращения передачи символов со специальным значением,

таких как пробелы, оболочке. Если передается более чем один параметр '-D'

для одного и того же значения NAME, то используется значение, стоящее

правее всех.

'-U NAME'

Макрос с именем NAME не определяется. Если передаются оба параметра

'-U' и '-D' для одного имени, то макрос не определяется, так как приоритет

параметра '-U' выше.

'-undef'

Все нестандартные макросы не определяются.

'-A PREDICATE(ANSWER)'

Создается утверждение с именем PREDICATE и значением ANSWER.

Возможно использование опции '-A-' для предотвращения создания всех

утверждений. Эта опция также уничтожает все макросы, указывающие на тип

используемой системы.

'-dM'

Вместо вывода результата обработки, выводится список директив '#define'

для всех определенных макросов при работе препроцессора, включая заранее

определенные макросы. Это позволяет выяснить определения всех макросов в

используемой версии препроцессора. Предполагая, что файл 'foo.h' не

существует, следующая команда

touch foo.h; cpp -dM foo.h

выявит значения всех заранее определенных макросов.

'-dD'

То же, что и '-dM' за исключением двух аспектов: заранее определенные

макросы не включаются и выводятся как все директивы '#define', так и

результат обработки. Весь вывод поступает в стандартный файл вывода.

'-M [-MG]'

Вмето вывода результата обработки, выводится информация в формате

команды 'make', описывающая зависимости основного исходного файла.

Препроцессор выводит информацию для 'make', которая включает в себя имя

объектного файла для данного исходного файла, запятую, и имена всех

подключаемых файлов. Если подключаемых файлов довольно большое количество,

то строка разбивается на несколько с помощью последовательности

'\'-newline.

Опция '-MG' указывает, что отсутствующие подключаемые файлы являются

генерируемыми файлами и предполагается что они находятся в одном каталоге

с исходными файлами. Она должна быть указана вместе с опцией '-M'.

Эта возможность используется для автоматического обновления make-файлов.

'-MM [-MG]'

То же, что и '-M', только используются файлы, указываемые директивой

'#include "FILE"'. Системные подключаемые файлы, указываемые директивой

'#include <FILE>', не используются.

'-MD FILE'

То же, что и '-M', только вся информация записывается в файл с именем

FILE. Используется в дополнение к стандартной компиляции файла. При указании

этой опции не запрещается стандартная компиляция в отличие от опции '-M'.

При вызове gcc аргумент FILE не указывается. Gcc создает файлы путем

замены ".c" на ".d" в конце имени исходного файла.

При использовании Mach возможно применение утилиты 'md' для объединения

нескольких файлов в один, который затем можно использовать с командой 'make'.

'-MMD FILE'

То же, что и '-MD' за исключением того, что используются подключаемые

файлы пользователя, а не системные подключаемые файлы.

'-H'

Выводятся имена всех используемых подключаемых файлов.

'-imacros FILE'

Обработка файла FILE в качестве ввода вне зависимости от вывода перед

обработкой стандартного входного файла. Так как вывод при обработке файла

FILE не используется, то единственным применением этой опции является

определение макросов в файле FILE для их последующего использования при

обработке исходного файла.

'-include FILE'

Обработка файла FILE в качестве входного с последующим включением

всех данных, полученных при обработке, в основной входной файл перед его

обработкой.

'idirafter DIR'

Имя каталога DIR добавляется ко второму пути поиска подключаемых файлов.

Каталоги, указанные во втором пути, просматриваются в том случае, если

требуемый подключаемый файл не был найден в каталогах, указанных в основном

пути поиска (он задается опцией '-I').

'-iprefix PREFIX'

Определяет значение PREFIX, как префикс для последующих опций

'-iwithprefix'.

'-iwithprefix DIR'

Имя каталога DIR добавляется ко второму пути поиска подключаемых файлов.

Имя каталога создается путем объединения значений PREFIX и DIR, где значение

PREFIX дополнительно указывается опцией '-iprefix'.

'-isystem DIR'

Каталог DIR добавляется в начало второго пути поиска подключаемых

файлов, помечая его как системный каталог. Таким образом, этот каталог

используется также как и стандартные системные каталоги.

`-lang-c'

`-lang-c89'

`-lang-c++'

`-lang-objc'

`-lang-objc++'

Указывается язык исходного файла. Опция '-lang-c' указывается по

умолчанию. Она допускает использование комментариев С++ (комментарии,

начинающиеся со строки '//' изаканчивающиеся в конце этой строки). Опция

'-lang-c89' запрещает использование комментариев С++. При указании опции

'-lang-c++' обрабатываются комментарии С++ и используются дополнительные

каталоги для подключаемых файлов С++. Опция '-lang-objc' допускает

использование директивы объектного С '#import'. Опция '-lang-objc++'

использует все возможности расширений объектного С и С++.

Эти опции создаются драйвером компилятора 'gcc', но не передаются

через командную строку при вызове 'gcc' до тех пор пока не будет указана

опция драйвера '-Wp'.

'-lint'

Производится поиск команд, заключенных в комментарии, для программы

проверки 'lint' с их последующим включением в текст программы с префиксом

'#pragma lint'. Например, комментарий '/\* NOTREACHED \*/' после обработки

препроцессором становится строкой '#pragma lint NOTREACHED'.

Использование этой опции возможно только при прямом вызове 'cpp'.

'gcc' не передает эту опцию из своей командной строки.

'-$'

Запрещается использование символа '$' в идентификаторах. Это

используется для согласования со стандартом ANSI. 'gcc' автоматически

передает эту опцию препроцессору при указании опции '-ansi', но сам 'gcc'

не распознает опцию '-$'. Ее нужно указать препроцессору отдельно для ее

применения без дополнительного воздействия опции '-ansi'.