5. Макросы

Макрос это тип сокращения, который можно заранее определить и использовать в дальнейшем. Существует довольно много возможностей, связанных с использованием макросов в C препроцессоре.

5.1 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 C 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 включены 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 C код.

Препроцессор C обрабатывает программу последовательно, поэтому макроопределения вступают в силу только в местах, где они используются. Поэтому, после обработки следующих данных C 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.

5.2 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 C 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)').

5.3 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.

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 C. 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 C стандарта ANSI.

'__STDC_VERSION__'

Этот макрос заменяется на номер версии стандарта C, длинной целой константой в форме 'YYYYMML', где YYYY и MM год и месяц выхода версии стандарта. Это указывает на версию стандарта C, 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 C++.

'__cplusplus'

Стандарт ANSI для C++ раньше требовал определения этой переменной. Хотя ее наличие больше не требуется, в GNU C++ она все еще определяется, как и в других известных компиляторах C++. Этот макрос может быть использован для определения каким компилятором был скомпилирован заголовок (C или C++).

'__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 C. Это исходный файл, указываемый в качестве параметра при вызове компилятора C.

'__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 '""'.

Нестандартные заранее определенные макросы

Обычно C 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 подключаемые 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'.

5.4 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 C 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 сринги фикации 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 C 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 C 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 C. 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.

5.5 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 C. 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 C 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 C. 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 C 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.

5.6 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 C 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.

5.7 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.

5.8 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.

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)'.

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 C 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 C, 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))
```

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 C 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.

Удвоение побочных эффектов

Во многих C 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 C 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').

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.

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 C 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.

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'.

5.9 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.

Вперед Назад Содержание