А.А. Вылиток

Препроцессор языка Си

Препроцессор языка Си – это обработчик макросов, аналог макрогенератора в макроассемблере. Он просматривает текст программы, написанной на Си, до того, как она будет обработана компилятором. Текст хранится в символьном файле, разбитом на строки символами-маркерами «конец строки».

В некоторых системах препроцессор может быть совмещен с процессом компиляции, в других может быть отдельной программой, результатом которой является файл без макросов на «чистом» Си.

Управляется препроцессор директивами. Директивой является строка, начинающаяся символом #. До и после # могут быть пробелы. За символом # должно следовать имя директивы.

Строки препроцессора распознаются до генерации макрорасширений. Поэтому, если макрос сгенерирует новую директиву препроцессора, она не будет выполнена.

Если в строке исходного файла содержится символ \ непосредственно перед маркером конца строки, то этот символ выбрасывается вместе с маркером и две соседние строки склеиваются в одну строку. Это происходит до обработки директив препроцессора.

Лексемами для препроцессора являются все лексемы языка Си и, кроме того, последовательности символов, задающие имена файлов, например, в директиве #include.

Макроопределения без параметров

Строка вида
`# define` *имя последовательность_лексем*
определяет макрос *имя* с телом *последовательность_лексем* (последовательность может быть пустой). Тело макроса начинается сразу за его именем (никаких знаков равенства и т.п. перед телом нет). Когда в тексте программы встречается имя макроса, оно заменяется его телом (макроподстановка).
Примеры

```
#define EOF  -1
#define EOT '\004'
#define BUF_SIZE  1024
#define ER_MESSAGE ">>error  %d:   %s   \n"
#define forever  for(;;) /* бесконечный цикл */

#define NUMBER_OF_FILES = 7 /* опасно! но лексически корректно,
                                    телом будет  =7     */
```

При таком определении, фрагмент `while (n!=NUMBER_OF_FILES)` будет преобразован к ошибочному фрагменту `while (n!==7)`.

Макроопределения с параметрами

Строка вида
`# define` *имя*(*список_идентификаторов*) *последовательность_лексем* ,

где между именем и открывающей скобкой нет ни одного пробельного символа, является макроопределением с параметрами, задаваемыми списком идентификаторов (список может быть пустым). Параметры в списке перечисляются через запятую. Число

фактических параметров при вызове макроса должно совпадать с числом формальных параметров. При макрорасширении каждое вхождение формального параметра в тело макроса заменяется на соответствующий фактический параметр.

Примеры

[1]

```
#define max(X,Y)  ((X) > (Y) ? (X) : (Y))
```

Фрагмент `a=max(b+c, f(d,e))` будет заменен на

```
a=((b+c) > (f(d,e)) ? (b+c) : (f(d,e)))
```

Вызов макроса похож на вызов функции, причем он подходит для аргументов любого типа, так что не нужно писать разные функции `max` для данных разных типов. Однако в отличие от вызова функции, одно из выражений, задающих фактические параметры в данном примере, вычисляется дважды, что может привести к неожиданным последствиям в случае использования побочных эффектов в этих выражениях:
`max( x++, y++);` ⇒ `a=(( x++) > (y++) ? (x++) : (y++));` — один из аргументов будет увеличен на 1, другой на 2.

[2]

Формальные параметры, вместо которых будут подставляться выражения, в теле макроса следует заключать в скобки, чтобы обеспечить нужный порядок вычислений. Вот пример потенциально ошибочного определения квадрата числа:

```
#define square(x) x*x  /* может быть ошибка при использовании */
```

Для вызова `square(y+1)` получим расширение `y+1*y+1`, не соответствующее квадрату аргумента.

[3]

Макрос `getchar` имеет пустой список параметров:

```
#define getchar() getc(stdin)
```

При вызове этого макроса используется пустой список аргументов:
`while (( c=getchar() != EOF)` …

[4]

Определение функциональных макросов для использования их в качестве операторов-выражений, может приводить к ошибкам. Например:

```
#define swap(a,b)  {unsigned long  _temp=a; a=b; b=_temp;}
```

Попробуем использовать данный макрос в следующем контексте:

```
if (x>y) swap(x,y); /*ошибка*/
else x=y;
```

Макрорасширение будет содержать лишнюю точку с запятой перед `else`:

```
if (x>y) {unsigned long  _temp=x; x=y; y=_temp;}; /*ошибка*/
else x=y;
```

В данной ситуации можно использовать цикл do-while с единственной итерацией.

```
#define swap(a,b)  do { unsigned long  _temp=a; a=b; b=_temp;} \
while(0)
```
Так как синтаксис цикла do-while требует точку с запятой после выражения, в результате макроподстановки получим безошибочный фрагмент:
```
 if (x>y) do {unsigned long  _temp=x; x=y; y=_temp;} while(0);
else x=y;
```

[5]

В результате макроподстановки может появиться новый вызов макроса, который в свою очередь заменяется макрорасширением. Рассмотрим пример:
```
#define sum(x,y)  add(y,x)
#define add(x,y)  ((x)+(y))
```
Вызов `sum(sum(a,b),c)` приведет к следующей последовательности макрорасширений:
```
sum(sum(a,b),c)  ⇒ add(c, sum(a,b))  ⇒  ((c)+(sum(a,b)) ⇒
                     ((c)+(add(b,a))  ⇒  ((c)+(((b)+(a))))
```

[6]

С помощью макросов можно переопределять функции. Следующий макрос переопределяет извлечение квадратного корня с целью особой обработки отрицательных аргументов:
```
#define sqrt(x) ((x)<0 ? sqrt(-(x)) : sqrt(x))
```

Отмена определения макроса

Директива вида

`#undef` *имя*

заставляет препроцессор «забыть» определение макроса *имя*. Использование этой директивы для неопределенного имени не считается ошибкой. После отмены определения имя может быть заново определено с помощью директивы `#define`. Чтобы избежать повторного переопределения, можно использовать директиву условной компиляции `#ifndef` и парную ей конструкцию `#endif`:

```
#ifndef SIZE
#define SIZE 1000
#endif
```

Включение файлов

Директива препроцессора вида

`#include <` *имя файла* `>` или

`#include "`*имя файла*`"`

подставляет весь текст, который находится в указанном файле, на место директивы. Имя файла записывается в формате, зависящем от реализации. Порядок поиска указанного файла также определяется реализацией. Основная цель использования формы вида "…"

состоит в доступе к заголовочным файлам, написанным самим программистом, в то время как форма < … > используется для ссылки на стандартные файлы реализации.

Условная компиляция

С помощью директив условной компиляции препроцессор включает в текст программы или исключает из него строки исходного текста в зависимости от истинности условия. В качестве условия используется константное выражение. Нулевое значение означает «ложь», ненулевое – «истину». Константное выражение должно быть целочисленным и не может содержать в себе перечислимых констант, преобразований типа и операторов sizeof.

БНФ для условной конструкции препроцессора выглядит так:

⟨ *условная конструкция*⟩ ::= ⟨ *if-строка*⟩ ⟨ *elif-части*⟩ [`#else` ⟨ *текст*⟩] `#endif`
⟨ *if-строка*⟩::=`#if` ⟨ *константное выражение*⟩ |
`#ifdef` ⟨ *идентификатор*⟩ | `#ifndef` ⟨ *идентификатор*⟩
⟨ *elif-части*⟩::= { `#elif` ⟨ *константное выражение*⟩ ⟨ *текст*⟩ }

Описанная выше конструкция обрабатывается так, что константные выражения в if- и elif-строках вычисляются по порядку, пока не будет обнаружено истинное выражение. Текст, следующий за директивой с истинным значением, обрабатывается обычным образом, остальные части игнорируются. Если все выражения ложны и присутствует строка `#else`, то следующий за ней текст обрабатывается обычным образом. Директивы `#ifdef` и `#ifndef` позволяют включать текст в случае, если следующий за директивой идентификатор определен или не определен (для `#ifndef`) . Вместо `#ifdef` и `#ifndef` можно использовать `#if` с выражениями `defined` ⟨ *идентификатор*⟩ и `!defined` ⟨ *идентификатор*⟩ соответственно. Например, чтобы застраховаться от повторного включения заголовочного файла hdr.h, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR
/* содержимое hdr.h */
…
#endif
```

Преобразование лексем в строки

Лексема #, которая появляется внутри макроопределения перед параметром, означает что параметр (вместе со знаком # перед ним) заменяется на подставляемый вместо параметра текст, заключенный в двойные кавычки.

Склеивание лексем в макрорасширениях

Склеивание лексем с целью формирования новых лексем осуществляется с помощью операции ##. Две лексемы, разделенные операцией ##, склеиваются в одну лексему.

```
#define TEMP(i)  temp ## i
TEMP(1)= TEMP(2+ m)+ y;
```

После обработки препроцессором это выражение будет иметь следующий вид:

```
temp1 = temp2 + m + y;
```