4. Препроцессор языка Си

Некоторые возможности языка Си обеспечиваются препроцессором, который работает на первом шаге компиляции. Наиболее часто используются две возможности: `#include`, вставляющая содержимое некоторого файла во время компиляции, и `#define`, заменяющая одни текстовые последовательности на другие. В этом параграфе обсуждаются условная компиляция и макроподстановка с аргументами.

4.11.1. Включение файла

Средство `#include` позволяет, в частности, легко манипулировать наборами `#define` и объявлений. Любая строка вида

`#include "`*`имя-файла`*`"`

или

`#include <`*`имя файла`*`>`

заменяется содержимым файла с именем *имя-файла*. Если *имя-файла* заключено в двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось или заключено в угловые скобки `<` и `>`, то поиск осуществляется по определенным в реализации правилам. Включаемый файл сам может содержать в себе строки `#include`.

Часто исходные файлы начинаются с нескольких строк `#include`, ссылающихся на общие инструкции `#define` и объявления `extern` или прототипы нужных библиотечных функций из заголовочных файлов вроде `<stdio.h>`. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к заголовкам, зависят от конкретной реализации.)

имя-файла

Средство `#include` — хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

4.11.2. Макроподстановка

Определение макроподстановки имеет вид:

`#define` *`имя замещающий текст`*

Макроподстановка используется для простейшей замены: во всех местах, где встречается лексема *имя*, вместо нее будет помещен *замещающий-текст*. Имена в `#define` задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст завершает строку, в которой расположено слово `#define`, но в длинных определениях его можно продолжить на следующих строках, поставив в конце каждой продолжаемой строки обратную наклонную черту `\`. Область видимости имени, определенного в `#define`, простирается от данного определения до конца файла. В определении макроподстановки могут фигурировать более ранние `#define`-определения. Подстановка осуществляется только для тех имен, которые расположены вне текстов, заключенных в кавычки. Например, если `YES` определено с помощью `#define`, то никакой подстановки в `printf ("YES")` или в `YESMAN` выполнено не будет.

Любое имя можно определить с произвольным замещающим текстом. Например,

```
#define forever for(;;) /* бесконечный цикл */
```

определяет новое слово `forever` для бесконечного цикла.

Макроподстановку можно определить с аргументами, вследствие чего замещающий текст будет варьироваться в зависимости от задаваемых параметров. Например, определим `max` следующим образом:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Хотя обращения к `max` выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае `A` и `B`) будет заменяться соответствующим ему аргументом. Так, строка

```
x = max(p+q, r+s);
```

будет заменена на строку

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Поскольку аргументы допускают любой вид замены, указанное определение `max` подходит для данных любого типа, так что не нужно писать разные `max` для данных разных типов, как это было бы в случае задания с помощью функций.

Если вы внимательно проанализируете работу `max`, то обнаружите некоторые подводные камни. Выражения вычисляются дважды, и если они вызывают побочный эффект (из за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max(i++, j++) /* НЕВЕРНО */
```

вызовет увеличение `i` и `j` дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Задумайтесь, что случится, если при определении

```
#define square(x) x*x /* НЕВЕРНО */
```

вызвать `square(z+1)`.

Тем не менее, макросредства имеют свои достоинства. Практическим примером их использования является частое применение `getchar` и `putchar` из `<stdio.h>`, реализованных с помощью макросов, чтобы избежать расходов времени от вызова функции на каждый обрабатываемый символ. Функции в `<ctype.h>` обычно также реализуются с помощью макросов.

Действие `#define` можно отменить с помощью `#undef`:

```
#undef getchar
int getchar(void) {…}
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем.

Имена формальных параметров не заменяются, если встречаются в заключенных в кавычки строках. Однако, если в замещающем тексте перед формальным параметром стоит знак `#`, этот параметр будет заменен на аргумент, заключенный в кавычки. Это может сочетаться с конкатенацией (склеиванием) строк, например, чтобы создать макрос отладочного вывода:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Обращение к

```
dprint(x/y);
```

развернется в

```
printf("x/y" " = %g\n", x/y);
```

а в результате конкатенации двух соседних строк получим

```
printf("x/y = %g\n", x/y);
```

Внутри фактического аргумента каждый знак " заменяется на \", а каждая \ на \\, так что результат подстановки приводит к правильной символьной константе.

Оператор ## позволяет в макрорасширениях конкатенировать аргументы. Если в замещающем тексте параметр соседствует с ##, то он заменяется соответствующим ему аргументом, а оператор ## и окружающие его символы-разделители выбрасываются. Например, в макроопределении paste конкатенируются два аргумента

```
#define paste(front, back) front ## back
```

так что paste(name, 1) сгенерирует имя name1.

Правила вложенных использований оператора ## не определены; другие подробности, относящиеся к ##, можно найти в приложении А.

Упражнение 4.14. Определите swap(t, x, y) в виде макроса, который осуществляет обмен значениями указанного типа t между аргументами x и y. (Примените блочную структуру.)

4.11.3. Условная компиляция

Самим ходом препроцессирования можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции.

Вычисляется константное целое выражение, заданное в строке #if. Это выражение не должно содержать ни одного оператора sizeof или приведения к типу и ни одной enum-константы. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до #endif, или #elif, или #else. (Инструкция препроцессора #elif похожа на else if.) Выражение defined(имя) в #if есть 1, если имя было определено, и 0 в противном случае.

Например, чтобы застраховаться от повторного включения заголовочного файла hdr.h, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR

/* здесь содержимое hdr.h */

#endif
```

При первом включении файла hdr.h будет определено имя HDR, а при последующих включениях препроцессор обнаружит, что имя HDR уже определено, и перескочит сразу на #endif. Этот прием может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя.

Вот пример цепочки проверок имени SYSTEM, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
```

```
#define HDR "default.h"
#endif
#include HDR
```

Инструкции `#ifdef` и `#ifndef` специально предназначены для проверки того, определено или нет заданное в них имя. И следовательно, первый пример, приведенный выше для иллюстрации `#if`, можно записать и в таком виде:

```
#ifndef HDR
#define HDR

/* здесь содержимое hdr.h */

#endif
```