Препроцессор

А. Г. Фенстер, fenster@fenster.name

3 апреля 2010 г.

На этом занятии будет кратко рассмотрен препроцессор языка C и его основные директивы.

1 Препроцессинг

Перед тем, как код программы поступит компилятору языка C, его обрабатывает *препроцессор*, команды которого начинаются с символа #. Обычно работа препроцессора состоит в подстановке значений вместо символов, определённых при помощи `#define`, вставке содержимого других файлов (`#include`) и прочих действиях, которые будут рассмотрены подробно ниже. После разбора всех этих команд полученный код передаётся уже непосредственно компилятору языка C.

В ОС Linux можно посмотреть на работу препроцессора, вызвав его командой `cpp`:

```
$ cpp test.c
```

Такую команду можно использовать в тех случаях, когда возникает подозрение, что после обработки директив препроцессора был получен неверный код и хочется проверить результат работы препроцессора.

Препроцессор языка C работает с текстом программы, ничего не зная про синтаксис языка (выполняется только разбиение текста программы на лексемы и удаление комментариев из текста).

Далее мы кратко разберём действие основных директив препроцессора, как обычно, останавливаясь в основном на неочевидных моментах.

2 Директива #include

Вместо строки `#include имя_файла` препроцессор подставит содержимое этого файла. Имя файла может быть указано либо в двойных, либо в «угловых» кавычках. В случае «угловых» кавычек поиск файла будет производиться в «стандартных» для компилятора путях, но не в текущем каталоге. В случае двойных кавычек текущий каталог также будет просматриваться.

```
#include <stdio.h>  /* поиск в стандартном месте */
#include "string.h" /* если в текущем каталоге есть string.h, будет вставлен он,
                       иначе -- из стандартного места */
#include "my.h"     /* поиск только в текущем каталоге */
```

Указанный файл перед вставкой также обрабатывается препроцессором.

Важно понимать, что `#include` ничего не знает о смысле содержимого файла с точки зрения языка C: `#include` — просто текстовая подстановка. Это приводит к проблемам двойной вставки и необходимости использовать include guards, о которых мы говорили на первом занятии.

3 Директивы #define и #undef

`#define идентификатор значение` определяет указанный идентификатор для препроцессора, присваивая ему данное значение. После этого все вхождения этого идентификатор в коде будут заменены на указанное значение. Значением также может быть пустая строка.

`#define N 100` не определяет константу языка C. Компилятор C просто не увидит в программе ни одного идентификатор `N`, т. к. все они будут заменены на число 100.

Файл обрабатывается препроцессором последовательно: идентификатор будет заменяться на значение только после строки, в которой он определён.

Начинающие часто по привычке ставят в конце строки `#define` точку с запятой. Как уже говорилось, препроцессор ничего не знает про синтаксис C, поэтому такое использование, скорее всего, приведёт к появлению ошибки компиляции (которую будет достаточно сложно обнаружить). Например, ошибочное определение `N` как `100;` может привести к тому, что код `int A[N];` будет преобразован к `int A[100;];`.

3.1 Функциональные макросы

При помощи `#define` можно также определять так называемые *функциональные макросы*:

```
#define MIN(a, b) ((a)<(b)?(a):(b))
```

В этом случае все вхождения `MIN` с двумя параметрами будут заменены на соответствующий текст (с подставленными параметрами). Обратите внимание на скобки, поставленные везде, где только можно. Без них будет неверно обработано, например, выражение `2 * MIN(a, b)` (если `MIN` определён без скобок вокруг `a` и `b`, результатом будет `2 * a<b?a:b`, что приведёт к сравнению `2 * a` с `b`).

Схожесть синтаксиса вызова функции и подстановки функционального макроса часто скрывает возможные проблемы. Например, нужно чётко понимать, что при использовании `MIN(f(x), y)` функция `f` может быть вызвана дважды, поскольку в код будет подставлена строка `((f(x))<(y)?(f(x)):(y))`.

Чтобы отличать визуально функциональные макросы от функций, часто стараются писать имена функциональных макросов заглавными буквами.

3.2 Директива #undef

Отмена ранее определённого идентификатора:

```
#undef идентификатор
```

4 Операции конкатенации и преобразования к строке

В случае, если в определении функционального макроса необходимо «склеить» две строки без пробела между ними, используется операция конкатенации `##`:

```
#define MAKE_IDENTIFIER(a, b) a##_##b

int MAKE_IDENTIFIER(vasya, 123);   /* определена переменная vasya_123 */
```

Операция преобразования к строке # (stringification) используется для получения параметра функционального макроса в виде строки C (с экранированными спецсимволами). Следующий макрос очень удобен для отладки:

```
#define DEBUG_PRINT(expr) printf("%s = %d\n", #expr, expr)

DEBUG_PRINT(x + y);      /* printf("%s = %d\n", "x + y", x + y);           */
DEBUG_PRINT(f("abcd"));  /* printf("%s = %d\n", "f(\"abcd\")", f("abcd")); */
```

5 Условные директивы `#if`, `#ifdef`, `#ifndef`

Директива `#ifdef` может использоваться для выбора кода, который поступит на компиляцию, в зависимости от того, определён ли некоторый идентификатор.

```
#ifdef DEBUG
printf("very long debug message\n");
#else
printf("error\n");
#endif
```

Аналогично, `#ifndef` срабатывает, если некоторый идентификатор не определён (это используется в include guards). Директива `#if` вычисляет значение выражения и подставляет код в результат, если значение выражения отлично от нуля.

5.1 Предопределённые идентификаторы

Компиляторы автоматически определяют некоторые идентификаторы, например, `__FILE__` — имя текущего файла (строка в кавычках); `__LINE__` — номер текущей строки в файле. Их весьма удобно использовать в отладочных сообщениях:

```
printf("%s: %d\n", __FILE__, __LINE__);
```

Также каждый компилятор определяет некоторый набор идентификаторов, по которым можно определить, что код компилируется именно им (и написать соответствующую ветку `#ifdef`). Например, GCC определяет `__GNUC__`.

6 Функциональные макросы из нескольких инструкций

При попытке написать функциональный макрос из нескольких инструкций возникает проблема с точкой с запятой в конце определения (например, при использовании макроса в условной инструкции). Пример: макрос

```
#define BAD(a, b) { printf("%d\n", a); printf("%d\n", b); }
```

нельзя использовать в таком коде:

```
if (something)
    BAD(3, 4);  /* эта точка с запятой вызывает проблемы */
else  ...
```

Правильным решением будет заключить подставляемый код в `do ... while (0)`:

```
#define GOOD(a, b) do { printf("%d\n", a); printf("%d\n", b); } while (0)
```

7 Директивы #error и #warning

Эти директивы могут использоваться для прерывания компиляции и печати предупреждения.