Лекция 9
Препроцессоры

Препроцессор C/C++ — это часть компилятора, которая подвергает программу различным текстовым преобразованиям до реальной трансляции исходного кода в объектный. Программист может давать препроцессору команды, называемые *директивами препроцессора* (preprocessor directives), которые, не являясь формальной частью языка, способны расширить область действия его среды программирования.

Препроцессор C++ включает следующие директивы.

#define	#error	#include
#if	#else	#elif
#endif	#ifdef	#ifndef
#undef	#line	#pragma

Все директивы препроцессора начинаются с символа #.

На заметку. *Препроцессор C++ — прямой потомок препроцессора C, и некоторые его средства оказались избыточными после введения в C++ новых элементов. Однако он по-прежнему является важной частью C++-среды программирования.*

Директива #define

Директива *#define* определяет имя макроса и используется для определения идентификатора и символьной последовательности, которая будет подставлена вместо идентификатора везде, где он встречается в исходном коде программы. Этот идентификатор называется макроименем, а процесс замены — *макроподстановкой* (реализацией макрорасширения). Общий формат использования этой директивы имеет следующий вид.

```
#define макроимя последовательность_символов
```

Обратите внимание на то, что здесь нет точки с запятой. Заданная *последовательность_символов* завершается только символом конца строки. Между элементами *макроимя* (имя_макроса) и *последовательность_символов* может быть любое количество пробелов.

Итак, после включения этой директивы каждое вхождение текстового фрагмента, определенное как *макроимя*, заменяется заданным элементом *последовательность_символов*. Например, если вы хотите использовать слово *UP* в качестве значения *1* и слово *DOWN* в качестве значения *0*, объявите такие директивы *#define*.

```
#define UP 1
#define DOWN 0
```

Данные директивы вынудят компилятор подставлять *1* или *0* каждый раз, когда в файле исходного кода встретится слово *UP* или *DOWN* соответственно. Например, при выполнении инструкции:

```
cout << UP << ' ' << DOWN << ' ' << UP + UP;
```

на экран будет выведено следующее:

1 0 2

После определения имени макроса его можно использовать как часть определения других макроимен. Например, следующий код определяет имена *ONE*, *TWO* и *THREE* и соответствующие им значения.

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

Если текстовая последовательность не помещается на строке, ее можно продолжить на следующей, поставив обратную косую черту в конце строки, как показано в этом примере.

```
#define LONG_STRING "Это очень длинная последовательность,\
которая используется в качестве примера."
```

Среди C++-программистов принято использовать для макроимен прописные буквы. Это соглашение позволяет с первого взгляда понять, что здесь используется макроподстановка. Кроме того, лучше всего поместить все директивы *#define* в начало файла или включить в отдельный файл, чтобы не искать их потом по всей программе.

Макроопределения, действующие как функции

Директива *#define* имеет еще одно назначение: макроимя может использоваться с аргументами. При каждом вхождении макроимени связанные с ним аргументы заменяются реальными аргументами, указанными в коде программы. Такие макроопределения действуют подобно функциям.

пример

```
/* Использование "функциональных" макроопределений. */
#include <iostream>
using namespace std;
#define MIN(a, b) (((a)<(b)) ? a : b)
int main()
{
int x, y;
x = 10;
y = 20;
cout << "Минимум равен: " << MIN(x, y);
return 0;
}
```

При компиляции этой программы выражение, определенное идентификатором *MIN (a, b)*, будет заменено, но *x* и *y* будут рассматриваться как операнды. Это значит, что cout инструкция после компиляции будет выглядеть так.

```
cout << "Минимум равен: " << (((x)<(y)) ? x : y);
```

Макроопределения, действующие как функции, — *это макроопределения, которые принимают аргументы.*

Кажущиеся избыточными круглые скобки, в которые заключено макроопределение *MIN*, необходимы, чтобы гарантировать правильное восприятие компилятором заменяемого выражения. На самом деле дополнительные круглые скобки должны применяться практически ко всем макроопределениям, действующим подобно функциям. Нужно всегда очень внимательно относиться к определению таких макросов; в противном случае возможно получение неожиданных результатов.

Рассмотрим, например, программу, которая использует макрос для определения четности значения.

```
// Эта программа дает неверный ответ.
#include <iostream>
using namespace std;
#define EVEN(a) a%2==0 ? 1 : 0
int main()
{
If (EVEN(9+1)) cout << "четное число";
else cout << "нечетное число ";
return 0;
}
```

Эта программа не будет работать корректно, поскольку не обеспечена правильная подстановка значений. При компиляции выражение *EVEN(9+1)* будет заменено следующим образом.

```
9+1%2==0 ? 1 : 0
```

Так как оператор % имеет более высокий приоритет, чем оператор +, то значит, что сначала выполнится операция деления по модулю (%) для числа 1, а затем ее результат будет сложен с числом 9, что (конечно же) не равно 0. Чтобы исправить ошибку, достаточно заключить в круглые скобки аргумент *a* в макроопределении *EVEN*, как показано в следующей (исправленной) версии той же программы.

```
// Эта программа работает корректно.
#include <iostream>
using namespace std;
#define EVEN(a) (a)%2==0 ? 1 : 0
```

```
int main()
{
if(EVEN(9+1)) cout << "четное число";
else cout << "нечетное число";
return 0;
}
```

Теперь сумма 9+1 вычисляется до выполнения операции деления по модулю.

Директива #error

Директива #error отображает сообщение об ошибке и дает указание компилятору остановить компиляцию. Она используется в основном для отладки. Общий формат ее записи таков.

```
#error сообщение
```

Обратите внимание на то, что элемент *сообщение* не заключен в двойные кавычки. При встрече с директивой *#error* отображается заданное *сообщение* и другая информация (она зависит от конкретной реализации рабочей среды), после чего компиляция прекращается. Чтобы узнать, какую информацию отображает в этом случае компилятор, достаточно провести эксперимент.

Директива #include

Директива *#include* обязывает компилятор включить либо стандартный заголовок, либо другой исходный файл, имя которого указано в директиве *#include*. Имя стандартных заголовков заключается в угловые скобки, как показано в примерах, приведенных в этой книге. Например, эта директива

```
#include <vector>
```

включает стандартный заголовок для векторов.

При включении другого исходного файла его имя может быть указано в двойных кавычках или угловых скобках. Например, следующие две директивы обязывают C++ прочитать и скомпилировать файл с именем *sample.h*:

```
#include <sample.h>
#include "sample.h"
```

Если имя файла заключено в угловые скобки, то поиск файла будет осуществляться в одном или нескольких специальных каталогах, определенных конкретной реализацией.

Если же имя файла заключено в кавычки, поиск файла выполняется, как правило, в текущем каталоге (что также определено конкретной реализацией). Во многих случаях это означает поиск текущего рабочего каталога. Если заданный

файл не найден, поиск повторяется с использованием первого способа (как если бы имя файла было заключено в угловые скобки).

Директивы условной компиляции

Существуют директивы, которые позволяют избирательно компилировать части исходного кода. Этот процесс, именуемый *условной компиляцией*, широко используется коммерческими фирмами по разработке программного обеспечения, которые создают и поддерживают множество различных версий одной программы.

Директивы #if, #else, #elif и #endif

Директивы #if, #ifdef, #ifndef, #else, #elif и #endif — *это директивы условной компиляции.*

Главная идея состоит в том, что если выражение, стоящее после директивы *#if* оказывается истинным, то будет скомпилирован код, расположенный между нею и директивой *#endif*. В противном случае данный код будет опущен. Директива *#endif* используется для обозначения конца блока *#if*.

Общая форма записи директивы *#if* выглядит так.

```
#if константное_выражение
    последовательность инструкций
#endif
```

Если *константное_выражение* является истинным, код, расположенный непосредственно за этой директивой, будет скомпилирован.

пример

```
// Простой пример использования директивы #if.
#include <iostream>
using namespace std;
#define MAX 100
int main()
{
#if MAX>10
cout << "Требуется дополнительная память\n";
#endif
// ...
return 0;
}
```

При выполнении эта программа отобразит сообщение *Требуется дополнительная память* на экране, поскольку, как определено в программе, значение константы *MAX* больше 10. Этот пример иллюстрирует важный момент: Выражение, которое стоит после директивы *#if*, вычисляется во время компиляции. Следовательно, оно должно содержать только идентификаторы, которые были

предварительно определены, или константы. Использование же переменных здесь исключено.

Поведение директивы *#else* во многом подобно поведению инструкции *else*. Она определяет альтернативу на случай невыполнения директивы *#if*. Чтобы показать, как работает директива *#else*, воспользуемся предыдущим примером, немного его расширив.

```
// Пример использования директив #if / #else.
#include <iostream>
using namespace std;
#define MAX 6
int main()
{
#if MAX>10
cout << "Требуется дополнительная память.\n");
#else
cout << "Достаточно имеющейся памяти.\n";
#endif
// . . .
return 0;
}
```

В этой программе для имени *MAX* определено значение, которое меньше 10, поэтому *#if*-ветвь кода не скомпилируется, но зато скомпилируется альтернативная *#else*-ветвь.В результате отобразится сообщение *Достаточно имеющейся памяти*.

Директива #elif используется для формирования многозвенной схемы *if-else-if*, представляющей несколько вариантов компиляции. После директивы *#elif* должно стоять константное выражение. Если это выражение истинно, следующий блок кода скомпилируется, и никакие другие *#elif*-выражения не будут тестироваться или компилироваться. В противном случае будет проверено следующее по очереди *#elif*-выражение. Вот как выглядит общий формат использования директивы *#elif*.

```
#if выражение
последовательность инструкций
#elif выражение 1
последовательность инструкций
#elif выражение 2
последовательность инструкций
#elif выражение 3
последовательность инструкций
// . . .
#elif выражение N
последовательность инструкций
```

```
#endif
```

Например, в этом фрагменте кода используется идентификатор *COMPILED_BY*, который позволяет определить, кем компилируется программа.

```
#define JOHN 0
#define BOB 1
#define TOM 2
#define COMPILED_BY JOHN
#if COMPILED_BY == JOHN
char who[] = "John";
#elif COMPILED_BY == BOB
char who[] = "Bob";
#else
char who[] = "Tom";
#endif
```

Директивы *#if* и *#elif* могут быть вложенными. В этом случае директива *#endif*, *#else* или *#elif* связывается с ближайшей директивой *#if* или *#elif*. Например, следующий фрагмент кода совершенно допустим.

```
#if COMPILED_BY == BOB
#if DEBUG == FULL
int port = 198;
#elif DEBUG == PARTIAL
int port = 200;
#endif
#else
cout << "Боб должен скомпилировать код" << "для отладки вывода данных.\n";
#endif
```

Директивы #ifdef и #ifndef

Директивы *#ifdef* и *#ifndef* предлагают еще два варианта условной компиляции, которые можно выразить как *"если определено"* и *"если не определено"*соответственно.

Общий формат использования директивы *#ifdef* :

```
#ifdef макроимя
    последовательность инструкций
#endif
```

Если *макроимя* предварительно определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.

Общий формат использования директивы *#ifndef:*.

```
#ifndef макроимя
   последовательность инструкций
#endif
```

Если *макроимя* не определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.

Как директива *#ifdef*, так и директива *#ifndef* может иметь директиву *#else* или *#elif*.

пример

```
#include <iostream>
using namespace std;
#define TOM
int main()
{
#ifdef TOM
cout << "Программист Том.\n";
#else
cout << "Программист неизвестен.\n";
#endif
#ifndef RALPH
cout << "Имя RALPH не определено.\n";
#endif
return 0;
}
```

При выполнении эта программа отображает следующее.

```
Программист Том.
Имя RALPH не определено.
```

Но если бы идентификатор *TOM* был не определен, то результат выполнения этой программы выглядел бы так.

```
Программист неизвестен.
Имя RALPH не определено.
```

Директивы *#ifdef* и *#ifndef* можно вкладывать точно так же, как и директивы *#if*.

Директива #undef используется для удаления предыдущего определения некоторого макроимени. Ее общий формат:

```
#undef макроимя
```

Пример

```
#define TIMEOUT 100
#define WAIT 0
// . . .
#undef TIMEOUT
#undef WAIT
```

Здесь имена *TIMEOUT* и *WAIT* определены до тех пор, пока не выполнится директива *#undef*.

Основное назначение директивы *#undef* — разрешить локализацию макроимен для тех частей кода, в которых они нужны.

Использование оператора defined

Помимо директивы *#ifdef* существует еще один способ выяснить, определено ли в программе некоторое макроимя. Для этого можно использовать директиву *#if* в сочетании с оператором времени компиляции *defined*. Например, чтобы узнать, определено ли макроимя *MYFILE*, можно использовать одну из следующих команд препроцессорной обработки.

```
#if defined MYFILE
```

или

```
#ifdef MYFILE
```

При необходимости, чтобы реверсировать условие проверки, можно предварить оператор *defined* символом !. Например, следующий фрагмент кода скомпилируется только в том случае, если макроимя *DEBUG* не определено.

```
#if !defined DEBUG
  cout << "Окончательная версия!\n";
#endif
```

О роли препроцессора

Препроцессор C++ — прямой потомок препроцессора языка C, причем без каких-либо усовершенствований. Однако его роль в C++ намного меньше роли, которую играет препроцессор в C. Дело в том, что многие задачи, выполняемые препроцессором в C, реализованы в C++ в виде элементов языка.

На данном этапе препроцессор уже частично избыточен. Например, два наиболее употребительных свойства директивы *#define* были заменены инструкциями C++. В частности, ее способность создавать константное значение и определять макроопределение, действующее подобно функциям, сейчас совершенно избыточна. В C++ есть более эффективные средства для выполнения этих задач. Для создания константы достаточно определить *const*-переменную. А с созданием встраиваемой (подставляемой) функции вполне справляется спецификатор *inline*. Оба эти средства лучше работают, чем соответствующие механизмы директивы *#define*.

Директива #line

Директива #line изменяет содержимое псевдопеременных _ _LINE_ _ и _ _FILE_ _.

Директива *#line* используется для изменения содержимого псевдопеременных_ *_LINE_ _* и_ *_FILE_ _*, которые являются зарезервированными идентификаторами (макроименами).

Псевдопеременная_ *_LINE_ _* содержит номер скомпилированной строки, а псевдопеременная_ *_FILE_ _*— имя компилируемого файла. Базовая форма записи этой команды имеет следующий вид.

#line номер "имя_файла"

Здесь *номер* — это любое положительное целое число, а *имя_файла* — любой допустимый идентификатор файла. Значение элемента *номер* становится номером текущей исходной строки, а значение элемента *имя_файла*— именем исходного файла.

Имя_файла — элемент необязательный. Директива *#line* используется, главным образом, в целях отладки и в специальных приложениях.

Директива #pragma

Работа директивы *#pragma* зависит от конкретной реализации компилятора. Она позволяет выдавать компилятору различные инструкции, предусмотренные создателем компилятора. Общий формат его использования таков.

#pragma имя

Здесь элемент *имя* представляет имя желаемой *#pragma*-инструкции. Если указанное имя не распознается компилятором, директива *#pragma* попросту игнорируется без сообщения об ошибке.

Операторы препроцессора "#" и "##"

В C++ предусмотрена поддержка двух операторов препроцессора: *"#"* и*"##"*. Эти операторы используются совместно с директивой *#define*. Оператор*"#"* преобразует следующий за ним аргумент в строку, заключенную в кавычки. Рассмотрим, например, следующую программу.

```
#include <iostream>
using namespace std;
#define mkstr(s) # s
int main()
{
cout << mkstr(Я в восторге от C++);
return 0;
}
```

Препроцессор C++ преобразует строку

```
cout << mkstr(Я в восторге от C++);
```

в строку

```
cout << "Я в восторге от C++";
```

Оператор ## используется для конкатенации двух лексем. Рассмотрим пример.

```
#include <iostream>
using namespace std;
#define concat(a, b) a ## b
int main()
{
int xy = 10;
cout << concat(x, y);
return 0;
}
```

Препроцессор преобразует строку

```
cout << concat (x, y);
```

в строку

```
cout << xy;
```

Если эти операторы вам кажутся странными, помните, что они не являются операторами "повседневного спроса" и редко используются в программах. Их основное назначение — позволить препроцессору обрабатывать некоторые специальные ситуации.

Зарезервированные макроимена

В языке C++ определено шесть встроенных макроимен.

```
_ _LINE_ _
_ _FILE_ _
_ _DATE_ _
_ _TIME_ _
_ _STDC_ _
_ _cplusplus
```

Макросы _ _*LINE*_ _ и _ _*FILE*_ _ описаны при рассмотрении директивы *#line*. Они содержат номер текущей строки и имя файла компилируемой программы.

Макрос **_ _*DATE*_ _** представляет собой строку в формате ***месяц/день/год***, которая означает дату трансляции исходного файла в объектный код.

Время трансляции исходного файла в объектный код содержится в виде строки в макросе ***_ _TIME_ _***. Формат этой строки следующий: ***часы.минуты.секунды***.

Точное назначение макроса ***_ _STDC_ _*** зависит от конкретной реализации компилятора. Как правило, если макрос_ _*STDC*_ _ определен, то компилятор примет только стандартный C/C++-код, который не содержит никаких нестандартных расширений.

Компилятор, соответствующий ANSI/ISO-стандарту C++, определяет макрос***_ _cplusplus*** как значение, содержащее по крайней мере шесть цифр. "Нестандартные" компиляторы должны использовать значение, содержащее пять (или даже меньше) цифр.