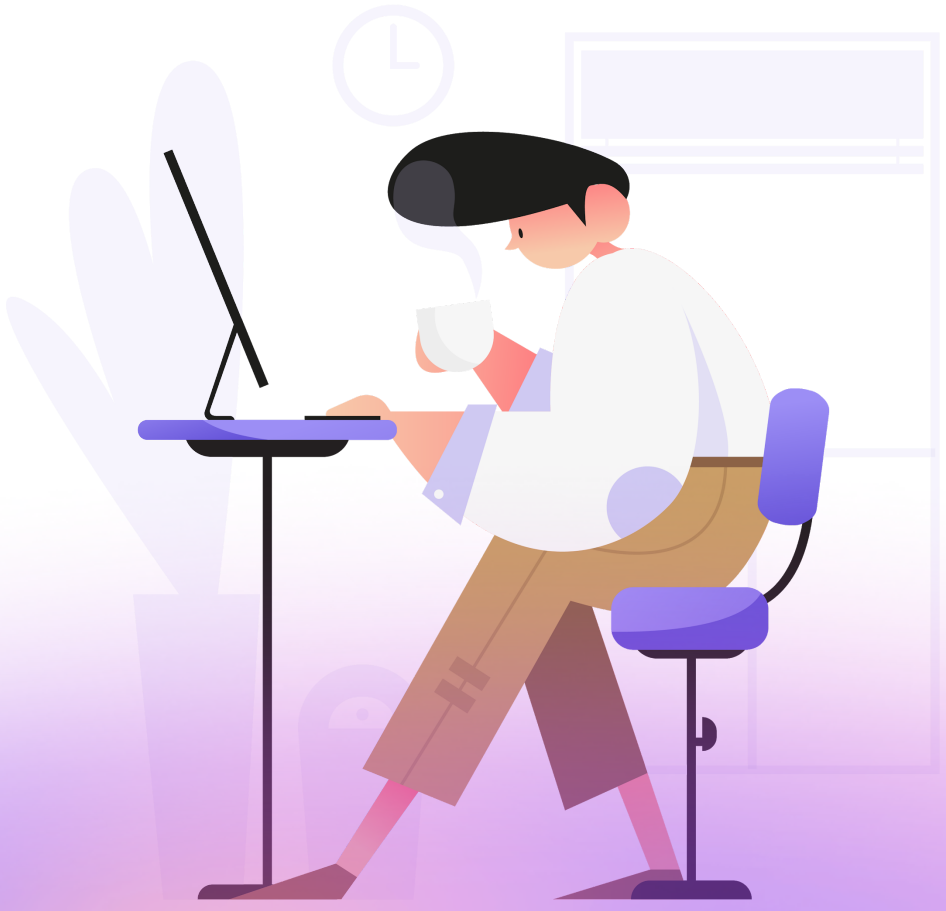
GeekBrains

Основы C++

Препроцесинг



На этом уроке

1. Познакомимся с понятием директивы препроцессора
2. Изучим дополнительные возможности препроцессора С
3. Научимся писать функциональные макросы и осуществлять условную компиляцию

Оглавление

Препроцесинг

На предыдущих занятиях мы познакомились со всеми базовыми элементами языка Си, они же применимы и в языке С++. (Такие базовые понятия, как полиморфизм, инкапсуляция и наследование, а также связанные с ними понятия классов и другие особенности языка, характерные только для С++, мы не затрагивали). Однако стандарт С не ограничивается описанием одного лишь языка. В нем также определено, что должен делать препроцессор, установлено, какие функции формируют стандартную библиотеку С, и детализировано, каким образом работают эти функции. В этой главе мы исследуем препроцессор.

Препроцессор, согласно своему названию, анализирует исходный файл программы на нулевой фазе компиляции, проще говоря, перед тем, как компилятор будет выполнять синтаксический и

семантический анализ исходного текста программы и перевод текста программы в исполняемые коды. Следуя указанным директивам, препроцессор заменяет символические выражения в тексте программы некоторыми значениями, которые они представляют. По существу, он преобразует один текст в другой, с учетом синтаксических правил языка Си. Компилятор языка Си сам вызывает препроцессор, кроме того, препроцессор может быть вызван и автономно. По завершении работы препроцессора можно получить распечатку исходного текста программы после выполнения нулевой фазы компиляции, где текст программы (т.н. “листинг”) еще читаемый на языке Си, но уже выполнены все подстановки и прочие действия в соответствии с **директивами препроцессора**. О получении и открытии такого листинга мы говорили на первом занятии, когда рассматривали этапы трансляции исходного кода.

Стандартные директивы

Использование директив препроцессора (например уже знакомой `#include`) позволяет вставлять в исходный файл программы содержимое другого исходного файла (чаще всего, библиотечного), запрещать или разрешать компиляцию некоторой части исходного файла в зависимости от того или иного условия, и т.п. Ниже приведены стандартные директивы препроцессора, реализованные во всех компиляторах, в алфавитном порядке:

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

Символ `#` должен быть первым в строке, перед этим символом и между ним и директивой, могут быть один или несколько пробелов или табуляций. Некоторые директивы могут содержать аргументы. Подробнее рассмотрим оформление текста директивы препроцессора чуть позднее на примере директивы `#define`.

Специальные символы аргументов препроцессора

`#` - предписывает преобразовать последующий аргумент в строку ASCII-символов, окруженную кавычками

`##` - предписывает срастить между собой строки, определенные слева и справа

`\` - разрешает переход на новую строку в определении макроса

Директивы могут быть записаны в любом месте исходного текста программы, при этом их действие распространяется от строки программы, в которой они записаны (объявлены), и до строки, в которой объявлена отменяющая их действие директива, если таковая имеется. А если отменяющей директивы нет – то до конца исходного файла. Дальнейшее изложение материала по директивам препроцессора должно быть упреждено информацией о самых первых шагах препроцессора.

Первые шаги препроцессора

В самом начале своей работы препроцессор должен провести исходный текст программы через ряд этапов преобразования. Препроцессор начинает свою работу с того, что устанавливает соответствие

символов исходного кода с исходным набором символов. При этом обрабатываются многобайтные символы и триграфы — расширения символов, которые обеспечивают интернациональное применение языка С.
Во вторую очередь препроцессор обнаруживает все вхождения обратной косой черты с последующим символом новой строки и удаляет их. В результате две физические строки,такие как:

```
printf("Это было вели\
        колепно!\n");
```

преобразуются в одну логическую строку:

```
printf("Это было великолепно!\n");
```

Обратите внимание, что в этом контексте символ “обратный слэш” (от английского slash - рубить, полоснуть) есть символ, используемый в языке Си для обозначения “разрубания”, или, если угодно, “разрезания” текущего выражения в исходном коде программы и продолжения этого выражения на следующей строке, а не символическое представление `\n`, завершающее текущую строку исходного кода.

Далее препроцессор разбивает текст программы на последовательность препроцессорных лексем, а также на последовательности пробельных символов и комментариев. (В базовой терминологии лексемы представляют собой группы, отделяемые друг от друга пробелами, табуляциями или разрывами строк. Сейчас интересно отметить, что каждый комментарий заменяется одним символом пробела. Таким образом, код следующего вида:

```
int/* это не похоже на пробел */fox;
```

превращается в

```
int fox;
```

Кроме того, в рамках реализации препроцессора может быть принято решение заменять каждую последовательность пробельных символов (кроме символа новой строки) одиночным пробелом. Наконец, программа готова для этапа предварительной обработки, и препроцессор начинает поиск своих потенциальных директив, обозначаемых символом `#` в начале строки. После ознакомления с самыми первыми преобразованиями, производимыми препроцессором языка Си, вернемся к изложению материала по директивам препроцессора с примерами. Вы уже неоднократно встречали директивы `#define` и `#include`. Теперь можно объединить и расширить полученные знания.

Директива #define

Директива `#define` обычно используется для подмены часто встречающихся в программе констант, ключевых слов, операторов и выражений их идентификаторами, т.е. последовательностями символов, которые называют **именованными константами** или последовательностями операторов и выражений, которые называют **макроподстановками**.
Синтаксис директивы:

```
#define имя текст_подстановки
```

Отменяет действие директивы `#define` директива `#undef` синтаксис которой:

```
#undef имя
```

Обратите внимание на отсутствие символа `;` в конце строки. Если его добавить, то компилятор воспримет его как составную часть `текст_подстановки` что может привести в дальнейшем к трудно уловимым ошибкам или ошибкам компиляции. Хотя такая ошибка чаще всего безобидная (после каждой макроподстановки неожиданно появится “пустая” строка), но в некоторых случаях (например, когда одно выражение разбивается на несколько макросов) это может привести к серьезным неприятностям.
Напомним, что **константами** в языке C/C++ называется один из двух базовых типов операндов. Операнды такого типа **принимают свои значения перед началом выполнения программы и**

сохраняют эти значения до завершения программы. Второй базовый тип операндов - переменные - был нами подробно рассмотрен на одном из предыдущих занятий. Самое существенное отличие констант от переменных вытекает из их определения и состоит в том, что им нельзя присвоить по ходу выполнения программы никакое значение, т.е. они не могут **самостоятельно** присутствовать слева от оператора присваивания (=). Иными словами, константы - это **rvalue**. Чтобы определить пользовательскую константу в простейшем случае служит директива `#define`.

Если по директиве `#define` подменяется целое выражение (или часть выражения), то такая замена называется **макроподстановкой**, или, для краткости, **макросом**. Далее `текст_подстановки` замещает имя в исходном файле везде, кроме ситуаций, когда имя встречается внутри строковой константы (но не представляет собой всю константу целиком) или внутри комментария, о чем подробнее поговорим далее на примерах. `Текст_подстановки` может быть пустым, в таком случае имя макроса пропускается в исходном тексте, не замещаясь ничем. Вводя и отменяя макроподстановки можно легко менять ход программы внутри одного модуля.

Текст директивы препроцессора (но не область ее действия) простирается до тех пор, пока не встретится завершающий директиву символ новой строки (кроме случаев с обратным слешем, рассмотренных выше). Другими словами, длина директивы чаще всего ограничена одной строкой. Каждая строка директивы (т.е. логическая строка) состоит из трех частей. Первая часть — это сама директива (сейчас мы говорим о `#define`). Вторая часть — выбранное программистом имя макроподстановки. Некоторые макросы, как в приведенном далее примере, представляют значения. Они называются объектными макросами (существуют также функциональные). Имя макроса не должно содержать пробелов. На макросы распространяются правила именования переменных: разрешены только буквы, цифры и символ подчеркивания, а первым символом не должна быть цифра. Третья часть (остаток строки) называется списком замены или телом макроса. Когда препроцессор обнаруживает в программе имя определенного ранее директивой `#define` макроса, он почти всегда заменяет его телом. Этот процесс перехода от макроса к подставляемому итоговому значению называется расширением. В строке `#define` могут быть указаны стандартные комментарии; ранее уже говорилось о том, что до начала работы препроцессора любой комментарий заменяется пробелом.

Вот пример широко распространенной на практике **подстановки именованной константы**:

```
#define ARRAYSZ 10
float x[ARRAYSZ];
```

Кроме того, если далее по тексту будут обращения вроде:

```
for (int i = 0; i < ARRAYSZ; i++) x[i] = x[i + 1];
```

то не нужно будет в дальнейшем для изменения размера массива просматривать весь текст программы в поиске таких обращений, достаточно будет лишь в одной строке переопределить:

```
#define ARRAYSZ 100
```

Обратите внимание в данном примере также на то, что имя именованной константы `ARRAYSZ` набрано из символов верхнего регистра (заглавными буквами). Это не является обязательным требованием, но является признаком хорошего тона в C/C++ - давать имена именованными константами и макросам, состоящие только из заглавных букв.

Вот пример **макроподстановки** тернарной операции:

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

Здесь `MAX(x, y)` – параметризованный макрос, который заменяется далее по тексту программы значением одного из своих параметров – `x` или `y`, в зависимости от того, который из них больше. Обратите внимание на несколько существенных моментов:

- у параметризованных макросов **не допускается** пробел между именем макроса и левой скобкой (открывающей список параметров). Иначе получится совсем другая макроподстановка, не параметризованная, а `текст_подстановки` которой начинается с левой скобки. И компилятор, скорее всего, не посчитает это ошибкой;

- при вызове макроса реальные аргументы (т.е. `x`, `y`, которые могут встретится в функции, в которой размещена макроподстановка) замещают в `текст_подстановки` формальные параметры (т.е. `x`, `y`, которые указаны в круглых скобках при `MAX(x,y)`). Чтобы быть уверенным, что такой замены не произойдет, рекомендуем заключать формальные параметры в теле `текст_подстановки` в круглые скобки, как в примере выше;
- Дополнительно напомним, что не рекомендуем завершать `текст_подстановки` символом `;`. Дело в том, что макрос `MAX(x, y)` из примера скорее всего будет вызываться в программе следующей строкой: `z = MAX(x,y);` что после макроподстановки с завершающим символом `;` превратится в строку: `z = x > y ? x : y; ;`, что в данном примере, конечно, не ошибка, просто дополнительный “пустой” оператор в тексте программы, признак плохого тона, но могло быть и хуже.

Далее покажем на примерах некоторые возможности и свойства директивы `#define`. Также будем использовать в дальнейших примерах директиву `#include`, она подключает к программе внешние по отношению к ней (библиотечные) файлы.

```
#include <cstdio>
// при желании можно использовать комментарии
#define TWO 2
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уайльд"
// обратная косая черта переносит определение на следующую строку
#define FOUR TWO*TWO
#define PX printf("X = %d.\n", x) // Не допускать пробелов между  printf и
открывающей (
#define FMT "X = %d.\n"

int main() {
   int x = TWO;
   PX;
   x = FOUR;
   printf(FMT, x);
   printf("%s\n", OW);
   printf("TWO: OW\n");
   return 0;
}
```

Давайте запустим программу и посмотрим, как она работает:

```
X = 2.
X = 4.
Логика - последнее убежище лишенных воображения. - Оскар Уайльд
TWO: OW
```

Вот, что здесь происходит. Оператор

```
int x = TWO;
// преобразуется в
int x = 2;
// поскольку вместо TWO было подставлено 2. Затем оператор
PX;
// приобретает следующий вид:
printf("X = %d.\n", x);
```

Подстановка осуществилась для всего оператора целиком. Здесь вы видите, что макрос может представлять любую строку, даже целое выражение С. Однако отметим, что это константная строка; макрос `PX` будет выводить только значение переменной по имени `x`. В следующей строке также демонстрируется новый прием. Может показаться, что `FOUR` будете заменено `4`, но на самом деле процесс был немного другим:

```
// строка
x = FOUR;
// преобразуется в строку
x = TWO*TWO;
// которая затем становится следующей:
x = 2*2;
```

На этом процесс расширения макроса завершен. Действительное умножение происходит не во время работы препроцессора, а на этапе компиляции, поскольку компилятор С на этой стадии вычисляет все константные выражения (т.е. выражения, которые содержат только константы). Препроцессор не выполняет вычислений, а просто совершенно буквально производит указанные с помощью директив подстановки. Обратите внимание, что определение макроса может включать другие макросы. (Некоторые компиляторы такую вложенность макросов не поддерживают). Следующая строка

```
printf (FMT, x);
// приводится к виду:
printf("X = %d.\n", x);
```

потому что `FMT` заменяется соответствующей строкой. Такой подход может оказаться удобным при наличии длинной управляющей строки, которую приходится применять несколько раз. Вместо этого можно было поступить следующим образом:

```
const char *fmt = "X = %d.\n";
```

Затем можно использовать `fmt` в качестве управляющей строки для `printf()`.

В следующей строке `OW` заменяется соответствующей строкой. Двойные кавычки делают строку замещения символьной строковой константой. Компилятор сохранит ее в массиве с завершающим нулевым символом. Таким образом, директива

```
#define CHAR 'Z'
// определяет символьную константу Z, а директива
#define CHAR "Z"
// определяет символьную строку: Z\0.
```

В этом примере мы применяли обратную косую черту непосредственно перед концом строки, распространяя директиву на следующую строку:

```
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уайльд"
// Обратите внимание, что вторая строка выровнена влево. Если бы директива
имела вид:
#define OW "Логика - последнее убежище лишенных\
            воображения. - Оскар Уайльд"
// то вывод был бы таким:
Логика - последнее убежище лишенных            воображения. - Оскар Уайльд
```

Пробелы с начала строки и до слова "воображения" считаются частью строки. Обычно где бы препроцессор ни обнаружил в программе один из макросов, он заменяет его литерально эквивалентным текстом замены. Если строка замещения содержит вложенные макросы, они также заменяются. Единственным исключением при замене является ситуация, когда найденный макрос заключен в двойные кавычки. Поэтому строка

```
printf("TWO: OW");
```

выводит текст: **`TWO: OW`** буквально вместо того, чтобы вывести

```
2: Логика - последнее убежище лишенных воображения. - Оскар Уайльд
```

Для вывода показанной строки понадобится следующий код:

```
printf("%d: %s\n", TWO, OW);
```

Здесь имена обоих макросов находятся за пределами двойных кавычек.

Символические константы

Когда должны использоваться символические константы? Вы должны их применять для большинства числовых констант. Если число представляет собой некоторую константу, участвующую в вычислениях, то символическое имя сделает ее назначение более понятным. Если число является размером массива, то символическое имя упростит его будущее изменение и корректировку границ выполнения циклов. Если число представляет собой системный код, такой как `EOF`, то символическое имя увеличит степень переносимости программы; понадобится изменять только одно определение `EOF`. Мнемонические значения, изменяемость и переносимость — все эти характеристики делают символические константы заслуживающими внимания. Однако ключевое слово `const`, которое теперь поддерживается в C, обеспечивает более гибкий способ создания констант. С помощью `const` можно создавать глобальные и локальные константы, числовые константы, константы в форме массивов и константы в виде структур. С другой стороны, константы-макросы могут использоваться для указания размеров стандартных массивов, а также инициализирующих значений для величин `const`.

```
#define LIMIT 20
const int LIM = 50; // допустимо

static int data1[LIMIT] ;
static int data2[LIM]; // не обязательно должно быть допустимым

const int LIM2 = 2 * LIMIT; // допустимо

const int LIM3 = 2 * LIM; // не обязательно должно быть допустимым
```

Обратите внимание на комментарий “не обязательно должно быть допустимым". В языке C предполагается, что размер массива для неавтоматических массивов задается целочисленным константным выражением, т.е. комбинацией целочисленных констант наподобие `50`, констант из перечислений и выражений `sizeof`. Значения, объявленные с применением `const`, сюда не входят. (В этом отношении C отличается от C++, где значения `const` могут быть частью константных выражений.) Тем не менее, реализация компилятора может адаптировать другие формы константных выражений. В результате, к примеру, GCC 4.7.3 не примет объявление для `data2`, но Clang 4.6 - примет.

Лексемы

Формально тело макроса должно быть строкой лексем, а не строкой символов. Лексемы препроцессора C — это отдельные “слова” в теле определения макроса. Они отделяются друг от друга пробельными символами. Например, определение

```
#define FOUR 2*2
// имеет одну лексему -- последовательность 2*2, но определение
#define SIX 2 * 3
// содержит три лексемы: 2, * и 3.
```

Строки символов и строки лексем отличаются в том, как трактуются последовательности из множества пробелов. Рассмотрим следующее определение:

```
#define EIGHT 4 * 8
```

Препроцессор, который интерпретирует тело макроса как строку символов, вместо `EIGHT` подставляет `4 * 8`. То есть дополнительные пробелы будут частью замены, но препроцессор, который интерпретирует тело как строку лексем, заменит `EIGHT` тремя лексемами, разделенными одиночными пробелами: `4 * 8`. Другими словами, интерпретация в виде строки символов трактует пробелы как часть тела, а интерпретация в виде строки лексем считает пробелы разделителями между лексемами внутри тела. На практике некоторые компиляторы C рассматривают тела макросов как строки, а не как лексемы. Это различие имеет практическое значение только для более сложных случаев использования по сравнению с приведенными здесь. Кстати, в компиляторе C принята более сложная трактовка лексем по сравнению с препроцессором. Компилятор понимает правила языка C и не обязательно требует наличия пробелов для отделения лексем друг от друга. Например, компилятор C будет интерпретировать `2 * 2` как три лексемы, поскольку он выясняет, что `2` является константой, а `*` — операцией. Во избежание ошибок и разночтений при работе с различными компиляторами рекомендуем явно определять границы лексем круглыми скобками, например так `#define EIGHT (4 * 8)`

Использование аргументов в директиве #define

С помощью аргументов можно создавать функциональные макросы, которые выглядят и действуют во многом подобно обычным функциям. Макрос с аргументами очень похож на функцию, потому что аргументы заключаются в круглые скобки. Определения функциональных макросов имеют один или более аргументов в скобках, и эти аргументы затем присутствуют в выражении замены

```
#define SQUARE(X) X*X
// Оно может применяться в программе следующим образом:
z = SQUARE(2);
```

Оператор выглядит похожим на вызов функции, хотя поведение макроса не обязательно будет идентичным. Далее иллюстрируется использование этого и второго макроса. В некоторых примерах также обращается внимание на возможные ловушки, поэтому читайте их внимательно.

```
#include <cstdio>
#define SQUARE(X) X * X
#define PR(X) printf("Результат: %d.\n", X)

int main (int argc, char** argv) {
    int x = 5;
    int z;
    printf("x = %d\n", x); z = SQUARE(x);
    printf("Вычисление SQUARE(x): "); PR(z);
    z = SQUARE(2);
    printf("Вычисление SQUARE(2): ");
    PR(z);
    printf("Вычисление SQUARE(x + 2): "); PR(SQUARE(x + 2));
    printf("Вычисление 100 / SQUARE(2): "); PR(100 / SQUARE(2));
    printf("x = %d.\n", x); printf("Вычисление SQUARE(++x): "); PR(SQUARE(++x));
    printf("После инкрементирования x = %x.\n", x) ;
    return 0;
}
```

Макрос SQUARE имеет следующее определение:

```
#define SQUARE(X) X*X
```

Здесь `SQUARE` — идентификатор макроса, `X` в `SQUARE (X)` — аргумент макроса, а `X*X` — список замены. Каждое вхождение `SQUARE(X)` в листинге заменяется на `X*X`. Отличие данного примера от предыдущих состоит в возможности использования в макросе любых символов помимо `X`. Символ `X` в определении макроса заменяется символом, который указан при вызове макроса в программе. Таким образом, `SQUARE(2)` заменяется `2*2`, так что `X` действительно играет роль аргумента. Однако, как вскоре будет показано, аргумент макроса не работает в точности как аргумент функции. Ниже представлены результаты выполнения программы. Обратите внимание, что некоторые вычисления дают результат, отличающийся от того, что можно было ожидать. На самом деле ваш компилятор может даже выдать не такой результат, как приведенный в предпоследней строке:

```
x = 5
Вычисление SQUARE(x): Результат: 25.
Вычисление SQUARE(2): Результат: 4.
Вычисление SQUARE(x + 2): Результат: 17.
Вычисление 100 / SQUARE(2): Результат: 100.
x = 5.
Вычисление SQUARE(++x): Результат: 42.
После инкрементирования x = 7.
```

Первые две строки вполне предсказуемы, но затем встречается несколько странных результатов. Вспомните, что `x` имеет значение `5`. Это может привести к предположению, что `SQUARE(x + 2)` должно быть `7 * 7`, или `49`, но выводится `17` — простое число, которое определенно не может быть квадратом чего бы то ни было. Причина такого вводящего в заблуждение вывода связана с тем, что, как уже говорилось, препроцессор не выполняет вычислений, а просто заменяет последовательности символов. Где бы в определении не появлялось `x`, препроцессор подставит вместо `x` символы `x + 2`.

```
// поэтому
x * x
// принимает вид:
x + 2 * x + 2
// Единственным умножением является 2 * x.
```

Если `x` равно `4`, выражение вычисляется следующим образом:

```
5 + 2 * 5 + 2 = 5 + 10 + 2 = 17
```

Этот пример подчеркивает важное отличие между вызовом функции и вызовом макроса. При вызове функции ей передается значение аргумента во время выполнения программы. При вызове макроса лексема аргумента передается в программу перед компиляцией; это другой процесс, происходящий в другое время. Можно ли исправить определение, чтобы вызов `SQUARE(x + 2)` выдавал `49`? Конечно. Нужны просто дополнительные круглые скобки:

```
#define SQUARE(x) (x)*(x)
```

Теперь `SQUARE(x + 2)` превращается в `(x + 2) * (x + 2)`, и вы получите ожидаемое умножение, поскольку круглые скобки останутся в заменяющей строке. Однако это не решает всех проблем. Рассмотрим события, которые приводят к тому, что следующая строка:

```
100 / SQUARE(2)
```

в выводе преобразуется к виду

```
100 / 2 * 2
```

Согласно приоритетам операций, выражение вычисляется слева направо: `(100 / 2) * 2`, или `50 * 2`, или `100`. Для устранения путаницы `SQUARE(x)` необходимо определить так:

```
#define SQUARE(x) (x * x)
```

В результате это дает ожидаемое `100 / (2 * 2)`, что в итоге вычисляется как `25`. В следующем определении учтены ошибки обоих примеров:

```
#define SQUARE(x) ((x) * (x))
```

Из всего продемонстрированного можно извлечь такой урок: применяйте столько круглых скобок, сколько необходимо для того, чтобы обеспечить корректный порядок выполнения операций. Но даже эти меры предосторожности не спасают от ошибки в последнем примере:

```
SQUARE(++x)
// В результате получается:
++x * ++x
```

Здесь `x` инкрементируется дважды — один раз до операции умножения и один раз после нее:

```
++x * ++x = 6 * 7 = 42
```

Из-за того, что выбор конкретного порядка выполнения операций оставлен за разработчиками реализаций, некоторые компиляторы генерируют умножение `7 * 6`. Есть компиляторы, которые могут инкрементировать оба операнда перед умножением, выдавая в результате `7 * 7`. На самом деле вычисление этого выражения приводит к ситуации, которая в стандарте называется неопределенным поведением. Тем не менее, во всех этих случаях `x` начинает со значения `5` и заканчивает значением `7`, хотя код выглядит так, будто инкрементирование происходит только один раз.
Простейшее решение этой проблемы — избегать использования `++x` как аргумента макроса, а также в операциях типа `++x + x++` и подобных. Вообще лучше не применять в макросах операции инкремента и декремента. Следует отметить, что выражение `++x` будет работать в качестве аргумента функции, т.к. оно вычисляется как значение `6`, которое затем передается функции.

Создание строки аргументов макроса: операция

Рассмотрим следующий функциональный макрос:

```
#define PSQR(X) printf("Квадрат X равен %d.\n", ((X) * (X)))
// Предположим, что этот макрос используется следующим образом:
PSQR(8);
// Вот каким будет вывод:
Квадрат X равен 64.
```

Обратите внимание, что определение `X` в строке, заключенной в двойные кавычки, трактуется как обычный текст, а не лексема, которую можно заменить. Представим, что вы хотите поместить аргумент макроса в строку. Язык C позволяет сделать это. Внутри заменяющей части функционального макроса символ `#` становится операцией препроцессора, которая преобразует лексемы в строки. Пусть `x` является параметром макроса, тогда `#x` — это имя параметра, преобразованное в строку `"x"`.

```
#define PSQR(x) printf("Квадрат " #x " равен %d.\n", ((x)* (x)))
int main() {
      int y = 5;
      PSQR(y);
      PSQR(2 + 4);
      return 0;
}
// Вывод выглядит следующим образом:
Квадрат y равен 25.
Квадрат 2 + 4 равен 36.
```

В первом вызове макроса `#x` заменяется строкой `"y"`, а во втором вызове вместо `#x` подставляется `"2 + 4"`. Конкатенация строк затем объединяет эти строки с другими строками в операторе `printf()` для получения финальной строки. Например, первый вызов макроса дает следующий оператор:

```
printf("Квадрат " "y" " равен %d.\n",((y)*(y)));
```

После этого конкатенация объединяет три расположенные рядом строки в одну: `"Квадрат у равен %d.\n"`

Средство слияния препроцессора: операция

Подобно `#`, операция `##` может применяться в заменяющей части функционального макроса. Вдобавок она может использоваться в заменяющей части объектного макроса. Операция `##` объединяет две лексемы в одну. Предположим мы могли бы записать такое определение:

```
#define XNAME(n) x ## n
// Тогда макрос
XNAME(4)
// будет расширен следующим образом:
x4
```

Ниже приведён более полный пример с выводом

```
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);

int main() {
    int XNAME(1) = 14; // превращается в int x1 = 14;
    int XNAME(2) = 20; // превращается в int x2 = 20;
    int x3 = 30;
    PRINT_XN(1); // превращается в printf("x1 = %d\n", x1) ;
    PRINT_XN(2); // превращается в printf("x2 = %d\n", x2);
    PRINT_XN(3); // превращается в printf("x3 = %d\n", x3) ;
    return 0;
}
// вывод:
x1 = 14
x2 = 20
x3 = 30
```

Обратите внимание, что в макросе `PRINT_XN()` операция `#` используется для объединения строк, а операция `##` — для объединения лексем в новый идентификатор.

Макросы с переменным числом аргументов:

Некоторые функции, скажем, `printf()`, принимают переменное количество аргументов. Заголовочный файл `stdvar.h` предоставляет инструменты для создания определяемых пользователем функций с переменным числом аргументов. В C99/C11 тоже самое сделано и для макросов. Идея заключается в том, что последний аргумент в списке арументов для определения макроса может быть троеточием. Если это так, то в заменяющей части может применяться предопределенный макрос `__VA_ARGS__`,который будет подставлен вместо троеточия. Для примера рассмотрим следующее определение:

```
#define PR(...) printf(__VA_ARGS__)
// Предположим, что в программе содержатся вызовы макроса вроде показанных
ниже:
PR("Здравствуйте");
PR("Вес = %d, доставка = $%.2f\n", wt, sp);
// Для первого вызова __VA_ARGS__ расширяется в один аргумент:
"Здравствуйте"
// Для второго вызова он расширяется в три аргумента:
"вес = %d, доставка = $%.2f\n", wt, sp
// Таким образом, результирующий код выглядит так:
printf("Здравствуйте" );
```

```
printf("вес = %d, доставка = $%.2f\n", wt, sp);
```

Далее приведен более сложный пример, в котором используются конкатенация строк и операция #.

```
#include <cstdio>
#include <cmath>

#define PR(X, ...) printf("Сообщение " #X ": " __VA_ARGS__)
int main (void) {
    double x = 48;
    double y;
    y = sqrt (x);
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4 f\n", x, y); return 0;
}
```

В первом вызове макроса `X` имеет значение `1`, так что `#X` становится `"1"`.

```
// В результате получается следующее расширение:
printf("Сообщение " "1" ": " "x = %g\n", x);
// Затем осуществляется конкатенация строк, сокращая вызов к такому виду:
printf("Сообщение 1: x = %g\n", x);
// В итоге имеем показанный ниже вывод:
Сообщение 1: x = 48
Сообщение 2: x = 48.00, y = 6.9282
```

Не забывайте, что троеточие должно быть последним аргументом макроса.

Выбор между макросом и функцией

Многие задачи могут быть решены за счет применения макроса с аргументами либо функции. Что должно использоваться? Здесь нет каких-то строго определенных правил, но есть ряд соображений, которые следует принимать во внимание:

- Макросы несколько сложнее в применении, чем обычные функции, т.к. макросы могут иметь неожиданные побочные эффекты, если вы проявите неосмотрительность. Некоторые компиляторы ограничивают определение макроса одной строкой, и вероятно лучше придерживаться этого ограничения, даже если в вашем компиляторе оно отсутствует.
- Выбор между макросом и функцией связан с достижением компромисса между быстродействием и размером кода. Макрос генерирует встраиваемый код, т.е. в программу помещаются операторы. Если макрос используется 20 раз, в программу вставляется 20 строк кода. Когда 20 раз применяется функция, в программе все равно содержится только одна копия ее операторов, что уменьшает размер кода. С другой стороны, поток управления программы должен переходить туда, где находится функция, и затем возвращаться в место ее вызова. Этот процесс отнимает больше времени, чем выполнение встраиваемого кода.
- Преимущество макросов в том, что они не заботятся о типах переменных. Причина связана с тем, что они имеют дело со строками символов, а не действительными значениями. Таким образом, макрос `SQUARE(x)` может с одинаковым успехом использоваться с типом `int` или `float`.
- Программисты обычно применяют макросы для простых функций, таких как перечисленные ниже:

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

```
#define ABS(X) ((X) < 0 ? - (X) : (X))
#define ISSIGN(X) ((X) == '+' || (X) == '-' ? 1 : 0)
```

Последний макрос имеет значение 1, или истинное, если x является символом алгебраического знака). Далее указано несколько моментов, о которых не следует забывать.

- Помните, что имя макроса не должно содержать пробелов, но пробелы в некоторых случаях допускаются в замещающей строке. В ANSI C разрешены пробелы в списке аргументов.
- Заключайте в скобки каждый аргумент и определение в целом. Это гарантирует корректное группирование элементов в выражении
- Используйте прописные буквы для имен функциональных макросов. Данное соглашение не так широко распространено, как применение прописных букв в именах константных макросов. Тем не менее, одна из веских причин использования прописных букв связана с тем, что это напоминает вам о возможных побочных эффектах макросов.
- Если вы намерены применять макрос вместо функции главным образом для ускорения работы программы, сначала попытайтесь выяснить, обеспечит ли это заметный выигрыш. Макрос, который используется в программе один раз, не приведет к значительному улучшению скорости ее выполнения. Макрос, находящийся внутри вложенного цикла, является намного лучшим кандидатом для ускорения работы программы. Многие системы предлагают профилировщики программ, которые помогают выявлять фрагменты кода, требующие наибольшего времени выполнения.
- Предположим, что вы разработали несколько нужных вам функциональных макросов. Должны ли вы набирать их каждый раз, когда пишется новая программа? Нет, если вы будете помнить о директиве `#include`

Директива препроцессора #undef

Мы только что подробно рассмотрели применение директивы препроцессора `#define`. Директива `#undef` отменяет заданное определение `#define`. Предположим, что есть следующее определение:

```
#define LIMIT 400
// Тогда директива:
#undef LIMIT
// отменит это определение.
```

Затем имя LIMIT можно переопределить, назначив новое значение. Попытка отменить определения LIMIT не вызовет ошибки даже в случае, если предварительное определение не делалось. Если вы хотите использовать некоторое имя, но не уверены в том, что оно не было определено ранее, на всякий случай его определение можно попытаться отменить.

Когда препроцессор встречает в какой-то директиве идентификатор, он считает его определенным или неопределенным. При этом определенный означает, что идентификатор определен директивой препроцессора. Если идентификатор является именем макроса, созданного ранее директивой `#define` в том же файле, и он не отменялся посредством `#undef`, то идентификатор определен. Имейте ввиду, что позиция директивы `#define` в файле будет зависеть от местоположения директивы `#include`, если макрос введен директивой `#define` из заголовочного файла.

Несколько предопределенных макросов, таких как `__DATE__` и `__FILE__` всегда считаются определенными, причем их определение не может быть отменено. Если идентификатор — не макрос, а, скажем, переменная с областью действия на уровне файла, то с точки зрения препроцессора он не определен. Определенным может быть объектный макрос, включая пустой макрос, или функциональный макрос:

```
#define LIMIT 1000
// идентификатор LIMIT определен
#define GOOD
// идентификатор GOOD определен
```

```
#define A(X) ((-(X))*(X))
// идентификатор A определен
int q;  // идентификатор q - не макрос, поэтому не определен
#undef GOOD
// идентификатор GOOD не определен
```

Директива #error.

Является своего рода аналогом контрольной точки для поиска ошибок при отладке программы. А именно: приводит к выводу компилятором диагностического сообщения которое включает любой текст, указанный в директиве. Если это возможно, процесс компиляции должен приостановиться.
Директива `#error` используется, как правило, в сочетании с “директивами условий” `#if`, `#elif`, `#else`, `#endif`, `#ifdef`, `#ifndef`.
Синтаксис директивы предельно прост:

```
#error "строка"
```

Использоваться она может, например, так:

```
#if ARRAYSZ % 16 != 0
#error "Размер массива должен быть кратным 16"
// или:
#if __x86_64__ != 1
#error "Необходимо скомпилировать программу на 64-разрядной ОС"
#endif
```

Процесс компиляции не проходит, когда компилятор запускается на 32-разрядной ОС, и завершается успешно, когда применяется архитектура x86_64.

Директива #pragma.

У современных компиляторов языка Си существуют настройки, которые можно модифицировать с помощью аргументов командной строки или через меню среды. Каждый компилятор имеет собственный набор таких настроек. Они могут применяться, например, для управления объемом памяти, выделяемой под автоматические переменные, для установки уровня строгости при проверке ошибок или для включения нестандартных языковых средств. Стандарт C99 поддерживает директиву препроцессора `#pragma`. Она позволяет передавать компилятору “псевдокомментарии”, имена которых должны быть знакомы компилятору или установленным библиотекам (обязательно сверяйтесь с документацией на свой компилятор и устанавливаемые библиотеки).

Псевдокомментарии директивы `#pragma` можно рассматривать как инструкции для компилятора, помещенные в исходный код программы. Например, во время разработки стандарта C99 на него ссылались как на C9X, и в одном из компиляторов использовалась следующая директива для включения поддержки этого стандарта:

```
#pragma c9x on
```

Псевдокомментарии, которые не распознаются данным конкретным компилятором, игнорируются. Псевдокомментарии могут быть включены внутри исходного файла, но не внутри функции, и имеют силу от точки включения и до точки, пока другой псевдокомментарий не изменит его статус, либо до конца файла.
Вот наиболее известные на практике псевдокомментарии – инструкции компилятору:
`#pragma OPTIMIZE{ON}` - включает оптимизацию исходного кода на этапе компиляции;
`#pragma OPTIMIZE{OFF}` - отключает оптимизацию исходного кода на этапе компиляции;
`#pragma COPYRIGHT “строка”` - помещает в объектный модуль и исполняемый файл программы “строку” об авторском праве разработчика на данную программу;

`#pragma VERSIONID "строка"` - ограничивает набор директив компилятора набором директив той версии компилятора, которая указана в "строка". "Строка" помещается в объектный файл, созданный компилятором, и может быть разной для различных частей одного и того – же программного модуля.

Предопределенные макроподстановки

Они есть во всех компиляторах (мы приведём только наиболее часто используемые). Эти макросы имеют специальное значение, они не могут быть переопределены. Вот примеры таких макроподстановок:

- `__DATE__` - Дата компиляции файла в формате mm dd yyyy, например, Aug 24 2020;
- `__FILE__` - Имя компилируемого файла;
- `__LINE__` - Номер текущей строки исходного файла (целочисленная константа);
- `__STDC_VERSION__` - Для C99 установлен в 1999011; для C11 установлен в 201112L, работает только в C
- `__TIME__` - Время компиляции файла в формате hh:mm:ss.
- `_WIN64`, `_WIN32` - константа индикатор сборки в среде Windows
- `LINUX`, `__linux__` - константа-индикатор сборки Linux
- `__APPLE__` - константа обозначающая любое устройство от Apple
- `__cplusplus` - константа-индикатор C++ в Visual Studio Compiler

Следует отметить, что стандарт C99 предоставляет предопределенный идентификатор `__func__`, который расширяется до строкового представления имени содержащей его функции. По этой причине данный идентификатор должен иметь область действия в пределах функции, в то время как макросы по существу располагают областью действия на уровне файла. Таким образом, `__func__` является предопределенным идентификатором языка C, а не предопределенным макросом.

```
void why_me();

int main() {
   printf("Файл: %s.\n", __FILE__);
   printf("Дата: %s.\n", __DATE__);
   printf("Время: %s.\n", __TIME__);
   printf("Это строка %d.\n", __LINE__);
   printf("Это функция %s\n", __func__);
   printf("Это Mac OS X? %d\n", __APPLE__);
   why_me();
   return 0;
}

void why_me() {
   printf("Это функция %s\n", __func__);
   printf("Это строка %d.\n", __LINE__);
}
```

Вот как выглядит вывод, полученный в результате запуска программы:

```
Файл: /Users/ivan-igorevich/Desktop/plgrnd/cplgr/src/main.cpp.
Дата: Jan  6 2021.
Время: 13:24:43.
Это строка 11.
Это функция main
Это Mac OS X? 1
Это функция why_me
Это строка 20.
```

Условная компиляция.

Позволяет компилировать только определенные части исходного кода программы в соответствии с выполнением или невыполнением на этапе препроцессора “директив условий”

```
#if
#elif
#else
#ifdef
#ifndef
#endif
```

Директивы `#if`, `#ifdef`, `#ifndef` отмечают начало фрагмента кода, который будет использован в условной компиляции. Синтаксис этих директив:

```
#if  константное_выражение
// фрагмент кода который должен компилироваться, если константное выражение
дает значение "истина" (не равно нулю). В противном случае компилятор
пропускает этот фрагмент кода
#endif

#ifdef  идентификатор
// фрагмент кода который должен компилироваться, если идентификатор определен
ранее
#endif

#ifndef  идентификатор
// фрагмент кода который должен компилироваться, если идентификатор не
определен ранее
#endif
```

Завершается фрагмент кода директивой `#endif`. Весь фрагмент кода от директив `#if`, `#ifdef`, `#ifndef` до соответствующей директивы `#endif` (включая сами эти директивы) принято называть **условным блоком**. Внутри условного блока могут находиться директивы `#elif` и `#else`. Синтаксис условного блока `#if` в этом случае:

```
#if константное_выражение
// фрагмент кода который должен компилироваться, если константное выражение
дает значение "истина" (не равно нулю). В противном случае компилятор
пропускает этот фрагмент кода
#else
// фрагмент кода который должен компилироваться, если константное выражение
дает значение "ложь" (равно нулю). В противном случае компилятор пропускает
этот фрагмент кода
#endif

#if константное_выражение_1
// фрагмент кода который должен компилироваться, если константное_выражение_1
"истина" (не равно нулю). В противном случае компилятор пропускает этот
фрагмент кода
#elif константное_выражение_2
// фрагмент кода который должен компилироваться, если константное_выражение_1
```

```
"ложь", а константное_выражение_2 "истина". В противном случае компилятор
пропускает этот фрагмент кода
#elif константное_выражение_n
// фрагмент кода, который должен компилироваться, если все предыдущие
константные выражения ложны, а это истинно. Если и оно ложно, то этот фрагмент
кода пропускается.
#else
// фрагмент кода, который компилируется, если все предыдущие константные
выражения ложны.
#endif
```

Как видим, в последнем случае применение директивы `#elif` есть ни что иное, как сокращенный вариант использования вложенных один в другой условных блоков `#if`. Допускается также вложение одного в другой условных блоков `#ifdef`, `#ifndef`, а также вложение блоков `#if`, `#ifdef`, `#ifndef` друг в друга в любом сочетании. Константное_выражение в директиве `#if` является обычным константным выражением языка Си, за исключением того, что в этом выражении нельзя использовать операции `sizeof`, приведение типов и константы перечисляемого типа. В большинстве компиляторов все необходимые вычисления в константном_выражении производятся в формате `long int`. Все имена макросов, появляющиеся в константном_выражении, замещаются подстановками до вычисления константного_выражения.
Часто в качестве константного_выражения при директиве `#if` выступает операция `defined`, или даже последовательность таких операций. Например:

```
#if !defined(THIS_FILE)
static char THIS_FILE[] = __FILE__;
#endif
```

В этом примере проверяется, объявлено ли внутри модуля программы имя файла, содержащего этот модуль, и если такого объявления еще не было, объявляется массив `THIS_FILE[]`, в который предкомпилятор записывает имя компилируемого файла (макрос __FILE__). Очевидно, что полным аналогом этого примера является:

```
#ifndef  THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

обратите внимание на то, что скобки вокруг идентификатора не обязательны.
Рассмотрим еще несколько конкретных примеров условной компиляции:

```
#ifdef MAVIS
// выполняется, если идентификатор MAVIS определен
#include "horse.h"
#define STABLES 5
#else
// выполняется, если идентификатор не определен
#include "cow.h"
#define STABLES 15
#endif
```

Директива `#ifdef` говорит о том, что если следующий за ней идентификатор (MAVIS) был определен препроцессором, необходимо обработать все директивы и скомпилировать весь код С до следующей директивы `#else` или `#endif` в зависимости от того, что встретится раньше. Если предусмотрена директива `#else`, то должен быть обработан весь код между `#else` и `#endif`, когда идентификатор не определен. Директива `#ifndef` может использоваться совместно с директивами `#else` и `#endif` тем же способом, что и `#ifdef`. Директива `#ifndef` выясняет, не определен ли следующий за ней идентификатор; она представляет собой инверсию директивы `#ifdef`. Эта директива часто

применяется для определения константы, если она еще не была определена. Обычно такая конструкция используется для предотвращения множественных определений одного и того же макроса при включении нескольких заголовочных файлов, каждый из которых может содержать определение. В этом случае определение в первом заголовочном файле становится активным, а последующие определения в других заголовочных файлах игнорируются.
Форма `#ifdef` `#else` во многом подобна оператору `if`/`else` языка C. Основное отличие в том, что препроцессор не распознает фигурные скобки как метод обозначения блока, поэтому для пометки блоков директив используются директивы `#else` (если есть) и `#endif` (должна присутствовать). Такие условные структуры могут быть вложенными. Как иллюстрируется далее, эти директивы можно применять также для пометки блоков операторов C.

```
#define JUST_CHECKING
#define LIMIT 4
int main() {
   int i;
   int total = 0;
   for (i = 1; i <= LIMIT; i++) {
       total += 2 * i * i + 1;
       #ifdef JUST_CHECKING
         printf("i=%d, промежуточная сумма = %d\n", i, total);
       #endif
   }
   printf("Итоговая сумма = %d\n", total);
   return 0;
}
```

В результате компиляции и выполнения программы будет получен следующий вывод:

```
i=1, промежуточная сумма = 3
i=2, промежуточная сумма = 12
i=3, промежуточная сумма = 31
i=4, промежуточная сумма = 64
Итоговая сумма = 64
```

Если опустить определение `JUST_CHECKING` (или поместить его в комментарий либо отменить определение с помощью директивы `#undef`) и повторно скомпилировать программу, отобразится только последняя строка. Таким приемом можно пользоваться, например, при отладке программы. Еще одной возможностью является применение `#ifdef` для выбора альтернативных блоков кода, приспособленных к разным реализациям C и кроссплатформенным сборкам.

Практическое задание

1. Описать макрокоманду, проверяющую, входит ли переданное ей число в диапазон от нуля (включительно) до переданного ей второго аргумента (исключительно)
2. Описать макрокоманду, получающую доступ к элементу двумерного массива (организованного динамически) посредством разыменования указателей
3. * Описать макрокоманду, возвращающую количество элементов локального массива, принимающую на вход ссылку на массив и его тип

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования C. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.
2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)