The C PreprocessorПрепроцессор C

(GNU Tools for STM32 10.3-2021.10.20211105-1100)

Table of Contents Оглавление

1 Overview Обзор : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 1

1.1 Character sets: Наборы символов : :: : : : : : : : : : : : : : : 1

1.2 Initial processing : Первичная обработка: : :: : : : : : : : : : : : : : : : : : : : : : : 2

1.3 Tokenization : Токенизация: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 4

1.4 The preprocessing language : Язык препроцессинга : : : : : : : : : : : : : : : : : 6

2 Header Files: файла заголовков : : : : : : : : : : : : : : : : : : : : : : : 7

2.1 Include Syntax : Включить синтаксис: : : : : : : : : : : : : : : : : : : : : : : : : : : : : 7

2.2 Include Operation : Операция включения : : : : : : : : : : : : : : : : : : : : : : : : : : 8

2.3 Search Path : Путь поиска: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 9

2.4 Once-Only Headers: Одноразовые заголовки: : : : : : : : : : : : : : : : : : : : : : : : : 9

2.5 Alternatives to Wrapper #ifndef : : : : : : : : : : : : :: : : : : : : : : : : : : : : : : 10

2.6 Computed Includes Вычислено Включает : : : : : : : : : : : : : : : : : : : : : : 10

2.7 Wrapper Headers : Заголовки-оболочки : : : : : : : : : : : : : : : : : : : : : : : : : 11

2.8 System Headers : Системные заголовки : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 12

3 Macros : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 13

3.1 Object-like Macros : Объектно-подобные макросы: : : : : : : : : : : :: : : : : : : : : 13

3.2 Function-like Macros : Функционально-подобные макросы: : : : : :: : : : : : : : 14

3.3 Macro Arguments : : : Аргументы макроса: : : : : : : : : : : : : : : : : : : : : : : : : 15

3.4 Stringizing : : : Строка : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 16

3.5 Concatenation : : : Конкатенация: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 17

3.6 Variadic Macros: : : Макросы Variadic: : : : : : : : : : : : : : : : : : : : : : : : : : : : : 18

3.7 Predefined Macros : : Предустановленные макросы: : : : : : : : : : : : : : : : : : 20

3.7.1 Standard Predefined Macros:Стандартные предопределенные макросы : : 20

3.7.2 Common Predefined Macros : Общие предопределенные макросы: : : : : 22

3.7.3 System-specific Predefined Macros : Системные предопределенные макросы: 33

3.7.4 C++ Named Operators : Именованные операторы C+ : : : : : : : : : : 34

3.8 Undefining and Redefining Macros : Отмена определения и переопределение макросов: 34

3.9 Directives Within Macro Arguments : Директивы внутри аргументов макроса: : : : 35

3.10 Macro Pitfalls : Подводные камни макросов : : : : : : : : : : : : : : : : : : : : : : : : 35

3.10.1 Misnesting : Неправильная вложенность: : : : : : : : : : : : : : : : : : : : : : : 35

3.10.2 Operator Precedence Problems : Проблемы приоритета оператора: : : : 36

3.10.3 Swallowing the Semicolon Проглатывание точки с запятой: : : :: : : : : : 37

3.10.4 Duplication of Side Effects : Дублирование побочных эффектов: : : : : : 37

3.10.5 Self-Referential Macros Самореферентные макросы: : : : : : : : : : : : : : : 38

3.10.6 Argument Prescan : Аргумент Prescan : : : : : : : : : : : : : : : : : : : : 39

3.10.7 Newlines in Arguments : Новые строки в аргументах: : : : : : : : : : : : 40

4 Conditionals : Условия : : : : : : : : : : : : : : : : : : : : : : 40

4.1 Conditional Uses : Условное использование: : : : : : : : : : : : : : : : : : : : : : : : : : 41

4.2 Conditional Syntax : Условный синтаксис : : : : : : : : : : : : : : : : : : : : : : : : : : : : 41

4.2.1 Ifdef : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 41

4.2.2 If : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :: : : : : : : : : : : 42

4.2.3 Defined : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 43

4.2.4 Else : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :: : : : : : : : : : 44

4.2.5 Elif : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 44

4.2.6 \_\_has\_attribute : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 45

4.2.7 \_\_has\_cpp\_attribute : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 45

4.2.8 \_\_has\_builtin : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 45

4.2.9 \_\_has\_include : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 46

4.3 Deleted Code : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 46

5 Diagnostics: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 47

6 Line Control : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 47

7 Pragmas: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 48

8 Other Directives : Другие Директивы : : : : : : : : : : : : 49

9 Preprocessor Output : : : : : : : : : : : : : : : : : : : : : : : : : : 50

10 Traditional Mode: : : : : : : : : : : : : : : : : : : : : : : : : : : : : 51

10.1 Traditional lexical analysis : Традиционный лексический анализ : : : : : 51

10.2 Traditional macros : Традиционные макросы: : : : : : : : : : : : : : : : : : : : : : 52

10.3 Traditional miscellany: Традиционный сборник: : : : : : : : :: : : : : : : : : : : : 53

10.4 Traditional warnings : Традиционные предупреждения: : : : : : : : : : : : : : 54

11 Implementation Details : Детали реализации : : : : 54

11.1 Implementation-defined behavior :Поведение, определяемое реализацией: 54

11.2 Implementation limits : Пределы реализации: : : : : : : : : : : : : : : : : : : : : : 56

11.3 Obsolete Features : Устаревшие функции : : : : : : : : : : : : : : : : : : : : : : : 57

11.3.1 Assertions : Утверждения: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 57

12 Invocation : Вызов: : : : : : : : : : : : : : : : : : : : : : : : : : 58

13 Environment Variables : Переменные среды: : : : : : : 67

GNU Free Documentation License: : : : : : : : : : : : : : : 69

ADDENDUM: How to use this License for your documents : : : : : : : : : 76

Index of Directives : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 77

Option Index : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 78

Concept Index : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : 80

1 Overview

Препроцессор C,часто известный как cpp, — это макропроцессор, который автоматически используется компилятор C,чтобы преобразовать вашу программу перед компиляцией.Он называется макропроцессором.потому что он позволяет вам определять макросы, которые являются краткими сокращениями для более длинных конструкций.Препроцессор C предназначен для использования только с исходными текстами C, C++ и **Objective-C**.код. В прошлом его использовали как обычный текстовый процессор. Он будет задыхаться при вводе который не подчиняется лексическим правилам языка Си. Например, апострофы будут интерпретироваться как начало символьных констант и вызывать ошибки. Кроме того, вы не можете полагаться на то, что он сохранит характеристики ввода, которые не являются существенными для языков семейства **С**. Если **Makefile** является предварительно обработаны, все жесткие вкладки будут удалены, а **Makefile** не будет работать. Сказав это, вы часто можете обойтись без использования **cpp** для программ, которые не написаны на **C**. Другое **Algol-ish** языки программирования часто безопасны (Ада и т. д.). Как и ассемблер, с осторожностью.Режим **‘-traditional**-**cpp’**сохраняет больше пробелов и в остальном является более либеральным.Многих проблем можно избежать, написав комментарии в стиле C или **C++** вместо нативных.языковые комментарии и сохранение макросов простыми.

Везде, где это возможно, вы должны использовать препроцессор, адаптированный к языку, на котором вы пишете. Современные версии ассемблера GNU имеют возможности макросов. Самый высокий уровень программирования языки имеют свой собственный механизм условной компиляции и включения. Если ничего не помогло, попробуйте обычный текстовый процессор, такой как **GNU M4**.

Препроцессоры **C** отличаются некоторыми деталями. В этом руководстве обсуждается препроцессор **GNU C**, который предоставляет небольшой расширенный набор функций стандарта **ISO C**. В режиме по умолчанию препроцессор **GNU C** не выполняет некоторые действия, требуемые стандартом. Это функции, которые редко используются, если вообще используются, и могут вызвать неожиданные изменения в значении программы, которая их не ожидает. Чтобы получить строгий стандарт **ISO C**, вы должны использовать опции **‘-std=c90’, ‘-std=c99’, ‘-std=c11’** или **‘-std=c17’** в зависимости от того, какую версию стандарта вы хотите. Чтобы получить всю обязательную диагностику, вы также должны использовать **‘-pedantic’**.

См. Глава 12 [Вызов], стр. 58. Это руководство описывает поведение препроцессора **ISO**.Чтобы свести к минимуму необоснованные различия, когда поведение препроцессора ISO не противоречит традиционной семантике, традиционный препроцессор должен вести себя точно так же. Различные существующие различия подробно описаны в разделе Глава 10 [Традиционный режим], стр. 51.Для ясности, если не указано иное, ссылки на **‘CPP’**в данном руководстве относятся к **GNU CPP**.

* 1. Character sets Наборы символов

Обработка набора символов **исходного кода в C** и родственных языках довольно сложна. Стандарт **C** обсуждает два набора символов, но на самом деле их как минимум четыре.Входные файлы для **CPP** могут иметь любой набор символов. Самое первое действие **CPP**, еще до того, как он начнет искать границы строк, — преобразовать файл в набор символов, который он использует для внутренней обработки. Этот набор в стандарте **C** называется исходным набором символов. Он должен быть изоморфен стандарту **ISO 10646**, также известному как **Unicode**. В **CPP** используется кодировка **UTF-8** **Unicode**. Наборы символов входных файлов задаются с помощью параметра

‘-finput-charset=’

Вся работа по предварительной обработке (предмет остальной части этого руководства) выполняется в исходном наборе символов. Если вы запросите текстовый вывод у препроцессора с параметром **‘-E’**, он будет в **UTF-8**. После завершения предварительной обработки строковые и символьные константы снова преобразуются в набор символов выполнения. Этот набор символов находится под контролем пользователя; по умолчанию является **UTF-8**, соответствующим исходному набору символов. Широкие строковые и символьные константы имеют свой собственный набор символов, который специально не упоминается в стандарте. Опять же, это находится под контролем пользователя. По умолчанию используется **UTF-16** или **UTF-32**, в зависимости от того, что соответствует типу **wchar\_t** цели и порядку байтов целевой машины.1 Восьмеричные и шестнадцатеричные управляющие последовательности не подвергаются преобразованию; **‘\x12’** имеет значение **0x12** независимо от текущего выбранного набора символов выполнения. Все другие escape-последовательности заменяются символом в исходном наборе символов, который они представляют, а затем преобразуются в набор символов выполнения, как и неэкранированные символы. В идентификаторах символы вне диапазона **ASCII** могут быть указаны с помощью **'\u'** и **'\ U'** экранируется или используется непосредственно во входной кодировке. Если строгое соответствие **ISO C90** указано с помощью такой опции, как

**‘-std=c90’** или используется **‘-fno-extended-identifiers’**, то эти конструкции не разрешены в идентификаторах.

Широкие строковые и символьные константы имеют свой собственный набор символов, который специально не упоминается в стандарте.Опять же, это находится под контролем пользователя. По умолчанию используется **UTF-16** или **UTF-32**, в зависимости от того, что соответствует типу **wchar\_t** цели и порядку байтов целевой машины.1 Восьмеричные и шестнадцатеричные управляющие последовательности не подвергаются преобразованию; **‘\x12’** имеет значение **0x12** независимо от текущего выбранного набора символов выполнения. Все другие escape-последовательности заменяются символом в исходном наборе символов, который они представляют,а затем преобразуются в набор символов выполнения, как и неэкранированные символы. В идентификаторах символы вне диапазона **ASCII** могут быть указаны с помощью **'\u'** и **'\ U'** экранируется или используется непосредственно во входной кодировке.Если строгое соответствие **ISO C90** указано с помощью такой опции, как **‘-std=c90’** или используется

**‘-fno-extended-identifiers’**, то эти конструкции не разрешены в идентификаторах.

1.2 **Initial processing** Первоначальная обработка

Препроцессор выполняет ряд текстовых преобразований на своем входе.Это происходит перед любой другой обработкой.Концептуально они происходят в строгом порядке, и весь файл выполняется через каждое преобразование до начала следующего. **CPP** фактически выполняет их все сразу по соображениям производительности.Эти преобразования примерно соответствуют первому три **«фазы перевода»**

**“phases of translation”**, описанные в стандарте **C**.

1. Входной файл читается в память и разбивается на строки.

Различные системы используют разные соглашения для обозначения конца строки. **GCC** принимает управляющие последовательности **ASCII LF, CR, LF и CR** как маркеры конца строки. Это канонические последовательности, используемые **Unix, DOS** и **VMS**, а также классической **Mac OS** (до **OSX**) соответственно. Таким образом, вы можете безопасно скопировать исходный код, написанный в любой из этих систем, в другую и использовать его без преобразования. (**GCC** может потерять след текущий номер строки, если в файле не всегда используется одно соглашение, как это иногда бывает, когда он редактируется на компьютерах с разными соглашениями, которые совместно используют сеть файловая система.)

Если в последней строке любого входного файла нет маркера конца строки, конец файла считается неявно поставляющим один. В стандарте **C** сказано, что это состояние провоцирует поведение **undefined**, поэтому **GCC** выдаст предупреждающее сообщение.

1. Если триграфы включены, они заменяются соответствующими им одиночными символами. По по умолчанию **GCC** игнорирует триграфы, но если вы запросите режим строгого соответствия с параметр **‘-std’** или вы указываете параметр **‘-trigraphs’** , после чего он их преобразует. Это девять последовательностей из трех символов, каждая из которых начинается с **‘??’**, которые определяются **ISO C** для обозначения одиночных символов. Они допускают использование устаревших систем, в которых отсутствуют некоторые из Знак препинания **C** для использования **C**. Например, ‘??/’ означает ‘\’, поэтому ’??/n’— это символ константа для новой строки

UTF-16 не соответствует требованиям стандарта **C** для широкого набора символов, но выбор

**16-битный wchar\_t** закреплен в некоторых системных **ABI**, поэтому мы не можем это исправить. Триграфы не популярны, и многие компиляторы реализуют их некорректно.

Переносимый код не должен полагаться на то, что триграфы либо преобразуются, либо игнорируются. **GCC ‘-Wtrigraphs’** предупредит вас, когда триграф может изменить значение

вашей программы, если она была сконвертирована. См. [Wtrigraphs], стр. 66.В строковой константе вы можете предотвратить смешение последовательности вопросительных знаков с триграфом, вставив обратную косую черту между знаками вопроса, или раздели строковый литерал в триграфе и используя конкатенацию строковых литералов.

**"(??\?)"** — это строка **‘(???)’**, а не **‘(?]’**. Традиционные компиляторы языка **C** не распознают

эти идиомы.

Девять триграфов и их заменители

Trigraph: ??(  **??) ??< ??> ??= ??/ ??’ ??! ??-**

Replacement: **[ ] { } # \ ^ | ~**

3. Continued lines are merged into one long line.

**3. Продолжающиеся строки сливаются в одну длинную строку.**

Непрерывная строка — это строка, которая заканчивается обратной косой чертой «**\**». Обратная косая черта удаляется, и следующая строка объединяется с текущей. Пробел не вставляется, поэтому вы можете разбить строку в любом месте, даже в середине слова. (Обычно удобнее разбивать строки только по пробелам.)Обратная косая черта в конце непрерывной строки обычно называется обратной косой чертой новой строки. Если между обратной косой чертой и концом строки есть пробел, это все еще продолжение строки. Однако, поскольку это обычно результат ошибки редактирования, и многие компиляторы не примут ее как продолжение строки, **GCC** предупредит вас об этом**.**

4All comments are replaced with single spaces.

Все комментарии заменены одиночными пробелами.

Комментарии бывают двух видов. Блочные комментарии начинаются с ‘/\*’ и продолжаются до тех пор, пока следующий ‘\*/’. Блочные комментарии не вкладывают друг в друга: /\* это /\* один комментарий \*/ текст за пределами комментарияСтроковые комментарии начинаются с «//» и продолжаются до конца текущей строки. Строка

комментарии тоже не вложены друг в друга, но это не имеет значения, потому что они заканчивались бы на в любом случае то же место.// это // один комменттекст за пределами комментарияМожно с уверенностью помещать комментарии к строкам внутри комментариев к блоку или наоборот./\* блокировать комментарий// содержит строчный комментарий

еще один комментарий

\*/ за пределами комментария

// строчный комментарий /\* содержит блочный комментарий \*/

Но будьте осторожны, комментируя один конец блочного комментария строковым комментарием.

// л.к. /\* начинается комментарий к блоку

ой! это уже не комментарий \*/

Комментарии не распознаются внутри строковых литералов. "/\* бла \*/" — это строковая константа.‘/\* blah \*/’, а не пустая строка.Строковые комментарии отсутствуют в стандарте C 1989 г., но они распознаютсяGCC в качестве расширения. В C++ и в версии стандарта C 1999 года они являются официальной частью языка.

Так как эти преобразования происходят до всех остальных обработок, вы можете разделить строку

механически с обратной косой чертой-новой строкой в любом месте. Вы можете закомментировать конец строки.Вы можете продолжить комментарий к следующей строке с помощью обратной косой черты-новой строки. Вы можете даже разделить «**/\***», «**\*/**» и «**//**» на несколько строк с помощью обратной косой черты-новой строки. Например:

**/\**

**\***

**\*/ # /\***

**\*/ defi\**

**ne FO\**

**O 10\**

эквивалентен **#define FOO 1020**. Все эти приемы крайне запутывают и не должны использоваться в коде, который должен быть читабельным.Не существует способа предотвратить интерпретацию обратной косой черты в конце строки как обратная косая черта-переход на новую строку. Однако это не может повлиять на корректность программы.

* 1. Tokenization Токенизация

После того, как текстовые преобразования завершены, входной файл преобразуется в последовательность токенов предварительной обработки. В основном они соответствуют синтаксическим токенам, используемым языком **C**.

компилятор, но есть несколько отличий. Пробелы отделяют токены; он сам по себе не является маркером любого рода. Токены не обязательно должны быть разделены пробелом, но часто необходимо, чтобы избежать неясностей. Столкнувшись с последовательностью символов, которая имеет более одной возможной лексемы, препроцессор проявляет жадность. Он всегда делает каждую фишку, начиная слева, максимально возможной, прежде чем переходить к следующей фишке. Например, **a++++++b** интерпретируется как **++ ++ + b**, а не как **++ + ++ b**, даже несмотря на то, что последняя лексема может быть частью действительной программы на языке **C**, а первая — нет.

После того, как входной файл разбит на токены, границы токенов никогда не меняются, за исключением

когда используется оператор предварительной обработки **‘##’** для вставки токенов вместе. См. раздел 3.5.

[Связь], стр. 17. Например,

**#define foo() bar**

**foo()baz**

**7! bar baz**

**not**

**7! Barbaz**

Компилятор не повторно размечает выходные данные препроцессора. Каждый токен предобработки становится одним токеном компилятора. Токены предварительной обработки делятся на пять широких классов: **идентификаторы**, **числа предварительной обработки, строковые литералы, знаки пунктуации и другие**. Идентификатор такой же, как идентификатор в **C**: любая последовательность букв, цифр или символов подчеркивания, начинающаяся с буквы или символа подчеркивания. Ключевые слова **C** не имеют значения для препроцессора; это обычные идентификаторы. Например, вы можете определить макрос, имя которого является ключевым словом. Определен единственный идентификатор, который может считаться ключевым словом предварительной обработки. См. Раздел 4.2.3 [Определение], с. 43. Это в основном верно для других языков, использующих препроцессор **C**. Тем не менее, некоторые ключевые слова **C++** имеют значение даже в препроцессоре. См. раздел 3.7.4 [Именованные операторы C++], стр. 34. В стандарте C 1999 года идентификаторы могут содержать буквы, которые не являются частью «базового исходного набора символов», по усмотрению реализации (например, латинские буквы с акцентом,

греческие буквы или китайские идеограммы). Это можно сделать с помощью расширенного набора символов или

управляющие последовательности ‘\u’ и ‘\U’.

Как расширение, **GCC** рассматривает «**$**» как букву. Это сделано для совместимости с некоторыми системами, такими как **VMS**, где «**$**» обычно используется в именах функций и объектов, определяемых системой. **‘$’** не является буквой в строго соответствующем режиме или если вы укажете опцию **‘-$’**. См. Глава 12 [Вызов], стр. 58. Число препроцессора имеет довольно странное определение. В эту категорию входят все обычные целочисленные константы и константы с плавающей запятой, ожидаемые от **C**, а также ряд других констант.вещи, которые можно было бы изначально не распознать как число. Формально номера предварительной обработки начинаются с необязательной точки, обязательной десятичной цифры, а затем продолжаются любой последовательностью.букв, цифр, знаков подчеркивания, точек и степеней. Экспоненты представляют собой двухсимвольные последовательности **e+’, ‘e-’, ‘E+’, ‘E-’, ‘p+’, ‘p-’, ‘P+’, ‘P-’.** (Показатели, начинающиеся с **‘p’** или **‘P’** используются для шестнадцатеричных констант с плавающей точкой.)

Цель этого необычного определения состоит в том, чтобы изолировать препроцессор от всей сложности числовых констант. Ему не нужно различать лексически допустимые и недопустимые числа с плавающей запятой, что сложно. Определение также позволяет вам разделить идентификатор в любой позиции и получить ровно две лексемы, которые затем можно вставить обратно вместе с оператором **‘##’**. Предварительная обработка чисел может привести к тому, что программы будут неверно интерпретированы. Например, 0xE+12 — это номер предварительной обработки, который не преобразуется в какую-либо допустимую числовую константу, поэтому это синтаксическая ошибка. Это не означает 0xE + 12, что вы могли иметь в виду. Строковые литералы — это строковые константы, символьные константы и имена файлов заголовков (аргумент **‘#include’**).2 Строковые константы и символьные константы просты:

**"..."** или же '...'. В любом случае встроенные кавычки должны быть экранированы обратной косой чертой: **'\''** — это константа символа для **'''**. Длина символьной константы не ограничена, но значение символьной константы, содержащей более одного символа, определяется реализацией. См. главу 11 [Подробности реализации], стр. 54.

Имена заголовочных файлов либо выглядят как строковые константы **"..."**, либо вместо этого пишутся в угловых скобках **<...>**. В любом случае, обратная косая черта — это обычный символ. Невозможно избежать закрывающей кавычки или угловой скобки. Препроцессор ищет заголовочный файл в разных местах в зависимости от того, какую форму вы используете. См. Раздел 2.2 [Операция включения], стр. 8. Ни один строковый литерал не может выходить за конец строки. Вместо этого вы можете использовать непрерывные строки или конкатенацию строковых констант.

Знаки препинания — это все обычные знаки пунктуации, которые имеют значение для C и C++. Все знаки пунктуации в ASCII, кроме трех, являются пунктуаторами C. Исключения составляют **‘@’,‘$’** и **‘‘’**. Кроме того, все двухсимвольные и трехсимвольные операторы являются знаками препинания. Есть также шесть орграфов, которые стандарт C++ называет альтернативными токенами, которые просто альтернативные способы написания других знаков препинания. Это вторая попытка исправить отсутствие пунктуации в устаревших системах. Он не имеет негативных побочных эффектов, в отличие от триграфов, но не так широко распространяется. Орграфы и соответствующие им нормальные знаки препинания В стандарте C термин «строковый литерал» используется только для обозначения того, что мы называем строковыми константами.

Digraph: Орграф: <% %> <: :> %: %:%:

Punctuator: Знак препинания: { } [ ] # ##

Любой другой одиночный байт считается «другим» “other” и без изменений передается на выход препроцессора. Компилятор C почти наверняка отвергнет исходный код, содержащий «другие» **“other”** токены. В **ASCII** единственными «другими» символами являются **‘@’, ‘$’, ‘‘’** и управляющие символы, кроме **NUL** (все биты равны нулю). (Обратите внимание, что **‘$’** обычно считается буквой.) Все байты с установленным старшим битом (числовой диапазон **0x7F–0xFF**), которые не были успешно интерпретированы как часть расширенного символа во входной кодировке, также являются «прочими» в настоящем коде. реализация. **NUL** — это особый случай из-за высокой вероятности того, что его появление случайно, а также из-за того, что он может быть невидим для пользователя (многие терминалы вообще не отображают **NUL**). В комментариях **NUL** молча игнорируются, как и любой другой символ. быть. В бегущем тексте **NUL** считается пустым пространством. Например, у этих двух директив есть то же самое значение.

**#define X^@1**

**#define X 1**

**(**где **‘^@’ — ASCII NUL).** Внутри строковых или символьных констант значения **NUL** сохраняются. В двух последних случаях препроцессор выдает предупреждение**.**

* 1. The preprocessing language Язык предварительной обработки

После токенизации поток токенов можно просто передать прямо на компилятор.парсер. Однако, если он содержит какие-либо операции на языке предварительной обработки, он будет преобразился первым. Этот этап примерно соответствует стандартной **«фазе перевода 4» “translation phase 4”** и именно это большинство людей считают работой препроцессора. Язык препроцессора состоит из директив, которые нужно выполнить, и макросов, которые нужно расширить.

Основные его возможности:

* Включение заголовочных файлов. Это файлы объявлений, которые можно подставить в вашу программу.
* Макрос расширения. Вы можете определить макросы, которые являются сокращениями для произвольных фрагментов кода C. Препроцессор заменит макросы их определениями на протяжении всей программы. Некоторые макросы определяются автоматически.
* Условная компиляция. Вы можете включать или исключать части программы в соответствии с различными условиями.
* Диагностика. Вы можете обнаружить проблемы во время компиляции и выдать ошибки или предупреждения.

Есть еще несколько менее полезных функций. За исключением расширения предопределенных макросов, все эти операции запускаются директивами предварительной обработки. Директивы предварительной обработки — это строки в вашей программе, которые начинаются с **‘#’**. Пробел разрешен до и после **‘#’**. За символом **#** следует идентификатор, имя директивы. Он определяет операцию, которую нужно выполнить. Директивы обычно называют как **‘#name’**, где name — имя директивы. Например, **‘#define’**— это директива, определяющая макрос.

**Chapter 2: Header Files Глава 2: Заголовочные файлы**

**«#»**, с которого начинается директива, не может исходить из расширения макроса. Кроме того, имя директивы не раскрывается с помощью макроса. Таким образом, если **foo** определен как макрос, расширяющийся до определения, то не делает **‘#foo’** допустимой директивой препроцессора.Набор допустимых имен директив является фиксированным. Программы не могут определять новые директивы препроцессинга.( предварительной обработки).

Для некоторых директив требуются аргументы; они составляют остальную часть строки директивы и должны отделяться от имени директивы пробелом. Например, после **‘#define’** должно следовать имя макроса и его предполагаемое расширение.Директива предварительной обработки не может охватывать более одной строки. Однако строка может быть продолжена обратной косой чертой-новой строкой или блочным комментарием, который выходит за конец строки. В любом случае, когда директива обрабатывается, продолжения уже объединены с первой строкой, чтобы получилась одна длинная строка.

* 1. Include Syntax

Как пользовательские, так и системные заголовочные файлы включаются с помощью директивы предварительной обработки «#include». У него есть два варианта:

**#include <file>**

Этот вариант используется для файлов заголовков системы. Он ищет файл с именем file в стандартном списке системных директорий. Вы можете добавить каталоги к этому списку с опцией ‘-I’ (см. Глава 12 [Вызов], стр. 58).

**#include "file"**

Этот вариант используется для файлов заголовков вашей программы. Он ищет файл с именем file сначала в каталоге, содержащем текущий файл, затем в каталогах с кавычками, а затем в тех же каталогах, что и <file>. Вы можете добавлять каталоги к списку каталогов цитат с помощью опции **‘-iquote’** .

Аргумент **#include**, заключенный в кавычки или угловые скобки, ведет себя как строковая константа, поскольку комментарии не распознаются, а имена макросов не расширяются. Таким образом, **#include <x/\*y>** указывает включение файла системного заголовка с именем **‘x/\*y’**. Однако, если в файле встречаются backslashes, они считаются обычными текстовыми символами, а не управляющими символами. Ни одна из escape-последовательностей символов не подходит для строковых констант в **C** обрабатываются. Таким образом, **#include "x\n\\y"** задает имя файла, содержащее три обратных слэша. (Некоторые системы интерпретируют

символ **‘\’** как разделитель пути. Все они также интерпретируют **'/'** так же. Наиболее удобно использовать только **'/'**. Если в строке после имени файла есть что-либо (кроме комментариев), это является ошибкой.

* 1. Include Operation Операция включения

Директива #include работает, предписывая препроцессору C сканировать указанный файл в качестве входных данных, прежде чем продолжить работу с остальной частью текущего файла. Вывод из препроцессора содержит уже сгенерированный вывод, за которым следует вывод, полученный из включенного файла,за которым следует вывод, полученный из текста после

директивы **«#include».**За например,если у вас есть файл заголовка **‘header.h’** следующим образом,

**char \*test (void);**

и основная**main** программа называется **‘program.c’** that uses the header file, like this,

**int x;**

**#include "header.h"**

**int**

**main (void)**

**{**

**puts (test ());**

**}**

компилятор увидит тот же поток токенов, что и при **if**

**int x;**

**char \*test (void);**

**int**

**main (void)**

**{**

**puts (test ())**

**}**

Включенные файлы не ограничены объявлениями и определениями макросов; это просто типичное использование**(typical uses).** Любой фрагмент программы на C может быть включен из другого файла. Включает файл может даже содержать начало инструкции, которая завершается в содержащем файле, или конец инструкции, которая была запущена во включающем файле.Однако включенный файл должны состоять из целых токенов. Комментарии и строковые литералы,которые не были закрыты до конца включаемого файла,недопустимы.Для восстановления после ошибки считается,что они заканчиваются на конец файла**.**

Чтобы не запутаться,лучше всего, если заголовочные файлы содержат только полные синтаксические единицы — объявления или определения функций, объявления типов и т. д.

Строка, следующая за директивой **‘#include’**, всегда обрабатывается препроцессором **C** как отдельная строка, даже если во включаемом файле отсутствует последняя строка новой строки.

* 1. Search Path Поисковый путь

По умолчанию препроцессор ищет файлы заголовков, включенные в кавычки директивы.

#include "файл" сначала относительно каталога текущего файла, а затем в предварительно сконфигурированном списке стандартных системных каталогов. Например, если «/usr/include/sys/stat.h»

содержит #include "types.h", GCC сначала ищет "types.h" в "/usr/include/sys", а затем

на своем обычном пути поиска.