Глава 1: Обзор 1

1 Обзор

Препроцессор **C**, часто известный как **cpp**, представляет собой макропроцессор, который автоматически используется компилятором **C** для преобразования вашей программы 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 исходным кодом на языках C, C++ и Objective-C. В прошлом его использовали как обычный текстовый процессор. Он задохнется на входе, который не подчиняется лексическим правилам C. Он задохнется на входе, который не подчиняется лексическим правилам C.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 имеют значения для языков семейства **C**. Если **Makefile** предварительно обрабатывается, все жесткие вкладки будут удалены, и **Makefile** не будет работать.

Сказав это, вы часто можете обойтись без использования **cpp** для вещей, которые не являются **C**. Другие языки программирования на Algol часто безопасны (Ада и т. д.) Так и сборка,с осторожностью. Режим **‘-traditional-cpp’** сохраняет больше пробелов,и в остальном более разрешительный.Многих проблем можно избежать, написав комментарии в стиле C или C++ вместо собственных комментариев. 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 помогает, попробуйте обычный текстовый процессор, например **GNU M4**. Препроцессоры **C** различаются в некоторых деталях. В этом руководстве обсуждается препроцессор **GNU C,**который предоставляет небольшой расширенный набор функций стандарта **ISO** **C.** В режиме по умолчанию препроцессор **GNU C** не делает некоторых вещей, требуемых 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 использоватьОпции **‘-std=c90’,** ‘-**std=c99’, ‘-std=c11’** или **‘-std=c17’** в зависимости от версии стандарт, который вы хотите. Чтобы получить всю обязательную диагностику, вы также должны использовать **‘-pedantic’** См. Глава 12 [Вызов], стр. 58.

В этом руководстве описывается 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 различия, которые существуют, подробно описаны в разделе Глава 10 [Традиционный режим], стр. 51. Для ясности, если не указано иное, ссылки на **‘CPP’** в этом руководстве относятся к **GNU CPP**. 1.1 Наборы символовОбработка 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.Файлы, вводимые в **CPP**, могут иметь любой набор символов. Самое первое действие **CPP**, еще до того, как он начнет искать границы строк, — преобразовать файл в используемый им набор символов.для внутренней обработки. Этот набор является тем, что стандарт **C** называет исходным набором символов. Он должен быть изоморфен стандарту ISO 10646, также известному как Unicode. **CPP** использует кодировку UTF-8.Unicode. Наборы символов входных файлов задаются с помощью опции **‘-finput-charset=’**.

1.1 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.Файлы, вводимые в CPP, могут иметь любой набор символов. Самое первое действие **CPP**, еще до того, как он начнет искать границы строк, — преобразовать файл в используемый им набор символов.для внутренней обработки. Этот набор является тем, что стандарт **C** называет исходным набором символов. Он должен быть изоморфен стандарту **ISO 10646**, также известному как **Unicode**. **CPP** использует кодировку **UTF-8**.Unicode. Наборы символов входных файлов задаются с помощью опции

**‘-finput-charset=’.**

Chapter 1: Overview

Вся работа по предварительной обработке (предмет остальной части этого руководства)выполняется в исходном наборе символов. Если вы запрашиваете текстовый вывод у препроцессора с опцией **‘-E’**, он будет в **UTF-8**. После завершения предварительной обработки строковые и символьные константы снова преобразовываются в набор символов выполнения.Этот набор символов находится под контролем пользователя; по умолчанию **UTF-8**,соответствующим исходному набору символов. Широкие строковые и символьные константы имеют свой собственный набор символов, который не упоминается в стандарте. Опять же, это под контролем пользователя. По умолчанию используется **UTF-16** или **UTF-32**, в зависимости от того, что соответствует целевому **wchar\_t тип,**в порядке следования байтов целевой машины Восьмеричные и шестнадцатеричные управляющие последовательности не подвергаться конверсии; **’\x12’** имеет значение **0x12** независимо от выбранного в данный момент набор символов выполнения. Все остальные escape-символы заменяются символом в исходнике.Набор символов, который они представляют,затем преобразуется в набор символов выполнения, как и неэкранированные символы.В идентификаторах символы вне диапазона **ASCII** могут быть указаны с помощью**‘\u’** и **‘\U’** **Экранирует(escapes)** или используется непосредственно во входной кодировке. Если указано строгое соответствие стандарту **ISO C90** с параметром, таким как **«-std=c90»,**или используется **‘-fno-extended-identifiers’**,тогда эти конструкции не допускаются в идентификаторах.

**1.2 Начальная обработка 1.2 Initial processing**

Препроцессор выполняет ряд текстовых преобразований на своем входе.Это происходит перед любой другой обработкой. Концептуально они происходят в строгом порядке, и весь файл проходит через каждое преобразование, прежде чем начнется следующее. **CPP** фактически выполняет их все сразу по соображениям производительности. Эти преобразования примерно соответствуют первым трем **«фазам перевода» “phases of translation”**, описанным в стандарте **C**.

1. *Входной файл считывается в память и разбивается на строки*.В разных системах используются разные соглашения для обозначения конца строки. **GCC** принимает управляющие последовательности **ASCII LF, CR LF** и **CR** в качестве маркеров 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 номер текущей строки, если в файле не всегда используется одно соглашение, как это иногда случается, когда он редактируется на компьютерах с разными соглашениями, которые совместно используют сетевую файловую систему.) Если последняя строка любого входного файла отсутствует маркер конца строки, считается, что конец файла неявно предоставляет один. Стандарт **C** говорит, что это условие вызывает неопределенное поведение, поэтому **GCC** выдаст предупреждающее сообщение.

2. Если разрешены триграфы, они заменяются соответствующими одиночными символами. По умолчанию **GCC** игнорирует триграфы, но если вы запросите строго соответствующий режим с параметром **'-std'** или укажете параметр **'-trigraphs'**, он преобразует их. Это девять последовательностей из трех символов, все начинаются с **‘??’**, которые определены **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’**— символьная константа для новой строки.**UTF-16** не удовлетворяет требованиям стандарта **C** для широкого набора символов, но выбор **16-bit** параметр **wchar\_t** закреплен в некоторых системных **ABI**, поэтому мы не можем это исправить.Триграфы не популярны, и многие компиляторы реализуют их неправильно. Переносимый код не должен полагаться на преобразование или игнорирование триграфов. С помощью **‘-Wtrigraphs’ GCC** предупредит вас, когда триграф может изменить значение вашего программа, если она была сконвертирована. См. *[Wtrigraphs], стр. 66.* *В строковой константе вы можете предотвратить путаницу в последовательности вопросительных знаков с триграфом, вставив обратную косую черту между вопросительными знаками или разделивстроковый литерал в триграфе и использование конкатенации строковых литералов.* **"(??\?)"** *— это строка* ***‘(???)’****, а не ‘***(?]’***. Традиционные компиляторы* **C** *не распознают эти идиомы.Девять триграфов и их замены*

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

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

3. Непрерывные строки сливаются в одну длинную строку.Непрерывная строка — это строка, заканчивающаяся обратной backslash **‘\’**.Backslash удаляется, а следующая строка присоединяется к текущей. Пробел не вставлен, поэтому вы можете разделить строку в любом месте, даже в середине слова. (Обычно удобнее разбивать строки только по пробелам.) **Backslash** в конце непрерывной строки обычно называется **Backslash** новой строки. Если между **Backslash** и концом строки есть пробел, это все еще продолжение строки. Однако, поскольку это обычно является результатом ошибки редактирования, и многие компиляторы не примут ее как продолжение строки, **GCC** предупредит вас об этом.

4. Все комментарии заменяются одинарными пробелами.Есть два вида комментариев. Блочные комментарии начинаются с ‘/\*’ и продолжаются до тех пор, пока следующий '\*/'. Блочные комментарии не вложены:

/\* this is /\* one comment \*/ text outside comment

Комментарии к строке начинаются с «//» и продолжаются до конца текущей строки. Линия комментарии тоже не вложены, но это не имеет значения, потому что они заканчивались бы на все равно то же место.

// this is // one comment

text outside comment

Можно безопасно помещать комментарии к строкам внутри комментариев к блоку или наоборот.

/\* block comment

// contains line comment

yet more comment

\*/ outside comment

// line comment /\* contains block comment \*/

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

// l.c. /\* block comment begins

oops! this isn’t a comment anymore \*/

Комментарии не распознаются внутри строковых литералов. **"/\* blah \*/"** — это строковая константа **‘/\* blah \*/’**, а не пустая строка Строковые комментарии отсутствуют в стандарте C 1989 года, но они распознаются**.GCC** в качестве расширения. В **C++** и в редакции стандарта C 1999 г. они являются официальная часть языка. Поскольку эти преобразования происходят до всех других обработок, вы можете механически разделить строку с помощью обратной косой черты-новой строки в любом месте. Конец строки можно закомментировать.Вы можете продолжить комментарий к следующей строке с помощью обратной косой черты-новой строки. Вы даже можете разделить **‘/\*’**, **‘\*/’** и **‘//’** на несколько строк с помощью обратной косой черты-перевода строки. Например:

/\

\*

\*/ # /\*

\*/ defi\

ne FO\

O 10\

20

эквивалентен #define FOO 1020. Все эти уловки очень запутывают и не должны использоваться в коде, предназначенном для чтения. Невозможно предотвратить интерпретацию обратной косой черты в конце строки как обратной косой черты-новой строки. Однако это не может повлиять на правильную программу.

**1.3 Токенизация**

После завершения текстовых преобразований входной файл преобразуется в *последовательность* **токенов** *предварительной обработки*. В основном они соответствуют *синтаксическим токенам*, используемым компилятором **C**, но есть несколько отличий. отделяют токены; он сам по себе не является маркером любого рода. Токены не обязательно должны быть разделены пробелом, но часто необходимо избегать двусмысленностей. При столкновении с последовательностью символов, которая имеет более чем одну возможную токенизацию, препроцессор активен.Он всегда делает каждый токен, начиная слева,как можно больше, прежде чем перейти к следующему токену.

Например, **a+++++b** интерпретируется какa **++ ++ + b**, а не как **++ + ++ b**, даже несмотря на то, что последняя токенизация может быть частью программы на **valid C**, а первая — нет. Как только входной файл разбит на токены, границы токенов никогда не меняются, за исключениемкогда используется оператор предварительной обработки **‘##’** для вставки токенов вместе. См. раздел **3.5. [Concatenation][Связь],** стр. 17. Например,

**#define foo() bar**

**foo()baz**

**|->bar baz**

**not**

**|->barbaz**

Компилятор не выполняет повторную токенизацию вывода препроцессора. Каждый токен препроцессора становится одним токеном компилятора.Токены предварительной обработки делятся на пять широких классов:идентификаторы, номера предварительной обработки,строковые литералы, знаки препинания и прочее. Идентификатор такой же, как идентификатор в **C**:любая последовательность букв, цифр или знаков подчеркивания, начинающаяся с буквы или знака подчеркивания.Ключевые слова C не имеют значения для препроцессора; это обычные идентификаторы.Например, вы можете определить макрос, имя которого является ключевым словом.Определяется единственный идентификатор, который можно считать ключевым словом предварительной обработки. См. раздел 4.2.3 **[Defined]** **[Определено**], стр. 43.В основном это относится к другим языкам, использующим препроцессор C.Тем не менее, некоторые ключевые слова **C++** имеют значение даже в препроцессоре. См. Раздел 3.7.4**[C++ Named Operators],** [**ИменованныеОператоры**], стр. 34.

Chapter 1: Overview 5