Приложение А. Справочное руководство

A1. Введение

Данное руководство описывает язык программирования Си, определенный 31 октября 1989 г. в соответствии с проектом, утвержденным в ANSI в качестве Американского национального стандарта для информационных систем: Язык программирования Си, X3.159-1989 (**"American National Standard for Information Systems - Programming Language C, X3.159-1989"**). Это описание - лишь один из вариантов предлагаемого стандарта, а не сам стандарт, однако мы специально заботились о том, чтобы сделать его надежным руководством по языку.

Настоящий документ в основном следует общей схеме описания, принятой в стандарте (публикация которого в свою очередь основывалась на первом издании этой книги), однако в организационном плане есть различия. Если не считать отклонений в названиях нескольких продуктов и отсутствия формальных определений лексем и препроцессора, грамматика языка здесь и грамматика в стандарте эквивалентны.

> Далее примечания (как и это) набираются с отступом от левого края страницы. В основном эти примечания касаются отличий стандарта от версии языка, описанной в первом издании этой книги, и от последующих нововведений в различных компиляторах.

A2. Соглашения о лексике

Программа состоит из одной или нескольких *единиц трансляции*, хранящихся в виде файлов. Каждая такая единица проходит несколько фаз трансляции, описанных в A12. Начальные фазы осуществляют лексические преобразования нижнего уровня, выполняют директивы, заданные в программе строками, начинающимися со знака #, обрабатывают макроопределения и производят макрорасширения. По завершении работы препроцессора (A12) программа представляется к виде последовательности лексем.

A2.1. Лексемы (tokens)

Существуют шесть классов лексем (или токенов): идентификаторы, ключевые слона, константы, строковые литералы, операторы и прочие разделители. Пробелы, горизонтальные и вертикальные табуляции, новые строки, переводы страницы и комментарии (имеющие общее название символы-разделители) рассматриваются компилятором только как разделители лексем и в остальном на результат трансляции влияние не оказывают. Любой из символов-разделителей годится, чтобы отделить друг от друга соседние идентификаторы, ключевые слова и константы.

Если входной поток уже до некоторого символа разбит на лексемы, то следующей лексемой будет самая длинная строка, которая может быть лексемой.

A2.2. Комментарий

Символы /* открывают комментарий, а символы */ закрывают его. Комментарии нельзя вкладывать друг в друга, их нельзя помещать внутрь строк или текстовых литералов.

A2.3. Идентификаторы

Идентификатор — это последовательность букв и цифр. Первым символом должна быть буква; знак подчеркивания _ считается буквой. Буквы нижнего и верхнего регистров различаются. Идентификаторы могут иметь любую длину; для внутренних идентификаторов значимыми являются первые 31 символ; в некоторых реализациях принято большее число значимых символов. К внутренним идентификаторам относятся имена макросов и все другие имена, не имеющие внешних связей (A11.2). На идентификаторы с внешними связями могут накладываться большие ограничения: иногда воспринимаются не более шести первых символов и могут не различаться буквы верхнего и нижнего регистров.

A2.4. Ключевые слова

Следующие идентификаторы зарезервированы в качестве ключевых слов и в другом смысле использоваться не могут:

```
auto
break
char
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

В некоторых реализациях резервируются также слова **fortran** и **asm**.

> Ключевые слова **const**, **signed** и **volatile** впервые появились в стандарте ANSI; **enum** и **void** - новые по отношению к первому изданию книги, но уже использовались; ранее зарезервированное **entry** нигде не использовалось и поэтому более не резервируется.

A2.5. Константы

Существует несколько видов констант. Каждая имеет свой тип данных; базовые типы рассматриваются в A4.2.

```
константа:
    целая–константа
    символьная-константа
    константа-с-плавающей-точкой
    константа-перечисление
```

A2.5.1. Целые константы

Целая константа, состоящая из последовательности цифр, воспринимается как восьмеричная, если она начинается с 0 (цифры нуль), и как десятичная в противном случае. Восьмеричная константа не

содержит цифр 8 и 9. Последовательность цифр, перед которой стоят 0x или 0X, рассматривается как шестнадцатеричное целое. В шестнадцатеричные цифры включены буквы от a (или A) до f (или F) со значениями от 10 до 15.

Целая константа может быть записана с буквой-суффиксом *u* (или *U*) для спецификации ее как беззнаковой константы. Она также может быть с буквой- суффиксом *l* (или *L*) для указания, что она имеет тип *long*.

Тип целой константы зависит от ее вида, значения и суффикса (о типах см. A4). Если константа - десятичная и не имеет суффикса, то она принимает первый из следующих типов, который годится для представления ее значения: *int*, *long int*, *unsigned long int*. Восьмеричная или шестнадцатеричная константа без суффикса принимает первый возможный из типов: *int*, *unsigned int*, *long int*, *unsigned long int*. Если константа имеет суффикс *u* или *U*, то она принимает первый возможный из типов: *unsigned int*, *unsigned long int*. Если константа имеет суффикс *l* или *L*, то она принимает первый возможный из типов: *long int*, *unsigned long int*. Если константа имеет суффикс *ul* или *UL*, то она принимает тип *unsigned long int*.

> Типы целых констант получили существенное развитие в сравнении с первой редакцией языка, в которой большие целые имели просто тип *long*. Суффиксы *U* и *u* введены впервые.

A2.5.2. Символьные константы

Символьная константа - это последовательность из одной или нескольких символов, заключенная в одиночные кавычки (например *'x'*). Если внутри одиночных кавычек расположен один символ, значением константы является числовое значение этого символа в кодировке, принятой на данной машине. Значение константы с несколькими символами зависит от реализации.

Символьная константа не может содержать в себе одиночную кавычку ' или символ новой строки; чтобы изобразить их и некоторые другие символы, могут быть использованы *эскейп-последовательности*:

новая строка (newline, linefeed)	**NL (LF)**	**\n**
горизонтальная табуляция (horizontal tab)	**HT**	**\t**
вертикальная табуляция (vertical tab)	**VT**	**\v**
возврат на шаг (backspace)	**BS**	**\b**
возврат каретки (carriage return)	**CR**	**\r**
перевод страницы (formfeed)	**FF**	**\f**
сигнал звонок (audible alert, bell)	**BEL**	**\a**
обратная наклонная черта (backslash)	\	\\
знак вопроса (question mark)	**?**	**\?**
одиночная кавычка (single quote)	**'**	**\'**
двойная кавычка (double quote)	**"**	**\"**
восьмеричный код (octal number)	**ooo**	**\ooo**
шестнадцатеричный код (hex number)	**hh**	**\xhh**

Эскейп-последовательность *\ooo* состоит из обратной наклонной черты, за которой следуют одна, две или три восьмеричные цифры, специфицирующие значение желаемого символа. Наиболее частым примером такой конструкции является \0 (за которой не следует цифра); она специфицирует NULL-символ. Эскейп-последовательность *\xhh* состоит из обратной наклонной черты с буквой *x*, за которыми следуют шестнадцатеричные цифры, специфицирующие значение желаемого символа. На количество

цифр нет ограничений, но результат будет не определен, если значение полученного символа превысит значение самого "большого" из допустимых символов. Если в данной реализации тип *char* трактуется как число со знаком, то значение и в восьмеричной, и в шестнадцатеричной эскейп-последовательности получается с помощью "распространения знака", как если бы выполнялась операция приведения к типу *char*. Если за \ не следует ни один из перечисленных выше символов, результат не определен.

В некоторых реализациях имеется расширенный набор символов, который не может быть охвачен типом *char*. Константа для такого набора пишется с буквой *L* впереди (например L'x') и называется расширенной символьной константой. Такая константа имеет тип *wchar_t* (целочисленный тип, определенный в стандартном заголовочном файле <**stddef.h**>). Как и в случае обычных символьных констант, здесь также возможны восьмеричные и шестнадцатеричные эскейп- последовательности; если специфицированное значение превысит тип *wchar_t*, результат будет не определен.

> Некоторые из приведенных эскейп-последовательностей новые (шестнадцатеричные в частности). Новым является и расширенный тип для символов. Наборам символов, обычно используемым в Америке и Западной Европе, подходит тип *char*, а тип *wchar_t* был добавлен главным образом для азиатских языков.

A2.5.3. Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, дробной части, **e** или **E** и целого (возможно, со знаком), представляющего порядок, и, возможно, суффикса типа, задаваемого одной из букв: **f**, **F**, **l** или **L**. И целая, и дробная часть представляют собой последовательность цифр. Либо целая часть, либо дробная часть (но не обе вместе) могут отсутствовать; также могут отсутствовать десятичная точка или *E* с порядком (но не обе одновременно). Тип определяется суффиксом: *F* или *f* определяют тип *float*, *L* или *l* - тип *long double*; при отсутствии суффикса подразумевается тип *double*.

> Суффиксы для констант с плавающей точкой являются нововведением.

A2.5.4. Константы-перечисления

Идентификаторы, объявленные как элементы перечисления (A8.4), являются константами типа *int*.

A2.6. Строковые литералы

Строковый литерал, который также называют строковой константой, - это последовательность символов, заключенная в двойные кавычки (Например, "..."). Строка имеет тип "массив символов" и память класса **static** (A4), которая инициализируется заданными символами. Представляются ли одинаковые строковые литералы одной копией или несколькими, зависит от реализации. Поведение программы, пытающейся изменить строковый литерал, не определено.

Написанные рядам строковые литералы объединяются (конкатенируются) в одну строку. После любой конкатенации к строке добавляется NULL-байт (\0), что позволяет программе, просматривающей строку, найти ее конец. Строковые литералы не могут содержать в себе символ новой строки или двойную кавычку; в них нужно использовать те же эскейп-последовательности, что и в символьных константах.

Как и в случае с символьными константами, строковый литерал с символами из расширенного набора должен начинаться с буквы **L** (например L"..."). Строковый литерал из расширенного набора имеет тип "массив из *wchar_t*". Конкатенация друг с другом обычных и "расширенных" строковых литералов не определена.

> То, что строковые литералы не обязательно представляются разными копиями, запрет на их модификацию, а также конкатенация соседних строковых литералов - нововведения ANSI-стандарта. "Расширенные" строковые литералы также объявлены впервые.

A3. Нотация синтаксиса

В нотации синтаксиса, используемой в этом руководстве, синтаксические понятия набираются курсивом, а слова и символы, воспринимаемые буквально, обычным шрифтом. Альтернативные конструкции обычно перечисляются в столбик (каждая альтернатива на отдельной строке); в редких случаях длинные списки небольших по размеру альтернатив располагаются в одной строке, помеченной словами "один из". Необязательное слово-термин или не термин снабжается индексом "*необ.*". Так, запись

{ *выражение*$_{необ}$ }

обозначает выражение, заключенное в фигурные скобки, которое в общем случае может отсутствовать. Полный перечень синтаксических конструкций приведен в A13.

> В отличие от грамматики, данной в первом издании этой книги, приведенная здесь грамматика старшинство и порядок выполнения операций в выражениях описывает явно.

A4. Что обозначают идентификаторы

Идентификаторы, или имена, ссылаются на разные объекты (в оригинале - *things*. - Примеч. ред.): функции; теги структур, объединений и перечислений; элементы структур или объединений; *typedef*-имена; метки и объекты. Объектом (называемым иногда переменной) является часть памяти, интерпретация которой зависит от двух главных характеристик: *класса памяти* н ее *типа*. Класс памяти сообщает о времени жизни памяти, связанной с идентифицируемым объектом, тип определяет, какого рода значения находятся в объекте. С любым именем ассоциируются своя область видимости (т. е. тот участок программы, где это имя известно) и атрибут связи, определяющий, обозначает ли это имя в другом файле тот же самый объект или функцию. Область видимости и атрибут связи обсуждаются в A11.

A4.1. Класс памяти

Существуют два класса памяти: *автоматический* и *статический*. Несколько ключевых слов в совокупности с контекстом объявлений объектов специфицируют класс памяти для этих объектов.

Автоматические объекты локальны в блоке (A9.3), при выходе из него они "исчезают". Объявление, заданное внутри блока, если в нем отсутствует спецификация класса памяти или указан спецификатор auto, создаст автоматический объект. Объект, помеченный в объявлении словом **register**, является автоматическим и размещается по возможности в регистре машины.

Статические объекты могут быть локальными в блоке или располагаться вне блоков, но в обоих случаях их значения сохраняются после выхода из блока (или функции) до повторного в него входа. Внутри блока (в том числе и в блоке, образующем тело функции) статические объекты в объявлениях помечаются словом **static**. Объекты, объявляемые вне всех блоков на одном уровне с определениями функций, всегда статические. С помощью ключевого слова **static** их можно сделать локальными в пределах транслируемой единицы (в этом случае они получают атрибут *внутренней связи*), и они становятся глобальными для всей программы, если опустить явное указание класса памяти или использовать ключевое слово **extern** (в этом случае они получают атрибут *внешней связи*).

A4.2. Базовые типы

Существует несколько базовых типов. Стандартный заголовочный файл <**limits.h**>, описанный в приложении B, определяет самое большое и самое малое значения для каждого типа в данной конкретной реализации. В приложении B приведены минимально возможные величины.

Размер объектов, объявляемых как символы, позволяет хранить любой символ из набора символов, принятого в машине. Если объект типа *char* действительно хранит символ из данного набора, то его значением является код этого символа, т. е. некоторое неотрицательное целое. Переменные типа *char* могут хранить и другие значения, но тогда диапазон их значений и особенно вопрос о том, знаковые эти значения или беззнаковые, зависит от реализации.

Беззнаковые символы, объявленные с помощью слов *unsigned char*, имеют ту же разрядность, что и обычные символы, но представляют неотрицательные значения; с помощью слов *signed char* можно явно объявить символы со знаком, которые занимают столько же места, как и обычные символы.

> Тип *unsigned char* не упоминался в первой редакции языка, но всеми использовался. Тип *signed char* - новый.

Помимо *char* среди целочисленных типов могут быть целые трех размеров: *short int*, *int* и *long int*. Обычные объекты типа *int* имеют естественный размер, принятый в архитектуре данной машины, другие размеры предназначены для специальных нужд. Более длинные целые по крайней мере покрывают все значения более коротких целых, однако в некоторых реализациях обычные целые могут быть эквивалентны коротким (*short*) или длинным (*long*) целым. Все типы *int* представляют значения со знаком, если не оговорено противное.

Для беззнаковых целых в объявлениях используется ключевое слово *unsigned*. Такие целые подчиняются арифметике по модулю 2 в степени *n*, где *n* - число битов в представлении числа, и, следовательно, в арифметике с беззнаковыми целыми никогда не бывает переполнения. Множество неотрицательных значений, которые могут храниться в объектах со знаком, является подмножеством значений, которые могут храниться в соответствующих объектах без знака; знаковое и беззнаковое представления каждого такого значения совпадают. Любые два из типов с плавающей точкой: с одинарной точностью (*float*), с двойной точностью (*double*) и с повышенной точностью (*long double*) могут быть синонимами, но каждый следующий тип этого списка должен по крайней мере обеспечивать точность предыдущего.

> *long double* - новый тип. В первой редакции языка синонимом для *double* был *long float*, теперь последний изъят из обращения.

Перечисления - единственные в своем роде типы, которым дается полный перечень значений; с каждым перечислением связывается множество именованных констант (A8.4). Перечисления ведут себя наподобие целых, но компилятор обычно выдает предупреждающее сообщение, если объекту некоторого перечислимого типа присваивается нечто, отличное от его константы, или выражение не из этого перечисления.

Поскольку объекты перечислений можно рассматривать как числа, перечисление относится к арифметическому типу. Типы *char* и *int* всех размеров, каждый из которых может быть со знаком или без знака, а также перечисления называют *целочисленными* (*integral*) типами. Типы *float*, *double* и *long double* называются типами с *плавающей точкой*.

Тип *void* специфицирует пустое множество значений. Он используется как "тип возвращаемого функцией значения" в том случае, когда она не генерирует никакого результирующего значения.

A4.3. Производные типы

Помимо базовых типов существует практически бесконечный класс производных типов, которые формируются из уже существующих и описывают следующие конструкции:

- *массивы* объектов заданного типа;
- *функции*, возвращающие объекты заданного типа;
- *указатели* на объекты заданного типа;
- *структуры*, содержащие последовательность объектов, возможно, различных заданных типов;
- *объединения*, каждое из которых может содержать любой из нескольких объектов различных заданных типов.

В общем случае приведенные методы конструирования объектов могут применяться рекурсивно.

A4.4. Квалификаторы типов

Тип объекта может снабжаться квалификатором. Объявление объекта с квалификатором **const** указывает на то, что его значение далее не будет изменяться; объявляя объект как **volatile**

(изменчивый, непостоянный (*англ.*)), мы указываем на его особые свойства для выполняемой компилятором оптимизации. Ни один из квалификаторов на диапазоны значений и арифметические свойства объектов не влияет. Квалификаторы обсуждаются в A8.2.

A5. Объекты и Lvalues

Объект - это некоторая именованная область памяти; **lvalue** - это выражение, обозначающее объект. Очевидным примером *lvalue* является идентификатор с соответствующим типом и классом памяти. Существуют операции, порождающие *lvalue*. Например, если *E* - выражение типа указатель, то **E* есть выражение для *lvalue*, обозначающего объект, на который указывает *E*. Термин "*lvalue*" произошел от записи присваивания E1 = E2, в которой левый (*left* - левый (*англ.*), отсюда буква *l*, *value* - значение) операнд E1 должен быть выражением *lvalue*. Описывая каждый оператор, мы сообщаем, ожидает ли он *lvalue* в качестве операндов и выдает ли *lvalue* в качестве результата.

A6. Преобразования

Некоторые операторы в зависимости от своих операндов могут вызывать преобразование их значений из одного типа в другой. В этом параграфе объясняется, что следует ожидать от таких преобразований. В A6.5 формулируются правила, по которым выполняются преобразования для большинства обычных операторов. При рассмотрении каждого отдельного оператора эти правила могут уточняться.

A6.1. Целочисленное повышение

Объект типа перечисление, символ, короткое целое, целое в битовом поле - все они со знаком или без могут использоваться в выражении там, где возможно применение целого. Если тип *int* позволяет "охватить" все значения исходного типа операнда, то операнд приводится к *int*, в противном случае он приводится к *unsigned int*. Эта процедура называется целочисленным повышением (*Integral promotion* – целочисленное повышение – иногда также переводят как "интегральное продвижение" – *Примеч. ред.*).

A6.2. Целочисленные преобразования

Любое целое приводится к некоторому заданному беззнаковому типу путем поиска конгруэнтного (т. е. имеющего то же двоичное представление) наименьшего неотрицательного значения и получения остатка от деления его на *nmax + 1*, где *nmax* - наибольшее число в этом беззнаковом типе. Для двоичного представления в дополнительном коде это означает либо выбрасывание лишних старших разрядов, если беззнаковый тип "уже" исходного типа, либо заполнение недостающих старших разрядов нулями (для значения без знака) или значением знака (для значения со знаком), если беззнаковый тип "шире" исходного.

В результате приведения любого целого к знаковому типу преобразуемое значение не меняется, если оно представимо в этом новом типе, и противном случае результат зависит от реализации.

A6.3. Целые и числа с плавающей точкой

При преобразовании из типа с плавающей точкой в целочисленный дробная часть значения отбрасывается; если полученное при этом значение нельзя представить в заданном целочисленном типе, то результат не определен. В частности, не определен результат преобразования отрицательных значений с плавающей точкой в беззнаковые целые.

Если значение преобразуется из целого в величину с плавающей точкой и она находится в допустимом диапазоне, но представляется в новом типе неточно, то результатом будет одно из двух значений нового типа, ближайших к исходному. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

A6.4. Типы с плавающей точкой

При преобразовании из типа с плавающей точкой меньшей точности в тип с плавающей точкой большей точности значение не изменяется. Если, наоборот, переход осуществляется от большей точности к меньшей и значение остается в допустимых пределах нового типа, то результатом будет одно из двух ближайших значений нового типа. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

A6.5. Арифметические преобразования

Во многих операциях преобразование типов операндов и определение типа результата осуществляются по одним и тем же правилам. Они состоят в том, что операнды приводятся к некоторому общему типу, который также является и типом результата. Эти правила называются *обычными арифметическими преобразованиями*.

- Если какой-либо из операндов имеет тип **long double**, то другой приводится к **long double**.
- В противном случае, если какой-либо из операндов имеет тип **double**, то другой приводится к **double**.
- В противном случае, если какой-либо из операндов имеет тип **float**, то другой приводится к **float**.
- В противном случае для обоих операндов осуществляется целочисленное повышение: затем, если один из операндов имеет тип **unsigned long int**, то и другой преобразуется в **unsigned long int**.
- В противном случае, если один из операндов принадлежит типу **long int**, а другой - **unsigned int**, то результат зависит от того, покрывает ли **long int** все значения **unsigned int**, и если это так, то **unsigned int** приводится к **long int**; если нет, то оба операнда преобразуются в **unsigned long int**.
- В противном случае, если один из операндов имеет тип **long int**, то другой приводится к **long int**.
- В противном случае, если один из операндов - **unsigned int**, то другой приводится к **unsigned int**.
- В противном случае оба операнда имеют тип **int**.

  Здесь есть два изменения. Во-первых, арифметика с операндами с плавающей точкой теперь может производиться с одинарной точностью, а не только с двойной; в первой редакции языка вся арифметика с плавающей точкой производилась с двойной точностью. Во-вторых, более короткий беззнаковый тип в комбинации с более длинным знаковым типом не распространяет свойство беззнаковости на тип результата; в первой редакции беззнаковый тип всегда доминировал. Новые правила немного сложнее, но до некоторой степени уменьшают вероятность появления неожиданных эффектов в комбинациях знаковых и беззнаковых величин. При сравнении беззнакового выражения со знаковым того же размера все же может возникнуть неожиданный результат.

A6.6. Указатели и целые

К указателю можно прибавлять (и вычитать из него) выражение целочисленного типа; последнее в этом случае подвергается преобразованию, описанному в A7.7 при рассмотрении оператора сложения.

К двум указателям на объекты одного типа, принадлежащие одному массиву, может применяться операция вычитания; результат приводится к целому посредством преобразования, описанного в A7.7 при рассмотрении оператора вычитания.

Целочисленное константное выражение со значением 0 или оно же, но приведенное к типу *void **, может быть преобразовано в указатель любого типа операторами приведения, присваивания и сравнения. Результатом будет NULL-указатель, который равен любому другому NULL-указателю того же типа, но не равен никакому указателю на реальный объект или функцию.

Для указателей допускаются и другие преобразования, но в связи с ними возникает проблема зависимости результата от реализации. Эти преобразования должны быть специфицированы явным оператором преобразования типа или оператором приведения (A7.5 и A8.8).

Указатель можно привести к целочисленному типу, достаточно большому для его хранения; требуемый размер зависит от реализации. Функция преобразования также зависит от реализации.

Объект целочисленного типа можно явно преобразовать в указатель. Если целое получено из указателя

и имеет достаточно большой размер, это преобразование даст тот же указатель; в противном случае результат зависит от реализации.

Указатель на один тип можно преобразовать в указатель на другой тип. Если исходный указатель ссылается на объект, должным образом не выровненный по границам слов памяти, то в результате может произойти ошибка адресации. Если требования на выравнивание у нового типа меньше или совпадают с требованиями на выравнивание первоначального типа, то гарантируется, что преобразование указателя в другой тип и обратно его не изменит; понятие "выравнивание" зависит от реализации, однако в любой реализации объекты типа *char* предъявляют минимальные требования на выравнивание. Как описано в A6.8, указатель может также преобразовываться в *void* * и обратно, значение указателя при этом не изменяется.

Указатель может быть преобразован в другой указатель того же типа с добавлением или удалением квалификаторов (A4.4, A8.2) того типа объекта, на который этот указатель показывает. Новый указатель, полученный добавлением квалификатора, имеет то же значение, но с дополнительными ограничениями, внесенными новыми квалификаторами. Операция по удалению квалификатора у объекта приводит к тому, что восстанавливается действие его начальных квалификаторов, заданных в объявлении этого объекта.

Наконец, указатель на функцию может быть преобразован в указатель на функцию другого типа. Вызов функции по преобразованному указателю зависит от реализации; однако, если указатель еще раз преобразовать к его исходному типу, результат будет идентичен вызову по первоначальному указателю.

A6.7. Тип void

Значение (несуществующее) объекта типа *void* никак нельзя использовать, его также нельзя явно или неявно привести к типу, отличному от *void*. Поскольку выражение типа *void* обозначает отсутствие значения, его можно применять только там, где не требуется значения. Например, в качестве выражения- инструкции (A9.2) или левого операнда у оператора "запятая" (A7.18). Выражение можно привести к тину *void* операцией приведения типа. Например, применительно к вызову функции, используемому в роли выражения-инструкции, операция приведения к *void* явным образом подчеркивает тот факт, что результат функции отбрасывается.

> Тип *void* не фигурировал в первом издании этой книги, однако за прошедшее время стал общеупотребительным.

A6.8. Указатели на void

Любой указатель на объект можно привести к типу *void* * без потери информации. Если результат подвергнуть обратному преобразованию, то мы получим прежний указатель. В отличие от преобразований указатель-в-указатель (рассмотренных в A6.6), которые требуют явных операторов приведения к типу, в присваиваниях и сравнениях указатель любого типа может выступать в паре с указателем типа *void* * без каких-либо предварительных преобразований типа.

> Такая интерпретация указателей *void* * - новая; ранее роль обобщенного указателя отводилась указателю типа *char* *. Стандарт ANSI официально разрешает использование указателей *void* * совместно с указателями других типов в присваиваниях и сравнениях; в иных комбинациях указателей стандарт требует явных преобразований типа.

A7. Выражения

Приоритеты описываемых операторов имеют тот же порядок, что и пункты данного параграфа (от высших к низшим). Например, для оператора +, описанного в A7.7, термин "операнды" означает "выражения, определенные в A7.1—A7.6". В каждом пункте описываются операторы, имеющие одинаковый приоритет, и указывается их ассоциативность (левая или правая). Приоритеты и ассоциативность всех операторов отражены в грамматике, приведенной в A13.

Приоритеты и ассоциативность полностью определены, а вот порядок вычисления выражения не определен за некоторым исключением даже для подвыражений с побочным эффектом. Это значит, что если в определении оператора последовательность вычисления его операндов специально не оговаривается, то в реализации можно свободно выбирать любой порядок вычислений и даже чередовать правый и левый порядок. Однако любой оператор использует значения своих операндов в точном соответствии с грамматическим разбором выражения, в котором он встречается.

> Это правило отменяет ранее предоставлявшуюся свободу в выборе порядка выполнения операций, которые математически коммутативны и ассоциативны, но которые в процессе вычислений могут таковыми не оказаться. Это изменение затрагивает только вычисления с плавающей точкой, выполняющиеся "на грани точности", и ситуации, когда возможно переполнение.

В языке не определен контроль за переполнением, делением на нуль и другими исключительными ситуациями, возникающими при вычислении выражения. В большинстве существующих реализаций Си при вычислении знаковых целочисленных выражений и присваивании переполнение игнорируется, но результат таких вычислений не определен. Трактовки деления на нуль и всех исключительных ситуаций, связанных с плавающей точкой, могут не совпадать в разных реализациях; иногда для обработки исключительных ситуаций предоставляется нестандартная библиотечная функция.

A7.1. Генерация указателя

Если тип выражения или подвыражения есть "массив из *T*", где *T* - некоторый тип, то значением этого выражения является указатель на первый элемент массива, и тип такого выражения заменяется на тип "указатель на *T*". Такая замена не делается, если выражение является операндом унарного оператора **&**, или операндом операций ++, **--**, **sizeof**, или левым операндом присваивания, или операндом оператора **.** (точка). Аналогично, выражение типа "функция, возвращающая *T*", кроме случая, когда оно является операндом для &, преобразуется в тип "указатель на функцию, возвращающую *T*".

A7.2. Первичные выражения

Первичные выражения это идентификаторы, константы, строки и выражения в скобках.

```
первичное - выражение:
    идентификатор
    константа
    строка
    (выражение)
```

Идентификатор, если он был должным образом объявлен (о том, как это делается, речь пойдет ниже), - первичное выражение. Тип идентификатора специфицируется в его объявлении. Идентификатор есть *lvalue*, если он обозначает объект (A5) арифметического типа, либо объект типа "структура", "объединение" или "указатель".

Константа - первичное выражение. Ее тип зависит от формы записи, которая была рассмотрена в A2.5.

Строковый литерал - первичное выражение. Изначально его тип - "массив из *char*" ("массив из *wchar_t*" для строки символов расширенного набора), но в соответствии с правилом, приведенным в A7.1, указанный тип обычно превращается в "указатель на *char*" ("указатель на *wchar_t*") с результирующим значением "указатель на первый символ строки". Для некоторых инициализаторов такая замена типа не делается. (см. A8.7)

Выражение в скобках - первичное выражение, тип и значение которого идентичны типу и значению этого же выражения без скобок. Наличие или отсутствие скобок не влияет на то, является ли данное выражение *lvalue* или нет.

A7.3. Постфиксные выражения

В постфиксных выражениях операторы выполняются слева направо.

```
постфиксное-выражение:
```

```
    первичное-выражение
    постфиксное-выражение [выражение]
    постфиксное-выражение (список-аргументов-выражений_необ)
    постфиксное-выражение.идентификатор
    постфиксное-выражение->идентификатор
    постфиксное-выражение ++
    постфиксное-выражение —-

список-аргументов-выражений:
    выражение-присваивание
    список-аргументов-выражений , выражение-присваивание
```

A7.3.1. Обращение к элементам массива

Постфиксное выражение, за которым следует выражение в квадратных скобках, есть постфиксное выражение, обозначающее обращение к индексируемому массиву. Одно из этих двух выражений должно принадлежать типу "указатель на *T*", где *T* - некоторый тип, а другое - целочисленному типу; тип результата индексирования есть *T*. Выражение E1[E2] по определению идентично выражению *((E1)+(E2)). Подробности см. в A8.6.

A7.3.2. Вызов функции

Вызов функции есть постфиксное выражение (оно называется именующим выражением функции - *function designator*), за которым следуют скобки, содержащие (возможно пустой) список разделенных запятыми выражений-присваиваний (A7.17), представляющих собой аргументы этой функции. Если постфиксное выражение — идентификатор, не объявленный в текущей области видимости, то считается, что этот идентификатор как бы неявно описан объявлением

```
extern int identifier();
```

помещенным в самом внутреннем блоке, содержащем вызов соответствующей функции. Постфиксное выражение (после, возможно неявного, описания и генерации указателя, см. A7.1) должно иметь тип "указатель на функцию, возвращающую *T*", где *T* - тип возвращаемого значения.

> В первой версии языка для именующего выражения функции допускался только тип "функция", и чтобы вызвать функцию через указатель, требовался явный оператор *. ANSI-стандарт поощряет практику некоторых существующих компиляторов, разрешающих иметь одинаковый синтаксис для обращения просто к функции и обращения к функции, специфицированной указателем. Возможность применения старого синтаксиса остается.

Термин *аргумент* используется для выражения, задаваемого в вызове функции; термин *параметр* - для обозначения получаемого ею объекта (или его идентификатора) в определении или объявлении функции. Вместо этих понятий иногда встречаются термины "фактический аргумент (параметр)" и "формальный аргумент (параметр)", имеющие те же смысловые различия.

При вызове функции каждый ее аргумент копируется; передача аргументов осуществляется строго через их значения. Функции разрешается изменять значения своих параметров, которые являются лишь копиями аргументов- выражений, но эти изменения не могут повлиять на значения самих аргументов. Однако можно передать указатель, чтобы позволить функции изменить значение объекта, на который указывает этот указатель.

Имеются два способа объявления функции. В новом способе типы параметров задаются явно и являются частью типа функции; такое объявление называется прототипом функции. При старом способе типы параметров не указываются. Способы объявления функций обсуждаются в A8.6.3 и A10.1.

Если вызов находится в области видимости объявления, написанного по-старому, каждый его аргумент подвергается операции повышения типа: для целочисленных аргументов осуществляется целочисленное повышение (A6.1), а для аргументов типа *float* - преобразование в *double*. Если число аргументов не соответствует количеству параметров в определении функции или если типы аргументов после повышения не согласуются с типами соответствующих параметров, результат вызова не определен. Критерий согласованности типов зависит от способа определения функции (старого или нового). При старом способе сравниваются повышенный тип аргумента в вызове и повышенный тип соответствующего параметра; при новом способе повышенный тип аргумента и тип параметра (без его

повышения) должны быть одинаковыми.

Если вызов находится в области видимости объявления, написанного по-новому, аргументы преобразуются, как если бы они присваивались переменным, имеющим типы соответствующих параметров прототипа. Число аргументов должно совпадать с числом явно описанных параметров, если только список параметров не заканчивается многоточием (, ...). В противном случае число аргументов должно быть больше числа параметров или равно ему; "скрывающиеся" под многоточием аргументы подвергаются операции повышения типа (так, как это было описано в предыдущем абзаце). Если определение функции задано по-старому, то типы параметров в прототипе, которые неявно присутствуют в вызове, должны соответствовать типам параметров в определении функции после их повышения.

> Эти правила особенно усложнились из-за того, что они призваны обслуживать смешанный способ (старого с новым) задания функций. По возможности его следует избегать.

Очередность вычисления аргументов не определяется, в разных компиляторах она различна. Однако гарантируется, что аргументы и именующее выражение функции вычисляются полностью (включая и побочные эффекты) до входа в нее. Любая функция допускает рекурсивное обращение.

A7.3.3. Обращение к структурам

Постфиксное выражение, за которым стоит точка с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть структурой или объединением, а идентификатор - именем элемента структуры или объединения. Значение - именованный элемент структуры или объединения, а тип значения - тип элемента структуры или объединения. Выражение является *lvalue*, если первое выражение - *lvalue* и если тип второго выражения - не "массив".

Постфиксное выражение, за которым стоит стрелка (составленная из знаков - и >) с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть указателем на структуру (объединение), а идентификатор - именем элемента структуры (объединения). Результат - именованный элемент структуры (объединения), на которую указывает указатель, а тип значения - тип элемента структуры (объединения); результат - *lvalue*, если тип не есть "массив".

Таким образом, выражение E1->MOS означает то же самое, что и выражение (*E1).MOS. Структуры и объединения рассматриваются в A8.3.

> В первом издании книги уже было приведено правило, по которому имя элемента должно принадлежать структуре или объединению, упомянутому в постфиксном выражении. Там, однако, оговаривалось, что оно не является строго обязательным. Последние компиляторы и ANSI делают его обязательным.

A7.3.4. Постфиксные операторы инкремента и декремента

Постфиксное выражение, за которым следует ++ или --, есть постфиксное выражение. Значением такого выражения является значение его операнда. После того как значение было взято, операнд увеличивается (++) или уменьшается (--) на 1. Операнд должен быть *lvalue*; информация об ограничениях, накладываемых на операнд, и деталях операций содержится в A7.7, где обсуждаются аддитивные операторы, и в A7.17, где рассматривается присваивание. Результат инкрементирования или декрементирования не есть *lvalue*.

A7.4. Унарные операторы

Выражения с унарными операторами выполняются справа налево.

```
унарное-выражение:
    постфиксное-выражение
    ++ унарное-выражение
    -- унарное-выражение
    унарный-оператор выражение-приведенное-к-типу
    sizeof унарное-выражение
    sizeof (имя-типа)
```

```
унарный-оператор: один из
    &   *   +   -   ~   !
```

A7.4.1. Префиксные операторы инкремента и декремента

Унарное выражение, перед которым стоит ++ или --, есть унарное выражение. Операнд увеличивается (++) или уменьшается (--) на 1.

Значением выражения является значение его операнда после увеличения (уменьшения). Операнд всегда должен быть *lvalue*; информация об ограничениях на операнд и о деталях операции содержится в A7.7, где обсуждаются аддитивные операторы, и в A7.17, где рассматривается присваивание. Результат инкрементирования и декрементирования не есть *lvalue*.

A7.4.2. Оператор получения адреса

Унарный оператор **&** обозначает операцию получения адреса своего операнда. Операнд должен быть либо *lvalue*, не ссылающимся ни на битовое поле, ни на объект, объявленный как *register*, либо иметь тип "функция". Результат - указатель на объект (или функцию), адресуемый этим *lvalue*. Если тип операнда есть *T*, то типом результата является "указатель на *T*".

A7.4.3. Оператор косвенного доступа

Унарный оператор ***** обозначает операцию косвенного доступа (раскрытия указателя), возвращающую объект (или функцию), на который указывает ее операнд. Результат есть *lvalue*, если операнд - указатель на объект арифметического типа или на объект типа "структура", "объединение" или "указатель". Если тип выражения - "указатель на *T*", то тип результата - *T*.

A7.4.4. Оператор унарный плюс

Операнд унарного + должен иметь арифметический тип, результат - значение операнда. Целочисленный операнд подвергается целочисленному повышению. Типом результата является повышенный тип операнда.

> Унарный + был добавлен для симметрии с унарным -.

A7.4.5. Оператор унарный минус

Операнд для унарного минуса должен иметь арифметический тип, результат - значение операнда с противоположным знаком. Целочисленный операнд подвергается целочисленному повышению. Отрицательное значение от беззнаковой величины вычисляется вычитанием из *nmax+1* приведенного к повышенному типу операнда, где *nmax* - максимальное число повышенного типа; однако минус нуль есть нуль. Типом результата будет повышенный тип операнда.

A7.4.6. Оператор побитового отрицания

Операнд оператора ~ должен иметь целочисленный тип, результат - дополнение операнда до единиц по всем разрядам. Выполняется целочисленное повышение типа операнда. Если операнд беззнаковый, то результат получается вычитанием его значения из самого большого числа повышенного типа. Если операнд знаковый, то результат вычисляется посредством приведения "повышенного операнда" к беззнаковому типу, выполнения операции ~ и обратного приведения его к знаковому типу. Тип результата - повышенный тип операнда.

A7.4.7. Оператор логического отрицания

Операнд оператора **!** должен иметь арифметический тип или быть указателем. Результат равен 1, если сравнение операнда с 0 дает истину, и равен 0 в противном случае. Тип результата - *int*.

A7.4.8. Оператор определения размера sizeof

Оператор **sizeof** дает число байтов, требуемое для хранения объекта того типа, который имеет его операнд. Операнд - либо выражение (которое не вычисляется), либо имя типа, записанное в скобках. Примененный к *char* оператор *sizeof* дает 1. Для массива результат равняется общему количеству байтов в массиве, для структуры или объединения - числу байтов в объекте, включая и байты- заполнители, которые понадобились бы, если бы из элементов составлялся массив. Размер массива из *n* элементов всегда равняется *n*, помноженному на размер отдельного его элемента. Данный оператор нельзя применять к операнду типа "функция", к незавершенному типу и к битовому полю. Результат - беззнаковая целочисленная константа: конкретный ее тип зависит от реализации. В стандартном заголовочном файле <**stddef.h**>(см. приложение В) этот тип определяется под именем **size_t**.

A7.5. Оператор приведения типа

Имя типа, записанное перед унарным выражением н скобках, вызывает приведение значения этого выражения к указанному типу.

```
выражение-приведенное-к-типу:
    унарное-выражение
    (имя-типа) выражение-приведенное-к-типу
```

Данная конструкция называется *приведением*. Имена типов даны в A8.8. Результат преобразований описан в A6. Выражение с приведением типа не является *lvalue*.

A7.6. Мультипликативные операторы

Мультипликативные операторы *****, / и **%** выполняются слева направо.

```
мультипликативное-выражение:
    выражение-приведенное-к-типу
    мультипликативное-выражение * выражение-приведенное-к-типу
    мультипликативное-выражение / выражение-приведенное-к-типу
    мультипликативное-выражение % выражение-приведенное-к-типу
```

Операнды операторов * и / должны быть арифметического типа, оператора % - целочисленного типа. Над операндами осуществляются обычные арифметические преобразования, которые приводят их значения к типу результата.

Бинарный оператор * обозначает умножение.

Бинарный оператор / получает частное, а % - остаток от деления первого операнда на второй; если второй операнд есть 0, то результат не определен. В противном случае всегда выполняется соотношение: (a / b) * b + a % b равняется a. Если оба операнда не отрицательные, то остаток не отрицательный и меньше делителя; в противном случае стандарт гарантирует только одно: что абсолютное значение остатка меньше абсолютного значения делителя.

A7.7. Аддитивные операторы

Аддитивные операторы + и **-** выполняются слева направо. Если операнды имеют арифметический тип, то осуществляются обычные арифметические преобразования. Для каждого оператора существует еще несколько дополнительных сочетаний типов.

```
аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение - мультипликативное-выражение
```

Результат выполнения оператора + есть сумма его операндов. Указатель на объект в массиве можно складывать с целочисленным значением. При этом последнее преобразуется в адресное смещение посредством умножения его на размер объекта, на который ссылается указатель. Сумма является указателем на объект того же типа; только ссылается этот указатель на другой объект того же массива, отстоящий от первоначального соответственно вычисленному смещению. Так, если *p* - указатель на объект в массиве, то *p+1* - указатель на следующий объект того же массива. Если полученный в результате суммирования указатель указывает за границы массива, то, кроме случая, когда он указывает на место, находящееся непосредственно за концом массива, результат будет

неопределенным.

> Возможность для указателя указывать на элемент, расположенный сразу за концом массива, является новой. Тем самым узаконена общепринятая практика организации циклического перебора элементов массива.

Результат выполнения оператора - (минус) есть разность операндов. Из указателя можно вычитать значение любого целочисленного типа с теми же преобразованиями и при тех же условиях, что и в сложении.

Если к двум указателям на объекты одного и того же типа применить оператор вычитания, то в результате получится целочисленное значение со знаком, представляющее собой расстояние между объектами, на которые указывают эти указатели: указатель на следующий объект на 1 больше указателя на предыдущий объект. Тип результата зависит от реализации: в стандартном заголовочном файле <**stddef.h**> он определен под именем **ptrdiff_t**. Значение не определено, если указатели указывают на объекты не одного и того же массива; однако если *p* указывает на последний элемент массива, то *p+1-p* имеет значение, равное 1.

A7.8. Операторы сдвига

Операторы сдвига << и >> выполняются слева направо. Для обоих операторов каждый операнд должен иметь целочисленный тип, и каждый из них подвергается целочисленному повышению. Тип результата совпадает с повышенным типом левого операнда. Результат не определен, если правый операнд отрицателен или его значение превышает число битов в типе левого выражения или равно ему.

```
сдвиговое-выражение:
    аддитивное -выражение
    сдвиговое-выражение >> аддитивное-выражение
    сдвиговое-выражение << аддитивное-выражение
```

Значение E1<<E2 равно значению E1 (рассматриваемому как цепочка битов), сдвинутому влево на E2 битов; при отсутствии переполнения такая операция эквивалентна умножению на 2^{E2}. Значение E1>>E2 равно значению E1, сдвинутому вправо на E2 битовые позиции. Если E1 - беззнаковое или имеет неотрицательное значение, то правый сдвиг эквивалентен делению на 2^{E2}, в противном случае результат зависит от реализации.

A7.9. Операторы отношения

Операторы отношения выполняются слева направо, однако это свойство едва ли может оказаться полезным: согласно грамматике языка выражение a < b < c трактуется так же, как (a < b) < c, а результат вычисления a < b может быть только 0 или 1.

```
выражение-отношения:
    сдвиговое -выражение
    выражение-отношения < сдвиговое-выражение
    выражение-отношения > сдвиговое-выражение
    выражение-отношения <= сдвиговое-выражение
    выражение-отношения >= сдвиговое-выражение
```

Операторы: < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) — все выдают 0, если специфицируемое отношение ложно, и 1, если оно истинно. Тип результата - *int*. Над арифметическими операндами выполняются обычные арифметические преобразования. Можно сравнивать указатели на объекты одного и того же типа (без учета квалификаторов); результат будет зависеть от относительного расположения в памяти. Допускается, однако, сравнение указателей на разные части одного и того же объекта: если два указателя указывают на один и тот же простой объект, то они равны; если они указывают на элементы одной структуры, то указатель на элемент с более поздним объявлением в структуре больше; если указатели указывают на элементы одного и того же объединения, то они равны; если указатели указывают на элементы некоторого массива, то сравнение этих указателей эквивалентно сравнению их индексов. Если *p* указывает на последний элемент массива, то *p+1* больше, чем *p*, хотя *p+1* указывает за границы массива. В остальных случаях результат сравнения не определен.

Эти правила несколько ослабили ограничения, установленные в первой редакции языка. Они позволяют сравнивать указатели на различные элементы структуры и объединения и легализуют сравнение с указателем на место, которое расположено непосредственно за концом массива.

A7.10. Операторы равенства

```
выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения
```

Операторы == (равно) и **!**= (не равно) аналогичны операторам отношения с той лишь разницей, что имеют более низкий приоритет. (Таким образом, a < b == c < d есть 1 тогда и только тогда, когда отношения a < b и c < d или оба истинны, или оба ложны.)

Операторы равенства подчиняются тем же правилам, что и операторы отношения. Кроме того, они дают возможность сравнивать указатель с целочисленным константным выражением, значение которого равно нулю, и с указателем на *void* (см. A6.6.).

A7.11. Оператор побитового И

```
И-выражение:
    выражение-равенства
    И-выражение & выражение-равенства
```

Выполняются обычные арифметические преобразования: результат - побитовое **AND** операндов. Оператор применяется только к целочисленным операндам.

A7.12. Оператор побитового исключающего ИЛИ

```
исключающее-ИЛИ-выражение:
    И-выражение
    исключающее-ИЛИ-выражение ^ И-выражение
```

Выполняются обычные арифметические преобразования; результат – побитовое **XOR** операндов. Оператор применяется только к целочисленным операндам.

A7.13. Оператор побитового ИЛИ

```
ИЛИ выражение:
    исключающее-ИЛИ-выражение
    ИЛИ-выражение | исключающее-ИЛИ-выражение
```

Выполняются обычные арифметические преобразования; результат - побитовое **OR** операндов. Оператор применяется только к целочисленным операндам.

A7.14. Оператор логического И

```
логическое-И-выражение:
    ИЛИ-выражение
    логическое-И-выражение && ИЛИ-выражение
```

Операторы **&&** выполняются слева направо. Оператор && выдает 1, если оба операнда не равны нулю, и 0 в противном случае. В отличие от &, && гарантирует, что вычисления будут проводиться слева направо: вычисляется первый операнд со всеми побочными эффектами; если он равен 0, то значение выражения есть 0. В противном случае вычисляется правый операнд, и, если он ранен 0, то значение выражения есть 0, в противном случае оно равно 1.

Операнды могут принадлежать к разным типам, но при этом каждый из них должен иметь либо арифметический тип, либо быть указателем. Тип результата - *int*.

A7.15. Оператор логического ИЛИ

```
логическое-ИЛИ-выражение:
    логическое-И-выражение
    логическое-ИЛИ-выражение || логическое-И-выражение
```

Операторы || выполняются слева направо. Оператор || выдает 1, если по крайней мере один из операндов не равен нулю, и 0 в противном случае. В отличие от |, оператор || гарантирует, что вычисления будут проводиться слева направо: вычисляется первый операнд, включая все побочные эффекты; если он не равен 0, то значение выражения есть 1. В противном случае вычисляется правый операнд, и если он не равен 0, то значение выражения есть 1, в противном случае оно равно 0.

Операнды могут принадлежать разным типам, но операнд должен иметь либо арифметический тип, либо быть указателем. Тип результата - *int*.

A7.16. Условный оператор

```
условное-выражение:
    логическое-ИЛИ-выражение
    логическое-ИЛИ-выражение ? выражение : условное-выражение
```

Вычисляется первое выражение, включая все побочные эффекты; если оно не равно 0, то результат есть значение второго выражения, в противном случае - значение третьего выражения. Вычисляется только один из двух последних операндов: второй или третий. Если второй и третий операнды арифметические, то выполняются обычные арифметические преобразования, приводящие к некоторому общему типу, который и будет типом результата. Если оба операнда имеют тип *void*, или являются структурами или объединениями одного и того же типа, или представляют собой указатели на объекты одного и того же типа, то результат будет иметь тот же тип, что и операнды. Если один из операндов имеет тип "указатель", а другой является константой 0, то 0 приводится к типу "указатель", этот же тип будет иметь и результат. Если один операнд является указателем на *void*, а второй - указателем другого типа, то последний преобразуется в указатель на *void*, который и будет типом результата.

При сравнении типов указателей квалификаторы типов (A8.2) объектов, на которые указатели ссылаются, во внимание не принимаются, но тип результата наследует квалификаторы обеих ветвей условного выражения.

A7.17. Выражения присваивания

Существует несколько операторов присваивания; они выполняются справа налево.

```
выражение-присваивания:
    условное-выражение
    унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из
    =
    *=
    /=
    %=
    +=
    -=
    <<=
    >>=
    &=
    ^=
    |=
```

Операторы присваивания в качестве левого операнда требуют *lvalue*, причем модифицируемого; это значит, что оно не может быть массивом, или иметь незавершенный тип, или быть функцией. Тип левого операнда, кроме того, не может иметь квалификатора *const*; и, если он является структурой или объединением, в них не должно быть элементов или подэлементов (для вложенных структур или объединений) с квалификаторами *const.*

Тип выражения присваивания соответствует типу его левого операнда, а значение равно значению его левого операнда после завершения присваивания.

В простом присваивании с оператором = значение выражения замещает объект, к которому обращается *lvalue*. При этом должно выполняться одно из следующих условий: оба операнда имеют арифметический тип (если типы операндов разные, правый операнд приводится к типу левого операнда); оба операнда есть структуры или объединения одного и того же типа; один операнд есть

указатель, а другой - указатель на *void*; левый операнд - указатель, а правый - константное выражение со значением 0; оба операнда - указатели на функции или объекты, имеющие одинаковый тип (за исключением возможного отсутствия *const* или *volatile* у правого операнда).

Выражение E1 *op*= E2 эквивалентно выражению E1 = E1 *op*(E2)с одним исключением: E1 вычисляется только один раз.

A7.18. Оператор запятая

```
выражение:
    выражение-присваивания
    выражение , выражение-присваивания
```

Два выражения, разделенные запятой, вычисляются слева направо, и значение левого выражения отбрасывается. Тип и значение результата совпадают с типом и значением правого операнда. Вычисление всех побочных эффектов левого операнда завершается перед началом вычисления правого операнда. В контексте, в котором запятая имеет специальное значение, например в списках аргументов функций (A7.3.2) или в списках инициализаторов (A8.7) (здесь в качестве синтаксических единиц фигурируют выражения присваивания), оператор запятая может появиться только в группирующих скобках. Например, в

```
f(a, (t=3, t+2), c)
```

три аргумента, из которых второй имеет значение 5.

A7.19. Константные выражения

Синтаксически, константное выражение - это выражение с ограниченным подмножеством операторов:

```
константное-выражение:
    условное-выражение
```

При указании *case*-меток в переключателе, задании границ массивов и длин полей битов, на месте значений перечислимых констант и инициализаторов, а также в некоторых выражениях для препроцессора требуются выражения, вычисление которых приводит к константе.

Константные выражения не могут содержать присваиваний, операторов инкрементирования и декрементирования, вызовов функций и операторов-запятых; перечисленные ограничения не распространяются на операнд оператора *sizeof*. Если требуется получить целочисленное константное выражение, то его операнды должны состоять из целых, перечислимых (*enum*), символьных констант и констант с плавающей точкой; операции приведения должны специфицировать целочисленный тип, а любая константа с плавающей точкой - приводиться к целому. Из этого следует, что в константном выражении не может быть массивов, операций косвенного обращения (раскрытия указателя), получения адреса и доступа к полям структуры. (Однако для *sizeof* возможны операнды любого вида.)

Для константных выражений в инициализаторах допускается большая свобода; операндами могут быть константы любого типа, а к внешним или статическим объектам и внешним и статическим массивам, индексируемым константными выражениями, возможно применять унарный оператор *&*. Унарный оператор *&* может также неявно присутствовать при использовании массива без индекса или функции без списка аргументов. Вычисление инициализатора должно давать константу или адрес ранее объявленного внешнего или статического объекта плюс-минус константа.

Меньшая свобода допускается для целочисленных константных выражений, используемых после *#if*: не разрешаются *sizeof*-выражения, константы типа *enum* и операции приведения типа. (см. A12.5.)

A8. Объявления

То, каким образом интерпретируется каждый идентификатор, специфицируется объявлениями; они не всегда резервируют память для описываемых ими идентификаторов. Объявления, резервирующие память, называются определениями и имеют следующий вид:

```
объявление:
    спецификаторы-объявления список-инициализаторов-объявителей(необ)
```

Объявители в *списке-инициализаторов-объявителей* содержат объявляемые идентификаторы; *спецификаторы-объявления* представляют собой последовательности, состоящие из спецификаторов типа и класса памяти.

```
спецификаторы-объявления:
    спецификатор-класса-памяти спецификаторы-объявления(необ)
    спецификатор-типа спецификаторы-объявления(необ)
    квалификатор-типа спецификаторы-объявления(необ)

список-инициализаторов-объявителей:
    инициализатор-объявитель
    список-инициализаторов-объявителей , инициализатор-объявитель

инициализатор-объявитель:
    объявитель
    объявитель = инициализатор
```

Объявители содержат подлежащие объявлению имена. Мы рассмотрим их позже, в A8.5. Объявление должно либо иметь по крайней мере один объявитель, либо его спецификатор типа должен определять тег структуры или объединения, либо - задавать элементы перечисления; пустое объявление недопустимо.

A8.1. Спецификаторы класса памяти

Класс памяти специфицируется следующим образом:

```
спецификатор-класса-памяти:
    auto
    register
    static
    extern
    typedef
```

Смысл классов памяти обсуждался в A4.

Спецификаторы *auto* и *register* дают объявляемым объектам класс автоматической памяти, и эти спецификаторы можно применять только внутри функции. Объявления с *auto* и *register* одновременно являются определениями и резервируют намять. Спецификатор *register* эквивалентен *auto*, но содержит подсказку, сообщающую, что в программе объявленные им объекты используются интенсивно. На регистрах может быть размещено лишь небольшое число объектов, причем определенного типа: указанные ограничения зависят от реализации. В любом случае к *register*- объекту нельзя применять (явно или неявно) унарный оператор *&*.

> Новым является правило, согласно которому вычислять адрес объекта класса *register* нельзя, а класса *auto* можно.

Спецификатор *static* дает объявляемым объектам класс статической памяти, он может использоваться и внутри, и вне функций. Внутри функции этот спецификатор вызывает выделение памяти и служит определением; его роль вне функций будет объяснена в A11.2.

Объявление со спецификатором *extern*, используемое внутри функции, объявляет, что для объявляемого объекта где-то выделена память; о ее роли вне функций будет сказано в A11.2.

Спецификатор *typedef* не резервирует никакой памяти и назван спецификатором класса памяти из соображений стандартности синтаксиса; речь об этом спецификаторе пойдет в A8.9.

Объявление может содержать не более одного спецификатора класса памяти. Если он в объявлении отсутствует, то действуют следующие правила: считается, что объекты, объявляемые внутри функций, имеют класс *auto*; функции, объявляемые внутри функций, - класс *extern*; объекты и функции, объявляемые вне функций, - статические и имеют *внешние связи* (см. A10, A11).

A8.2. Спецификаторы типа

Спецификаторы типа определяются следующим образом:

```
спецификатор-типа:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
  структуры-или-объединения-спецификатор
  спецификатор-перечисления
  typedef-имя
```

Вместе с *int* допускается использование еще какого-то одного слова - *long* или *short*; причем сочетание *long int* имеет тот же смысл, что и просто *long*: аналогично *short int* - то же самое, что и *short*. Слово *long* может употребляться вместе с *double*. С *int* и другими его модификациями (*short*, *long* или *char*) разрешается употреблять одно из слов *signed* или *unsigned*. Любое из последних может использоваться самостоятельно, в этом случае подразумевается *int*.

Спецификатор *signed* бывает полезен, когда требуется обеспечить, чтобы объекты типа *char* имели знак; его можно применять и к другим целочисленным типам, но в этих случаях он избыточен.

За исключением описанных выше случаев объявление не может содержать более одного спецификатора типа. Если в объявлении нет ни одного спецификатора типа, то имеется в виду тип *int*.

Для указания особых свойств объявляемых объектов предназначаются квалификаторы:

```
квалификатор-типа:
    const
    volatile
```

Квалификаторы типа могут употребляться с любым спецификатором типа. Разрешается инициализировать *const*-объект, однако присваивать ему что-либо в дальнейшем запрещается. Смысл квалификатора *volatile* зависит от реализации.

> Средства *const* и *volatile* (изменчивый) введены стандартом ANSI. Квалификатор *const* применяется, чтобы разместить объекты в памяти, открытой только на чтение (ПЗУ), или чтобы способствовать возможной оптимизации. Назначение квалификатора *volatile* - подавить оптимизацию, которая без этого указания могла бы быть проведена. Например, в машинах, где адреса регистров ввода-вывода отображены на адресное пространство памяти, указатель на регистр некоторого устройства мог бы быть объявлен как *volatile*, чтобы запретить компилятору экономить очевидно избыточную ссылку через указатель. Компилятор может игнорировать указанные квалификаторы, однако обязан сигнализировать о явных попытках изменить значение *const*-объектов.

A8.3. Объявления структур и объединений

Структура - это объект, состоящий из последовательности именованных элементов различных типов. *Объединение* - объект, который в каждый момент времени содержит один из нескольких элементов различных типов. Объявления структур и объединений имеют один и тот же вид.

```
спецификатор структуры-или-объединения:
    структуры-или-объединения идентификатор_необ {список-объявлений-структуры}
    структуры-или-объединения идентификатор

структура-или-объединение:
    struct
    union
```

Список-объявлений-структуры является последовательностью объявлений элементов структуры или объединения:

```
список-объявлений-структуры:
    объявление-структуры
    список-объявлений-структуры объявление-структуры

объявление-структуры:
    список-спецификаторов-квалификаторов список-структуры-объявителей;
```

```
список-спецификаторов-квалификаторов:
    спецификатор-типа список-спецификаторов-квалификаторов_необ
    квалификатор-типа список-спецификаторов-квалификаторов_необ

список-структуры-объявителей:
    структуры-объявитель
    список-структуры-объявителей , структуры-объявитель
```

Обычно *объявление-структуры* является просто объявлением для элементов структуры или объединения. Элементы структуры, в свою очередь, могут состоять из заданного числа разрядов (битов). Такой элемент называется *битовым полем* или просто полем. Его размер отделяется от имени поля двоеточием:

```
структуры-объявитель:
    объявитель
    объявитель_необ : константное-выражение
```

Спецификатор типа, имеющий вид

```
структуры-или-объединения идентификатор { список-объявлений-структуры }
```

объявляет идентификатор *тегом* структуры или объединения, специфицированных списком. Последующее объявление в той же или внутренней области видимости может обращаться к тому же типу, используя в спецификаторе тег без списка:

```
структуры-или-объединения идентификатор
```

Если спецификатор с тегом, но без списка появляется там, где тег не объявлен, специфицируется *незавершенный тип*. Объекты с незавершенным типом структуры или объединения могут упоминаться в контексте, где не требуется знать их размер — например в объявлениях (но не определениях) для описания указателя или создания *typedef*, но не в иных случаях. Тип становится завершенным при появлении последующего спецификатора с этим тегом, содержащего список объявлений. Даже в спецификаторах со списком объявляемый тип структуры или объединения является незавершенным внутри списка и становится завершенным только после появления символа }, заканчивающего спецификатор.

Структура не может содержать элементов незавершенного типа. Следовательно, невозможно объявить структуру или объединение, которые содержат сами себя. Однако, кроме придания имени типу структуры или объединения, тег позволяет определять структуры, обращающиеся сами к себе; структура или объединение могут содержать указатели на самих себя, поскольку указатели на незавершенные типы объявлять можно.

Особое правило применяется к объявлениям вида

```
структуры-или-объединения идентификатор;
```

которые объявляют структуру или объединение, но не имеют списка объявления и объявителя. Даже если идентификатор имеет тег структуры или объединения во внешней области видимости (A11.1), это объявление делает идентификатор тегом новой структуры или объединения незавершенного типа во внутренней области видимости.

> Это невразумительное правило - новое в ANSI. Оно предназначено для взаимно рекурсивных структур, объявленных во внутренней области видимости, но теги которых могут быть уже объявлены во внешней области видимости.

Спецификатор структуры или объединения со списком, но без тега создает уникальный тип, к которому можно обращаться непосредственно только в объявлении, частью которого он является.

Имена элементов и тегов не конфликтуют друг с другом или обычными переменными. Имя элемента не может появляться дважды в одной и той же структуре или объединении, но тот же элемент можно использовать в разных структурах или объединениях.

> В первой редакции этой книги имена элементов структуры и объединения не связывались со своими родителями. Однако в компиляторах эта связь стала обычной задолго до появления

стандарта ANSI.

Элемент структуры или объединения, не являющийся полем, может иметь любой тип объекта. Поле (которое не имеет объявителя и, следовательно, может быть безымянным) имеет тип *int*, *unsigned int* или *signed int* и интерпретируется как объект целочисленного типа указанной в битах длины. Считается ли поле *int* знаковым или беззнаковым, зависит от реализации. Соседний элемент-поле упаковывается в ячейки памяти в зависимости от реализации в зависящем от реализации направлении. Когда следующее за полем другое поле не влезает в частично заполненную ячейку памяти, оно может оказаться разделенным между двумя ячейками, или ячейка может быть забита балластом. Безымянное поле нулевой ширины обязательно приводит к такой забивке, так что следующее поле начнется с края следующей ячейки памяти.

> Стандарт ANSI делает поля еще более зависимыми от реализации, чем в первой редакции книги. Чтобы хранить битовые поля в "зависящем от реализации" виде без квалификации, желательно прочитать правила языка. Структуры с битовыми полями могут служить переносимым способом для попытки уменьшить размеры памяти под структуру (вероятно, ценой увеличения кода программы и времени на доступ к полям) или непереносимым способом для описания распределения памяти на битовом уровне. Во втором случае необходимо понимать правила местной реализации.

Элементы структуры имеют возрастающие по мере объявления элементов адреса. Элементы структуры, не являющиеся полями, выравниваются по границам адресов в зависимости от своего типа; таким образом, в структуре могут быть безымянные дыры. Если указатель на структуру приводится к типу указателя на ее первый элемент, результат указывает на первый элемент.

Объединение можно представить себе как структуру, все элементы которой начинаются со смещением 0 и размеры которой достаточны для хранения любого из элементов. В любой момент времени в объединении хранится не больше одного элемента. Если указатель на объединение приводится к типу указателя на один из элементов, результат указывает на этот элемент.

Вот простой пример объявления структуры:

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Эта структура содержит массив из 20 символов, число типа *int* и два указателя на подобную структуру. Если дано такое объявление, то

```
struct tnode s, *sp;
```

объявит *s* как структуру заданного вида, а *sp* - как указатель на такую структуру. Согласно приведенным определениям выражение

```
sp->count
```

обращается к элементу *count* в структуре, на которую указывает *sp*;

```
s.left
```

- указатель на левое поддерево в структуре *s*, а

s.right->tword[0]

- это первый символ из *tword* - элемента правого поддерева *s*.

Вообще говоря, невозможно проконтролировать, тот ли используется элемент объединения, которому последний раз присваивалось значение. Однако гарантируется выполнение правила, облегчающего работу с элементами объединения: если объединение содержит несколько структур, начинающихся с общей для них последовательности данных, и если объединение в текущий момент содержит одну из этих структур, то к общей части данных разрешается обращаться через любую из указанных структур.

Так, правомерен следующий фрагмент программы:

```
union {
    struct {
        int type;
    } n;

    struct {
        int type;
        int intnode;
    } ni;

    struct {
        int type;
        float floatnode;
    } nf;
} u;

...

u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...

if (u.n.type == FLOAT)
... sin(u.nf.floatnode) ...
```

A8.4. Перечисления

Перечисления - это уникальный тип, значения которого покрываются множеством именованных констант, называемых перечислителями. Вид спецификатора перечисления заимствован у структур и объединений.

спецификатор-перечисления:
    enum *идентификатор*$_{необ}$ {*список-перечислителей*}
    enum *идентификатор*

список-перечислителей:
    перечислитель
    список-перечислителей , *перечислитель*

перечислитель:
    идентификатор
    идентификатор = *константное-выражение*

Идентификаторы, входящие в список перечислителей, объявляются константами типа *int* и могут употребляться везде, где требуется константа. Если в этом списке нет ни одного перечислителя со знаком =, то значения констант начинаются с 0 и увеличиваются на 1 по мере чтения объявления слева направо. Перечислитель со знаком = дает соответствующему идентификатору значение; последующие идентификаторы продолжают прогрессию от заданного значения.

Имена перечислителей, используемые в одной области видимости, должны отличаться друг от друга и от имен обычных переменных, однако их значения могут и совпадать.

Роль идентификатора в *переч-спецификаторе* аналогична роли тега структуры в *структ-спецификаторе*: он является именем некоторого конкретного перечисления. Правила для списков и переч-спецификаторов (с тегами и без) те же, что и для спецификаторов структур или объединений, с той лишь оговоркой, что элементы перечислений не бывают незавершенного типа; тег переч-спецификатора без списка перечислителей должен иметь в пределах области видимости спецификатор со списком.

В первой версии языка перечислений не было, но они уже несколько лет применяются.

A8.5. Объявители

Объявители имеют следующий синтаксис:

объявитель:
    указатель$_{необ}$ *собственно-объявитель*

собственно-объявитель:
    идентификатор
    (*объявитель*)

```
    собственно-объявитель [константное-выражение_необ]
    собственно-объявитель (список-типов-параметров)
    собственно-объявитель (список-идентификаторов_необ)

указатель:
    * список-квалификаторов-типа_необ
    * список-квалификаторов-типа_необ указатель

список-квалификаторов-типа:
    квалификатор-типа
    список-квалификаторов-типа квалификатор-типа
```

У структуры объявителя много сходных черт со структурой подвыражений, поскольку в объявителе, как и в подвыражении, допускаются операции косвенного обращения, обращения к функции и получения элемента массива (с тем же порядком применения).

A8.6. Что означают объявители

Список объявителей располагается сразу после спецификаторов типа и указателя класса памяти. Главный элемент любого объявителя - это объявляемый им идентификатор; в простейшем случае объявитель из него одного и состоит, что отражено в первой строке продукции грамматики с именем *собственно- объявитель*. Спецификаторы класса памяти относятся непосредственно к идентификатору, а его тип зависит от вида объявителя. Объявитель следует воспринимать как утверждение: если в выражении идентификатор появляется в том же контексте, что и в объявителе, то он обозначает объект специфицируемого типа.

Если соединить спецификаторы объявления, относящиеся к типу (A8.2), и некоторый конкретный объявитель, то объявление примет вид "T D", где *T* - тип, а *D* - объявитель. Эта запись индуктивно придает тип идентификатору любого объявителя.

В объявлении T D, где *D* - просто идентификатор, тип идентификатора есть *T*.

В объявлении T D, где *D* имеет вид

```
( D1 )
```

тип идентификатора в *D1* тот же, что и в *D*. Скобки не изменяют тип, но могут повлиять на результаты его "привязки" к идентификаторам в сложных объявителях.

A8.6.1. Объявители указателей

В объявления T D, где *D* имеет вид

```
* список-квалификаторов-типа_необ D1
```

а тип идентификатора объявления T D1 есть "*модификатор-типа T*", тип идентификатора *D* есть "*модификатор-типа список-квалификаторов-типа указатель на T*". Квалификаторы, следующие за *, относятся к самому указателю, а не к объекту, на который он указывает. Рассмотрим, например, объявление

```
int *ap[];
```

Здесь *ap[]* играет роль *D1*; объявление *int ap[]* следует расшифровать (см. ниже) как "массив из *int*": список квалификаторов типа здесь пуст, а модификатор типа есть "*массив из*". Следовательно, на самом деле объявление *ap* гласит: "массив из указателей на *int*". Вот еще примеры объявлений:

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

В них объявляются целое *i* и указатель на целое *pi*. Значение указателя *cpi* неизменно; *cpi* всегда будет указывать в одно и то же место, даже если значение, на которое он указывает, станет иным. Целое *ci* есть константа, оно измениться не может (хотя может инициализироваться, как в данном случае). Тип указателя *pci* произносится как "указатель на *const int*"; сам указатель можно изменить; при этом он будет указывать на другое место, но значение, на которое он будет указывать, с помощью *pci* изменить нельзя.

A8.6.2. Объявители массивов

В объявления T D, где D имеет вид

D1 [*константное-выражение*$_{необ}$]

и где тип идентификатора объявления T D1 есть "*модификатор-типа T*", тип идентификатора D есть "*модификатор-типа* массив из T". Если константное выражение присутствует, то оно должно быть целочисленным и больше 0. Если константное выражение, специфицирующее количество элементов в массиве, отсутствует, то массив имеет незавершенный тип.

Массив можно конструировать из объектов арифметического типа, указателей, структур и объединений, а также других массивов (генерируя при этом многомерные массивы). Любой тип, из которого конструируется массив, должен быть завершенным, он не может быть, например, структурой или массивом незавершенного типа. Это значит, что для многомерного массива пустой может быть только первая размерность. Незавершенный тип массива получает свое завершение либо в другом объявлении этого массива (A10.2), либо при его инициализации (A8.7). Например, запись

```
float fa[17], *afp[17];
```

объявляет массив из чисел типа *float* и массив из указателей на числа типа *float*. Аналогично

```
static int x3d[3][5][7];
```

объявляет статический трехмерный массив целых размера 3 x 5 x 7. На самом деле, если быть точными, *x3d* является массивом из трех элементов, каждый из которых есть массив из пяти элементов, содержащих по 7 значений типа *int*.

Операция индексирования E1[E2] определена так, что она идентична операции *(E1+E2). Следовательно, несмотря на асимметричность записи, индексирование - коммутативная операция. Учитывая правила преобразования, применяемые для оператора + и массивов (A6.6, A7.1, A7.7), можно сказать, что если E1 - массив, а E2 - целое, то E1[E2] обозначает E2-й элемент массива E1.

Так, x3d[i][j][k] означает то же самое, что и *(x3d[i][j]+k). Первое подвыражение, x3d[i][j], согласно A7.1, приводится к типу "указатель на массив целых"; по A7.7 сложение включает умножение на размер объекта типа *int*. Из этих же правил следует, что массивы запоминаются "построчно" (последние индексы меняются чаще) и что первая размерность в объявлении помогает определить количество памяти, занимаемой массивом, однако в вычислении адреса элемента массива участия не принимает.

A8.6.3. Объявители функций

В новом способе объявление функции T D, где D имеет вид

D1(*список-типов-параметров*)

и тип идентификатора объявления T D1 есть "*модификатор-типа* T", тип идентификатора в D есть "*модификатор-типа* функция с аргументами *список-типов- параметров*, возвращающая T". Параметры имеют следующий синтаксис:

```
список-типов-параметров:
    список-параметров
    список-параметров , ...

список-параметров:
    объявление-параметра
    список-параметров, объявление-параметра

объявление-параметра:
    спецификаторы-объявления объявитель
    спецификатор-объявления абстрактный-объявитель(необ)
```

При новом способе объявления функций список параметров специфицирует их типы, а если функция вообще не имеет параметров, на месте списка типов указывается одно слово - *void*. Если список типов параметров заканчивается многоточием ",...", то функция может иметь больше аргументов, чем число явно описанных параметров. (См. A7.3.2.)

Типы параметров, являющихся массивами и функциями, заменяются на указатели в соответствии с правилами преобразования параметров (A10.1). Единственный спецификатор класса памяти, который разрешается использовать в объявлении параметра, - это *register*, однако он игнорируется, если объявитель функции не является заголовком ее определения. Аналогично, если объявители в объявлениях параметров содержат идентификаторы, а объявитель функции не является заголовком определения функции, то эти идентификаторы тотчас же выводятся из текущей области видимости.

При старом способе объявление функции T D, где D имеет вид

D1(*список-идентификаторов*$_{необ}$)

и тип идентификатора объявления T D1 есть "*модификатор-типа* T", тип идентификатора в D есть "*модификатор-типа* функция от неспецифицированных аргументов, возвращающая T". Параметры, если они есть, имеют следующий вид:

список-идентификаторов:
    идентификатор
    список-идентификаторов, идентификатор

При старом способе, если объявитель функции не используется в качестве заголовка определения функции (A10.1), список идентификаторов должен отсутствовать. Никакой информации о типах параметров в объявлениях не содержится.

Например, объявление

```
int f(), *fpi(), (*pfi)();
```

объявляет функцию *f*, возвращающую число типа *int*, функцию *fpi*, возвращающую указатель на число типа *int*, и указатель *pfi* на функцию, возвращающую число типа *int*. Ни для одной функции в объявлении не указаны типы параметров; все функции описаны старым способом.

Вот как выглядит объявление в новой записи:

```
int strcpy(char *dest, const char *source), rand(void);
```

Здесь *strcpy* - функция с двумя аргументами, возвращающая значение типа *int*; первый аргумент - указатель на значение типа *char*, а второй - указатель на неизменяющееся значение типа *char*. Имена параметров играют роль хороших комментариев. Вторая функция, *rand*, аргументов не имеет и возвращает *int*.

> Объявители функций с прототипами параметров - наиболее важное нововведение ANSI-стандарта. В сравнении со старым способом, принятым в первой редакции языка, они позволяют проверять и приводить к нужному типу аргументы во всех вызовах. Следует однако отметить, что их введение привнесло в язык некоторую сумятицу и необходимость согласования обеих форм. Чтобы обеспечить совместимость, потребовались некоторые "синтаксические уродства", а именно *void*, для явного указания на отсутствие параметров.
> Многоточие ", ..." применительно к функциям с варьируемым числом аргументов - также новинка, которая вместе со стандартным заголовочным файлом макросов <**stdarg.h**> формализует неофициально используемый, но официально запрещенный в первой редакции механизм. Указанные способы записи заимствованы из языка C++.

A8.7. Инициализация

С помощью иниц-объявителя можно указать начальное значение объявляемого объекта. Инициализатору, представляющему собой выражение или список инициализаторов, заключенный в фигурные скобки, предшествует знак =. Этот список может завершаться запятой; ее назначение сделать форматирование более четким.

инициализатор:
    выражение-присваивания
    { *список-инициализаторов* }
    { *список-инициализаторов*, }

список-инициализаторов:
    инициализатор

```
список-инициализаторов , инициализатор
```

В инициализаторе статического объекта или массива все выражения должны быть константными (A7.19). Если инициализатор *auto*- и *register*-объекта или массива находится в списке, заключенном в фигурных скобки, то входящие в него выражения также должны быть константными. Однако в случае автоматического объекта с одним выражением инициализатор не обязан быть константным выражением, он просто должки иметь соответствующий объекту тип.

> В первой редакции не разрешалась инициализация автоматических структур, объединений и массивов. ANSI-стандарт позволяет это; однако, если инициализатор не может быть представлен одним простым выражением, инициализация может быть выполнена только с помощью константных конструкций.

Статический объект, инициализация которого явно не указана, инициализируется так, как если бы ему (или его элементам) присваивалась константа 0. Начальное значение автоматического объекта, явным образом не инициализированного, не определено.

Инициализатор указателя или объекта арифметического типа - это одно выражение (возможно, заключенное в фигурные скобки), которое присваивается объекту.

Инициализатор структуры - это либо выражение того же структурного типа, либо заключенные в фигурные скобки инициализаторы ее элементов, заданные по порядку. Безымянные битовые поля игнорируются и не инициализируются. Если инициализаторов в списке меньше, чем элементов, то оставшиеся элементы инициализируются нулем. Инициализаторов не должно быть больше числа элементов.

Инициализатор массива - это список инициализаторов его элементов, заключенный в фигурные скобки. Если размер массива не известен, то он считается равным числу инициализаторов, при этом тип его становится завершенным. Если размер массива известен, то число инициализаторов не должно превышать числа его элементов; если инициализаторов меньше, оставшиеся элементы обнуляются.

Как особый выделен случай инициализации массива символов. Последний можно инициализировать с помощью строкового литерала; символы инициализируют элементы массива в том порядке, как они заданы в строковом литерале. Точно так же, с помощью литерала из расширенного набора символов (A2.6), можно инициализировать массив типа *wchar_t*. Если размер массива не известен, то он определяется числом символов в строке, включая и завершающий NULL-символ; если размер массива известен, то число символов в строке, не считая завершающего NULL-символа, не должно превышать его размера.

Инициализатором объединения может быть либо выражение того же типа, либо заключенный в фигурные скобки инициализатор его первого элемента.

> В первой версии языка не позволялось инициализировать объединения. Правило "первого элемента" не отличается изяществом, однако не требует нового синтаксиса. Стандарт ANSI проясняет еще и семантику не инициализируемых явно объединений.

Введем для структуры и массива обобщенное имя: *агрегат*. Если агрегат содержит элементы агрегатного типа, то правила инициализации применяются рекурсивно. Фигурные скобки в некоторых случаях инициализации можно опускать. Если инициализатор элемента агрегата, который сам является агрегатом, начинается с левой фигурной скобки, то этот подагрегат инициализируется последующим списком разделенных запятыми инициализаторов; считается ошибкой, если количество инициализаторов подагрегата превышает число его элементов. Если, однако, инициализатор подагрегата не начинается с левой фигурной скобки, то чтобы его инициализировать, нужно отсчитать соответствующее число элементов из списка; при этом остальные элементы инициализируются следующими инициализаторами агрегата, для которого данный подагрегат является частью.

Например

```
int x[] = { 1, 3, 5 };
```

объявляет и инициализирует *x* как одномерный массив с тремя элементами, поскольку размер не был

указан, а список состоит из трех инициализаторов.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

представляет собой инициализацию с полным набором фигурных скобок: 1, 3 и 5 инициализируют первую строку в массиве y[0], т. е. y[0][0], y[0][1] и y[0][2]. Аналогично инициализируются следующие две строки: y[1] и y[2]. Инициализаторов не хватило на весь массив, поэтому элементы строки y[3] будут нулевыми. В точности тот же результат был бы достигнут с помощью следующего объявления:

```
float y[4][3] = {
   1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для *y* начинается с левой фигурной скобки, но для y[0] скобки нет, поэтому из списка будут взяты три элемента. Аналогично по три элемента будут взяты для y[1], а затем и для y[2]. В

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

инициализируется первый столбец матрицы *y*, все же другие элементы остаются нулевыми.

Наконец,

```
char msg[] = "Синтаксическая ошибка в строке %s\n";
```

представляет собой пример массива символов, элементы которого инициализируются с помощью строки; в его размере учитывается и завершающий NULL-символ.

A8.8. Имена типов

В ряде случаев возникает потребность в применении имени типа данных (например при явном приведении к типу, в указании типов параметров внутри объявлений функций, в аргументе оператора *sizeof*). Эта потребность реализуется с помощью *имени типа*, определение которого синтаксически почти совпадает с объявлением объекта того же типа. Оно отличается от объявления лишь тем, что не содержит имени объекта.

имя-типа:
    список-спецификаторов-квалификаторов абстрактный-объявитель$_{необ}$

абстрактный-объявитель:
    указатель
    указатель$_{необ}$ собственно-абстрактный-объявитель

собственно-абстрактный-объявитель:
    (*абстрактный-объявитель*)
    собственно-абстрактный-объявитель$_{необ}$[*константное-выражение*$_{необ}$]
    собственно-абстрактный-объявитель$_{необ}$(*список-типов-параметров*$_{необ}$)

Можно указать одно-единственное место в абстрактном объявителе, где мог бы оказаться идентификатор, если бы данная конструкция была полноценным объявителем. Именованный тип совпадает с типом этого "невидимого идентификатора". Например

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

соответственно обозначают типы *int*, "указатель на *int*", "массив из трех указателей на *int*", "указатель на массив из неизвестного количества *int*", "функция неизвестного количества параметров, возвращающая указатель на *int*", "массив неизвестного количества указателей на функции без параметров, каждая из которых возвращает *int*".

A8.9. Объявление *typedef*

Объявления, в которых спецификатор класса памяти есть *typedef*, не объявляют объектов - они определяют идентификаторы, представляющие собой имена типов. Эти идентификаторы называются *typedef*-именами.

```
typedef-имя:
    идентификатор
```

Объявление *typedef* приписывает тип каждому имени своего объявителя обычным способом (см. A8.6.). С этого момента *typedef*-имя синтаксически эквивалентно ключевому слову спецификатора типа, обозначающему связанный с ним тип. Например, после

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

допустимы следующие объявления:

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

b принадлежит типу *long*, *bp* — типу "указатель на *long*"; *z* — это структура заданного вида, а *zp* - принадлежит типу "указатель на такую структуру".

Объявление *typedef* не вводит новых типов, оно только дает имена типам, которые могли бы быть специфицированы и другим способом. Например, *b* имеет тот же тип, что и любой другой объект типа *long*.

typedef-имена могут быть перекрыты другими определениями во внутренней области видимости, но при условии, что в них присутствует указание типа. Например

```
extern Blockno;
```

не переобъявляет Blockno, а вот

```
extern int Blockno;
```

переобъявляет.

A8.10. Эквивалентность типов

Два списка спецификаторов типа эквивалентны, если они содержат одинаковый набор спецификаторов типа с учетом синонимичности названий (например, *long* и *int long* считаются одинаковыми типами). Структуры, объединения и перечисления с разными тегами считаются разными, а каждое безтеговое объединение, структура или перечисление представляет собой уникальный тип.

Два типа считаются совпадающими, если их абстрактные объявители (A8.8) после замены всех *typedef*-имен их типами и выбрасывания имен параметров функций составят эквивалентные списки спецификаторов типов. При сравнении учитываются размеры массивов и типы параметров функция.

A9. Инструкции

За исключением оговоренных случаев инструкции выполняются том порядке, как они написаны. Инструкции не имеют значений и выполняются, чтобы произвести определенные действия. Все виды инструкций можно разбить на несколько групп:

```
инструкция:
    помеченная–инструкция
    инструкция–выражение
    составная-инструкция
    инструкция-выбора
    циклическая-инструкция
    инструкция-перехода
```

A9.1. Помеченные инструкции

Инструкции может предшествовать метка.

```
помеченная-инструкция:
    идентификатор : инструкция
    case константное-выражение : инструкция
    default : инструкция
```

Метка, состоящая из идентификатора, одновременно служит и объявлением этого идентификатора. Единственное назначение идентификатора-метки - указать место перехода для *goto*. Областью видимости идентификатора-метки является текущая функция. Так как метки имеют свое собственное пространство имен, они не "конфликтуют" с другими идентификаторами и не могут быть перекрыты (см. A11.1.).

case-метки и *default*-метки используются в инструкции *switch* (A9.4). Константное выражение в *case* должно быть целочисленным.

Сами по себе метки не изменяют порядка вычислений.

A9.2. Инструкция-выражение

Наиболее употребительный вид инструкции - это инструкция-выражение.

```
инструкция-выражение:
    выражение_необ ;
```

Чаще всего инструкция-выражение - это присваивание или вызов функции. Все действия, реализующие побочный эффект выражения, завершаются, прежде чем начинает выполняться следующая инструкция. Если выражение в инструкции опущено, то она называется пустой; пустая инструкция часто используется для обозначения пустого тела циклической инструкции или в качестве места для метки.

A9.3. Составная инструкция

Так как в местах, где по синтаксису полагается одна инструкция, иногда возникает необходимость выполнить несколько, предусматривается возможность задания составной инструкции (которую также называют блоком). Тело определения функции есть составная инструкция;

```
составная-инструкция:
    { список-объявлений список-инструкций_необ }

список-объявлений:
    объявление
    список-объявлений объявление

список-инструкций:
    инструкция
    список-инструкций инструкция
```

Если идентификатор из списка объявлений находился в области видимости объемлющего блока, то действие внешнего объявления при входе внутрь данного блока приостанавливается (A11.1), а после выхода из него возобновляется. Внутри блока идентификатор может быть объявлен только один раз. Для каждого отдельного пространства имен эти правила действуют независимо (A11); идентификаторы из разных пространств имен всегда различны.

Инициализация автоматических объектов осуществляется при каждом входе в блок и продолжается по мере продвижения по объявителям. При передаче управления внутрь блока никакие инициализации не выполняются. Инициализации статических объектов осуществляются только один раз перед запуском программы.

A9.4. Инструкции выбора

Инструкции выбора осуществляют отбор одной из нескольких альтернатив, определяющих порядок выполнения инструкций.

```
инструкция-выбора:
    if ( выражение ) инструкция
    if ( выражение ) инструкция else инструкция
    switch ( выражение ) инструкция
```

Оба вида *if*-инструкций содержат выражение, которое должно иметь арифметический тип или тип указателя. Сначала вычисляется выражение со всеми его побочными эффектами, результат сравнивается с 0. В случае несовпадения с 0 выполняется первая подинструкция. В случае совпадения с 0 для второго типа *if* выполняется вторая подинструкция. Связанная со словом *else* неоднозначность разрешается тем, что слово *else* соотносят с последней еще не имеющей *else if*- инструкцией, расположенной в одном с этим *else* блоке и на одном уровне вложенности блоков.

Инструкция *switch* вызывает передачу управления на одну из нескольких инструкций в зависимости от значения выражения, которое должен иметь целочисленный тип.

Управляемая с помощью *switch* подинструкция обычно составная. Любая инструкция внутри этой подинструкции может быть помечена одной или несколькими *case*- метками (A9.1). Управляющее выражение подвергается целочисленному повышению (A6.1), а *case*-константы приводятся к повышенному типу. После такого преобразования никакие две *case*-константы в одной инструкции *switch* не должны иметь одинаковых значений. Со *switch*-инструкцией может быть связано не более одной *default*-метки. Конструкции *switch* допускается вкладывать друг в друга; *case* и *default*-метки относятся к самой внутренней *switch*-инструкции из тех, которые их содержат.

Инструкция *switch* выполняется следующим образом. Вычисляется выражение со всеми побочными эффектами, и результат сравнивается с каждой *case*-константой. Если одна из *case*-констант равна значению выражения, управление переходит на инструкцию с соответствующей *case*-меткой. Если ни с одной из *case*-констант нет совпадения, управление передается на инструкцию с *default*-меткой, если такая имеется, в противном случае ни одна из подинструкций *switch* не выполняется.

В первой версии языка требовалось, чтобы выражение и *case*-константы в *switch* были типа *int*.

A9.5. Циклические инструкции

Циклические инструкции специфицируют циклы.

циклическая-инструкция:
    while (*выражение*) *инструкция*
    do *инструкция* while (*выражение*)
    for (*выражение*$_{необ}$; *выражение*$_{необ}$; *выражение*$_{необ}$) *инструкция*

В инструкциях *while* и *do* выполнение подинструкций повторяется до тех пор, пока значение выражения не станет нулем. Выражение должно иметь арифметический тип или тип указателя. В *while* вычисление выражения со всеми побочными эффектами и проверка осуществляются перед каждым выполнением инструкции, а в *do* — после.

В инструкции *for* первое выражение вычисляется один раз, тем самым осуществляется инициализация цикла. На тип этого выражения никакие ограничения не накладываются. Второе выражение должно иметь арифметический тип или тип указателя; оно вычисляется перед каждой итерацией. Как только его значение становится равным 0, *for* прекращает свою работу. Третье выражение вычисляется после каждой итерации и, следовательно, выполняет повторную инициализацию цикла. Никаких ограничений на его тип нет. Побочные эффекты всех трех выражений заканчиваются по завершении их вычислений. Если подинструкция не содержит в себе *continue*) то

for (*выражение*$_1$; *выражение*$_2$; *выражение*$_3$) *инструкция*

эквивалентно конструкции

выражение$_1$;
    while (*выражение*$_2$) {
        инструкция
        выражение$_3$;
}

Любое из трех выражений цикла может быть опущено. Считается, что отсутствие второго выражения равносильно сравнению с нулем ненулевой константы.

A9.6. Инструкции перехода

Инструкции перехода осуществляют безусловную передачу управления.

```
инструкция-перехода:
    goto идентификатор;
    continue;
    break;
    return выражение_необ;
```

В *goto*-инструкции идентификатор должен быть меткой (A9.1), расположенной в текущей функции. Управление передается на помеченную инструкцию.

Инструкцию *continue* можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. Говоря более точно, для каждой из конструкций

```
while (...){
    ...
contin: ;
}

do {
    ...
contin: ;
} while (...);

for (...){
    ...
contin: ;
}
```

инструкция *continue*, если она не помещена в еще более внутренний цикл, делает то же самое, что и *goto contin*.

Инструкция *break* встречается в циклической или в *switch*-инструкции, и только в них. Она завершает работу самой внутренней циклической или *switch*- инструкции, содержащей данную инструкцию *break*, после чего управление переходит к следующей инструкции.

С помощью *return* функция возвращает управление в программу, откуда была вызвана. Если за *return* следует выражение, то его значение возвращается вызвавшей эту функцию программе. Значение выражения приводится к типу так, как если бы оно присваивалось переменной, имеющей тот же тип, что и функция.

Ситуация, когда "путь" вычислений приводит в конец функции (т. е. на последнюю закрывающую фигурную скобку), равносильна выполнению *return*- инструкции без выражения. При этом, а также в случае явного задания *return* без выражения возвращаемое значение не определено.

A10. Внешние объявления

То, что подготовлено в качестве ввода для Си-компилятора, называется единицей трансляции. Она состоит из последовательности внешних объявлений, каждое из которых представляет собой либо объявление, либо определение функции.

```
единица-трансляции:
    внешнее-объявление
    единица-трансляции внешнее-объявление

внешнее-объявление:
    определение-функции
    объявление
```

Область видимости внешних объявлений простирается до конца единицы трансляции, в которой они объявлены, точно так же, как область видимости объявлений в блоке распространяется до конца этого блока. Синтаксис внешнего объявления не отличается от синтаксиса любого другого объявления за одним исключением: код функции можно определять только с помощью внешнего объявления.

A10.1. Определение функции

Определение функции имеет следующий вид:

определение-функции:
    спецификаторы-объявления$_{необ}$ *объявитель список-объявлений*$_{необ}$
    составная-инструкция

Из спецификаторов класса памяти в спецификаторах-объявлениях возможны только *extern* и *static*; различия между последними рассматриваются в A11.2.

Типом возвращаемого функцией значения может быть арифметический тип, структура, объединение, указатель и *void*, но не "функция" и не "массив". Объявитель в объявлении функции должен явно указывать на то, что описываемый им идентификатор имеет тип "функция", т. е. он должен иметь одну из следующих двух форм (A8.6.3):

собственно-объявитель (*список-типов-параметров*)
собственно-объявитель (*список-идентификаторов*$_{необ}$)

где *собственно-объявитель* есть идентификатор или идентификатор, заключенный в скобки. Заметим, что тип "функция" посредством *typedef* получить нельзя.

Первая форма соответствует определению функции новым способом, для которого характерно объявление параметров в списке-типов-параметров вместе с их типами; за объявителем не должно быть списка-объявлений. Если список-типов-параметров не состоит из одного-единственного слова *void*, показывающего, что параметров у функции нет, то в каждом объявителе в списке-типов-параметров обязан присутствовать идентификатор. Если список-типов-параметров заканчивается знаками ", ...", то вызов функции может иметь аргументов больше, чем параметров; в таком случае, чтобы обращаться к дополнительным аргументам, следует пользоваться механизмом макроса *va_arg* из заголовочного файла <**stdarg.h**>, описанного в приложении B. Функции с переменным числом аргументов должны иметь по крайней мере один именованный параметр.

Вторая форма - определение функции старым способом. Список-идентификаторов содержит имена параметров, а список-объявлений приписывает им типы. В списке- объявлении разрешено объявлять только именованные параметры, инициализация запрещается, и из спецификаторов класса памяти возможен только *register*.

И в том и другом способе определения функции мыслится, что все параметры как бы объявлены в самом начале составной инструкции, образующей тело функции, и совпадающие с ними имена здесь объявляться не должны (хотя, как и любые идентификаторы, их можно переобъявить в более внутренних блоках). Объявление параметра "массив из *типа*" можно трактовать как "указатель на *тип*", аналогично объявлению параметра объявление "функция, возвращающая *тип*" можно трактовать как "указатель на функцию, возвращающую *тип*". В момент вызова функции ее аргументы соответствующим образом преобразуются и присваиваются параметрам (см. A7.3.2).

> Новый способ определения функций введен ANSI-стандартом. Есть также небольшие изменения в операции повышения типа; в первой версии языка параметры типа *float* следовало читать как *double*. Различие между *float* и *double* становилось заметным, лишь когда внутри функции генерировался указатель на параметр.

Ниже приведен пример определения функции новым способом:

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a: b;
    return (m > c) ? m : c;
}
```

Здесь *int*-спецификаторы-объявления; max(int a, int b, int c) - объявитель функции, а { ... } - блок, задающий ее код. Определение старым способом той же функции выглядит следующим образом:

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

где max(a, b, c) – объявитель, а int a, b, c - список-объявлений для параметров.

A10.2. Внешние объявления

Внешние объявления специфицируют характеристики объектов, функций и других идентификаторов. Термин "внешний" здесь используется, чтобы подчеркнуть тот факт, что объявления расположены вне функций; впрямую с ключевым словом *extern* ("внешний") он не связан. Класс памяти для объекта с внешним объявлением либо вообще не указывается, либо специфицируется как *extern* или *static*.

В одной единице трансляции для одного идентификатора может содержаться несколько внешних объявлений, если они согласуются друг с другом по типу и способу связи и если для этого идентификатора существует не более одного определения.

Два объявления объекта или функции считаются согласованными по типу в соответствии с правилами, рассмотренными в A8.10. Кроме того, если объявления отличаются лишь тем, что в одном из них тип структуры, объединения или перечисления незавершен (A8.3), а в другом соответствующий ему тип с тем же тегом завершен, то такие типы считаются согласованными. Если два типа массива (A8.6.2) отличаются лишь тем, что один завершенный, а другой незавершенный, то такие типы также считаются согласованными. Наконец, если один тип специфицирует функцию старым способом, а другой - ту же функцию новым способом (с объявлениями параметров), то такие типы также считаются согласованными.

Если первое внешнее объявление функции или объекта помечено спецификатором *static*, то объявленный идентификатор имеет *внутреннюю связь*; в противном случае - *внешнюю связь*. Способы связей обсуждаются в A11.2.

Внешнее объявление объекта считается определением, если оно имеет инициализатор. Внешнее объявление, в котором нет инициализатора и нет спецификатора *extern*, считается *пробным определением*. Если в единице трансляции появится определение объекта, то все его пробные определения просто станут избыточными объявлениями. Если никакого определения для этого объекта в единице трансляции не обнаружится, то все его пробные определения будут трактоваться как одно определение с инициализатором 0.

Каждый объект должен иметь ровно одно определение. Для объекта с внутренней связью это правило относится к каждой отдельной единице трансляции, поскольку объекты с внутренними связями в каждой единице уникальны. В случае объектов с внешними связями указанное правило действует в отношении всей программы в целом.

> Хотя правило одного определения формулируется несколько иначе, чем в первой версии языка, по существу оно совпадает с прежним. Некоторые реализации его ослабляют, более широко трактуя понятие пробного определения. В другом варианте указанного правила, который распространен в системах UNIX и признан как общепринятое расширение стандарта, все пробные определения объектов с внешними связями из всех транслируемых единиц программы рассматриваются вместе, а не отдельно в каждой единице. Если где-то в программе обнаруживается определение, то пробные определения становятся просто объявлениями, но, если никакого определения не встретилось, то все пробные определения становятся одним-единственным определением с инициализатором 0.

A11. Область видимости и связи

Каждый раз компилировать всю программу целиком нет необходимости. Исходный текст можно хранить в нескольких файлах, представляющих собой единицы трансляции. Ранее скомпилированные программы могут загружаться из библиотек. Связи между функциями программы могут осуществляться через вызовы и внешние данные.

Следовательно, существуют два вида областей видимости: первая - это *лексическая область* идентификатора: т. е. область в тексте программы, где имеют смысл все его характеристики; вторая область - это область, ассоциируемая с объектами и функциями, имеющими внешние связи, устанавливаемые между идентификаторами из раздельно компилируемых единиц трансляции.

A11.1. Лексическая область видимости

Каждый идентификатор попадает в одно из нескольких пространств имен. Эти пространства никак не связаны друг с другом. Один и тот же идентификатор может использоваться в разных смыслах даже в одной области видимости, если он принадлежит разным пространствам имен. Ниже через точку с запятой перечислены классы объектов, имена которых представляют собой отдельные независимые пространства: объекты, функции, *typedef*-имена и *enum*-константы; метки инструкций; теги структур, объединений и перечислений; элементы каждой отдельной структуры или объединения.

> Сформулированные правила несколько отличаются от прежних, описанных в первом издании. Метки инструкций не имели раньше собственного пространства; теги структур и теги объединений (а в некоторых реализациях и теги перечислений) имели отдельные пространства. Размещение тегов структур, объединений и перечислений в одном общем пространстве - это дополнительное ограничение, которого раньше не было. Наиболее существенное отклонение от первой редакции в том, что каждая отдельная структура (или объединение) создает свое собственное пространство имен для своих элементов. Таким образом, одно и то же имя может использоваться в нескольких различных структурах. Это правило широко применяется уже несколько лет.

Лексическая область видимости идентификатора объекта (или функции), объявленного во внешнем объявлении, начинается с места, где заканчивается его объявитель, и простирается до конца единицы трансляции, в которой он объявлен. Область видимости параметра в определении функции начинается с начала блока, представляющего собой тело функции, и распространяется на всю функцию; область видимости параметра в описании функции заканчивается в конце этого описания. Область видимости идентификатора, объявленного в начале блока, начинается от места, где заканчивается его объявитель, и продолжается до конца этого блока. Областью видимости метки является вся функция, где эта метка встречается. Область видимости тега структуры, объединения или перечисления начинается от его появления в спецификаторе типа и продолжается до конца единицы трансляции для объявления внешнего уровня и до конца блока для объявления внутри функции.

Если идентификатор явно объявлен в начале блока (в том числе тела функции), то любое объявление того же идентификатора, находящееся снаружи этого блока, временно перестает действовать вплоть до конца блока.

A11.2. Связи

Если встречается несколько объявлений, имеющих одинаковый идентификатор и описывающих объект (или функцию), то все эти объявления в случае внешней связи относятся к одному объекту (функции) - уникальному для всей программы; если же связь внутренняя, то свойство уникальности распространяется только на единицу трансляции.

Как говорилось в A10.2, если первое внешнее объявление имеет спецификатор *static*, то оно описывает идентификатор с внутренней связью, если такого спецификатора нет, то - с внешней связью. Если объявление находится внутри блока и не содержит *extern*, то соответствующий идентификатор ни с чем не связан и уникален для данной функции. Если объявление содержит *extern* и блок находится к области видимости внешнего объявления этого идентификатора, то последний имеет ту же связь и относится к тому же объекту (функции). Однако если ни одного внешнего объявления для этого идентификатора нет, то он имеет внешнюю связь.

A12. Препроцессирование

Препроцессор выполняет макроподстановку, условную компиляцию, включение именованных файлов. Строки, начинающиеся со знака # (перед которым возможны символы-разделители), устанавливают связь с препроцессором. Их синтаксис не зависит от остальной части языка; они могут появляться где угодно и оказывать влияние (независимо от области видимости) вплоть до конца транслируемой единицы. Границы строк принимаются во внимание: каждая строка анализируется отдельно (однако есть возможность "склеивать" строки, см. A12.2). Лексемами для препроцессора являются все лексемы

языка и последовательности символов, задающие имена файлов, как, например, в директиве #include (A12.4). Кроме того, любой символ, неопределенный каким-либо другим способом, воспринимается как лексема. Влияние символов-разделителей, отличающихся от пробелов и горизонтальных табуляций, внутри строк препроцессора не определено.

Само препроцессирование проистекает в нескольких логически последовательных фазах. В отдельных реализациях некоторые фазы объединены.

1. Трехзнаковые последовательности, описанные в A12.1, заменяются их эквивалентами. Между строками вставляются символы новой строки, если того требует операционная система.
2. Выбрасываются пары символов, состоящие из обратной наклонной черты с последующим символом новой строки; тем самым осуществляется "склеивание" строк (A12.2).
3. Программа разбивается на лексемы, разделенные символами-разделителями. Комментарии заменяются единичными пробелами. Затем выполняются директивы препроцессора и макроподстановки (A12.3-A12.10).
4. Эскейп-последовательности в символьных константах и строковых литералах (A2.5.2, A2.6) заменяются на символы, которые они обозначают. Соседние строковые литералы конкатенируются.
5. Результат транслируется. Затем устанавливаются связи с другими программами и библиотеками посредством сбора необходимых программ и данных и соединения ссылок на внешние функции и объекты с их определениями.

A12.1. Трехзнаковые последовательности

Множество символов, из которых набираются исходные Си-программы, основано на семибитовом ASCII-коде. Однако он шире, чем инвариантный код символов ISO 646-1983 (ISO 646-1983 Invariant Code Set). Чтобы дать возможность пользоваться сокращенным набором символов, все указанные ниже трехзнаковые последовательности заменяются на соответствующие им единичные символы. Замена осуществляется до любой иной обработки.

```
??=    #
??(    [
??<    {
??/    \
??)    ]
??>    }
??'    ^
??!    |
??-    ~
```

Никакие другие замены, кроме указанных, не делаются.

> Трехзнаковые последовательности введены ANSI-стандартом.

A12.2. Склеивание строк

Строка, заканчивающаяся обратной наклонной чертой, соединяется со следующей, поскольку символ \ и следующий за ним символ новой строки выбрасываются. Это делается перед "разбиением" текста на лексемы.

A12.3. Макроопределение и макрорасширение

Управляющая строка вида

```
#define идентификатор последовательность-лексем
```

заставляет препроцессор заменять *идентификатор* на *последовательность-лексем*; символы-разделители в начале и в конце последовательности лексем выбрасываются. Повторная строка *#define* с тем же идентификатором считается ошибкой, если последовательности лексем неидентичны (несовпадения в символах- разделителях при сравнении во внимание не принимаются). Строка вида

```
#define идентификатор(список-идентификаторов) последовательность-лексем
```

где между первым идентификатором и знаком (не должно быть ни одного символа- разделителя, представляет собой макроопределение с параметрами, задаваемыми списком идентификаторов. Как и в первом варианте, символы-разделители в начале и в конце последовательности лексем выбрасываются, и макрос может быть повторно определен только с тем же списком параметров и с той же последовательностью лексем. Управляющая строка вида

```
#undef идентификатор
```

предписывает препроцессору "забыть" определение, данное идентификатору. Применение *#undef* к неизвестному идентификатору ошибкой не считается.

Если макроопределение было задано вторым способом, то текстовая последовательность, состоящая из его идентификатора, возможно, со следующими за ним символами-разделителями, знака (, списка лексем, разделенных запятыми, и знака), представляет собой вызов макроса. Аргументами вызова макроса являются лексемы, разделенные запятыми (запятые, "закрытые" кавычками или вложенными скобками, в разделении аргументов не участвуют). Аргументы при их выделении макрорасширениям не подвергаются. Количество аргументов в вызове макроса должно соответствовать количеству параметров макроопределения. После выделения аргументов окружающие их символы-разделители выбрасываются. Затем в замещающей последовательности лексем макроса идентификаторы-параметры (если они не закавычены) заменяются на соответствующие им аргументы. Если в замещающей последовательности перед параметром не стоит знак # и ни перед ним, ни после него нет знака ##, то лексемы аргумента проверяются: не содержат ли они в себе макровызова, и если содержат, то прежде чем аргумент будет подставлен, производится соответствующее ему макрорасширение.

На процесс подстановки влияют два специальных оператора. Первый -это оператор #, который ставится перед параметром. Он требует, чтобы подставляемый вместо параметра и знака # (перед ним) текст был заключен в двойные кавычки. При этом в строковых литералах и символьных константах аргумента перед каждой двойной кавычкой " (включая и обрамляющие строки), а также перед каждой обратной наклонной чертой \ вставляется \.

Второй оператор записывается как ##. Если последовательность лексем в любого вида макроопределении содержит оператор ##, то сразу после подстановки параметров он вместе с окружающими его символами-разделителями выбрасывается, благодаря чему "склеиваются" соседние лексемы, образуя тем самым новую лексему. Результат не определен при получении неправильных лексем или когда генерируемый текст зависит от порядка применения операторов ##. Кроме того, ## не может стоять ни в начале, ни в конце замещающей последовательности лексем.

В макросах обоих видов замещающая последовательность лексем повторно просматривается на предмет обнаружения там новых *define*-имен. Однако, если некоторый идентификатор уже был заменен в данном расширении, повторное появление такого идентификатора не вызовет его замены.

Если полученное расширение начинается со знака #, оно не будет воспринято как директива препроцессора.

> В ANSI-стандарте процесс макрорасширения описан более точно, чем в первом издании книги. Наиболее важные изменения касаются введения операторов # и ##, которые предоставляют возможность осуществлять расширения внутри строк и конкатенацию лексем. Некоторые из новых правил, особенно касающиеся конкатенации, могут показаться несколько странными. (См. приведенные ниже примеры.)

Описанные возможности можно использовать для показа смысловой сущности констант, как, например, в

```
#define TABSIZE 100
int table[TABSIZE];
```

Определение

```
#define ABSDIFF(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

задает макрос, возвращающий абсолютное значение разности его аргументов. В отличие от функции,

делающей то же самое, аргументы и возвращаемое значение здесь могут иметь любой арифметический тип и даже быть указателями. Кроме того, аргументы, каждый из которых может иметь побочный эффект, вычисляются дважды: один раз - при проверке, другой раз - при вычислении результата.

Если имеется определение

```
#define tempfile(dir) #dir "/%s"
```

то макровызов *tempfile(/usr/tmp)* даст в результате

```
"/usr/tmp" "/%s"
```

Далее эти две строки превратятся в одну строку. По макросу

```
#define cat(x,y) x ## y
```

вызов *cat(var, 123)* сгенерирует *var123*. Однако *cat (cat (1,2),3)* не даст желаемого, так как оператор ## воспрепятствует получению правильных аргументов для внешнего вызова *cat*. В результате будет выдана следующая цепочка лексем:

```
cat ( 1 , 2 )3
```

где)3 (результат "склеивания" последней лексемы первого аргумента с первой лексемой второго аргумента) не является правильной лексемой. Если второй уровень макроопределения задан в виде

```
#define xcat(x,y) cat(x,y)
```

то никаких коллизий здесь не возникает;

```
xcat(xcat(1, 2), 3)
```

в итоге даст 123, поскольку сам *xcat* не использует оператора ##.

Аналогично сработает и ABSDIFF(ABSDIFF(a, b), c), и мы получим правильный результат.

A12.4. Включение файла

Управляющая строка

```
#include <имя-файла>
```

заменяется на содержимое файла с именем *имя-файла*. Среди символов, составляющих *имя-файла*, не должно быть знака > и символа новой строки. Результат не определен, если *имя-файла* содержит любой из символов ", ', \ или пару символов /*. Порядок поиска указанного файла зависит от реализации.

Подобным же образом выполняется управляющая строка

```
#include "имя-файла"
```

Сначала поиск осуществляется по тем же правилам, по каким компилятор ищет первоначальный исходный файл (механизм этого поиска зависит от реализации), а в случае неудачи осуществляется методом поиска, принятым в #include первого типа. Результат остается неопределенным, если имя файла содержит ", \ или /*; использование знака > разрешается.

Наконец, директива

```
#include последовательность-лексем
```

не совпадающая ни с одной из предыдущих форм, рассматривает *последовательность лексем* как текст, который в результате всех макроподстановок должен дать *#include* <...> или *#include "..."*. Сгенерированная таким образом директива далее будет интерпретироваться в соответствии с полученной формой.

Файлы, вставляемые с помощью *#include*, сами могут содержать в себе директивы *#include*.

A12.5. Условная компиляция

Части программы могут компилироваться условно, если они оформлены в соответствии со следующим схематично изображенным синтаксисом:

```
условная - конструкция -препроцессора:
    if-строка текст elif-части else-часть_необ #endif
if-строка:
    #if константное-выражение
    #ifdef идентификатор
    #ifndef идентификатор
elif-части:
    elif-строка текст
    elif-части_необ
elif-строка:
    #elif константное-выражение
else-часть:
    else-строка текст
else-строка:
    #else
```

Каждая из директив (*if*-строка, *elif*-строка, *else*-строка и *#endif*) записывается на отдельной строке. Константные выражения в *#if* и последующих строках *#elif* вычисляются по порядку, пока не обнаружится выражение с ненулевым (истинным) значением; текст, следующий за строкой с нулевым значением, выбрасывается. Текст, расположенный за директивой с ненулевым значением, обрабатывается обычным образом. Под словом "*текст*" здесь имеется в виду любая последовательность строк, включая строки препроцессора, которые не являются частью условной структуры; текст может быть и пустым. Если строка *#if* или *#elif* с ненулевым значением выражения найдена и ее текст обработан, то последующие строки *#elif* и *#else* вместе со своими текстами выбрасываются. Если все выражения имеют нулевые значения и присутствует строка *#else*, то следующий за ней текст обрабатывается обычным образом. Тексты "неактивных" ветвей условных конструкций, за исключением тех, которые заведуют вложенностью условных конструкций, игнорируются.

Константные выражения в *#if* и *#elif* являются объектами для обычной макроподстановки. Более того, прежде чем просматривать выражения вида

```
defined идентификатор
```

и

```
defined ( идентификатор )
```

на предмет наличия в них макровызова, они заменяются на 1L или 0L в зависимости от того, был или не был определен препроцессором указанный в них идентификатор. Все идентификаторы, оставшиеся после макрорасширения, заменяются на 0L. Наконец, предполагается, что любая целая константа всегда имеет суффикс L, т. е. вся арифметика имеет дело с операндами только типа *long* или *unsigned long*.

Константное выражение (A7.19) здесь используется с ограничениями: оно должно быть целочисленным, не может содержать в себе перечислимых констант, преобразований типа и операторов *sizeof*.

Управляющие строки

```
#ifdef идентификатор
#ifndef идентификатор
```

эквивалентны соответственно строкам

```
#if defined идентификатор
#if !defined идентификатор
```

> Строки *#elif* не было в первой версии языка, хотя она и использовалась в некоторых препроцессорах. Оператор препроцессора *defined* - также новый.

A12.6. Нумерация строк

Для удобства работы с другими препроцессорами, генерирующими Си-программы, можно использовать одну из следующих директив:

```
#line константа "имя-файла"
```

```
#line константа
```

Эти директивы предписывают компилятору считать, что указанные десятичное целое и идентификатор являются номером следующей строки и именем текущего файла соответственно. Если имя файла отсутствует, то ранее запомненное имя не изменяется. Расширения макровызовов в директиве *#line* выполняются до интерпретации последней.

A12.7. Генерация сообщения об ошибке

Строка препроцессора вида

```
#error последовательность-лексем_необ
```

приказывает ему выдать диагностическое сообщение, включающее заданную последовательность лексем.

A12.8. Прагма

Управляющая строка вида

```
#pragma последовательность-лексем_необ
```

призывает препроцессор выполнить зависящие от реализации действия. Неопознанная прагма игнорируется.

A12.9. Пустая директива

Строка препроцессора вида

```
#
```

не вызывает никаких действий.

A12.10. Заранее определенные имена

Препроцессор "понимает" несколько заранее определенных идентификаторов; их он заменяет специальной информацией. Эти идентификаторы (и оператор препроцессора *defined* в том числе) нельзя повторно переопределять, к ним нельзя также применять директиву *#undef*. Это следующие идентификаторы:

```
__LINE__ Номер текущей строки исходного текста, десятичная константа.
__FILE__ Имя компилируемого файла, строка.
__DATE__ Дата компиляции в виде "MMM DD YYYY",строка.
__TIME__ Время компиляции в виде "hh:mm:ss", строка.
__STDC__ Константа 1. Предполагается, что этот идентификатор определен как 1
         только в тех реализациях, которые следуют стандарту.
```

> Строки *#error* и *#pragma* впервые введены ANSI-стандартом. Заранее определенные макросы препроцессора также до сих пор не описывались, хотя и использовались в некоторых реализациях.

A13. Грамматика

Ниже приведены грамматические правила, которые мы уже рассматривали в данном приложении. Они имеют то же содержание, но даны в ином порядке.

Здесь не приводятся определения следующих символов-терминов: *целая-константа*, *символьная-константа*, *константа-с-плавающей-точкой*, *идентификатор*, *строка* и *константа-перечисление*. Слова, набранные обычным латинским шрифтом (не курсивом), и знаки рассматриваются как символы-термины и используются точно в том виде, как записаны. Данную грамматику можно механически трансформировать в текст, понятный системе автоматической генерации грамматического распознавателя. Для этого помимо добавления некоторых синтаксических пометок, предназначенных для указания альтернативных продукций, потребуется расшифровка конструкции со

словами "один из" и дублирование каждой продукции, использующей символ с индексом *необ.*, причем один вариант продукции должен быть написан с этим символом, а другой - без него. С одним изменением, а именно - удалением продукции *typedef-имя*:*идентификатор* и объявлением *typedef-имени* символом-термином, данная грамматика будет понятна генератору грамматического распознавателя YACC. Ей присуще лишь одно противоречие, вызываемое неоднозначностью конструкции *if-else*.

единица–трансляции:
    внешнее-объявление
    единица-трансляции внешнее-объявление

внешнее-объявление:
    определение-функции
    объявление

определение функции:
    спецификаторы-объявления$_{необ}$ *объявитель*
    список-объявлений$_{необ}$ *составная-инструкция*

объявление:
    спецификаторы-объявления список-инициализаторов-объявителей$_{необ}$

список-объявлений:
    объявление
    список-объявлений объявление

спецификаторы-объявления:
    спецификатор-класса-памяти спецификаторы-объявления$_{необ}$
    спецификатор-типа спецификаторы-объявления$_{необ}$
    квалификатор-типа спецификаторы-объявления$_{необ}$

спецификатор-класса-памяти: один из
    auto   register   static   extern   typedef

спецификатор-типа: один из
    void char short int long float double signed unsigned
    спецификатор-структуры-или-объединения
    спецификатор-перечисления
    typedef-имя

квалификатор-типа: один из
    const   volatile

спецификатор-структуры-или-объединения:
    структуры-или-объединения-идентификатор$_{необ}$ { *список-объявлений-структуры* }
    структуры-или-объединения идентификатор

структура-или-объединение: одно из
    struct   union

список-объявлений-структуры:
    объявление-структуры
    список-объявлений-структуры объявление-структуры

список-объявителей-иициализаторов:
    объявитель-инициализатор
    список-объявителей-инициализаторов , объявитель-инициализатор

объявитель-инициализатор:
    объявитель
    объявитель = инициализатор

объявление-структуры:
    список-спецификаторов-квалификаторов список-объявителей-структуры

список-спецификаторов-квалификаторов:
    спецификатор-типа список-спецификаторов-квалификаторов$_{необ}$
    квалификатор-типа список-спецификаторов-квалификаторов$_{необ}$

список-структуры-объявителей:
    структуры–объявитель
    список-структуры-объявителей , структуры-объявитель

структуры-объявитель:
    объявитель
    объявитель$_{необ}$: *константное-выражение*

спецификатор-перечисления:
    enum *идентификатор*$_{необ}$ { *список-перечислителей* }
    enum *идентификатор*

список-перечислителей:

```
    перечислитель
    список-перечислителей перечислитель

перечислитель:
    идентификатор
    указатель_необ собственно-объявитель

собственно-объявитель:
    идентификатор
    ( объявитель )
    собственно-объявитель [ константное-выражение_необ ]
    собственно-объявитель ( список-типов-параметров )
    собственно-объявитель ( список-идентификаторов_необ )

указатель:
    * список~квалификаторов-типа_необ
    * список-квалификаторов-типа_необ указатель

список-квалификаторов-типа:
    квалификатор-типа
    список-квалификаторов-типа квалификатор-типа

список-типов-параметров:
    список-параметров
    список-параметров , ...

список-параметров:
    объявление-параметра
    список-параметров , объявление-параметра

объявление-параметра:
    спецификаторы-объявления объявитель
    спецификаторы-объявления абстрактный-объявитель_необ

список-идентификаторов:
    идентификатор
    список-идентификаторов , идентификатор

инициализатор:
    выражение-присваивания
    { список-инициализаторов }
    { список-инициализаторов, }

список-инициализаторов:
    инициализатор
    список-инициализаторов , инициализатор

имя-типа:
    список-спецификаторое-квалификаторов абстрактный-объявитель_необ

абстрактный-объявитель:
    указатель
    указатель_необ собственно-абстрактный-объявитель

собственно-абстрактный-объявитель:
    ( абстрактный-объявитель )
    собственно-абстрактный-объявитель_необ [константное-выражение_необ]
    собственно-абстрактный-объявитель_необ (список-типов-параметров_необ)

typedef-имя:
    идентификатор

инструкция:
    помеченная-инструкция
    инструкция—выражение
    составная-инструкция
    инструкция-выбора
    циклическая-инструкция
    инструкция-перехода

помеченная-инструкция:
    идентификатор : инструкция
    case константное-выражение : инструкция
    default : инструкция

инструкция-выражение:
    выражение_необ;

составная-инструкция:
    ( список-объявлений_необ список-инструкций_необ)

список-инструкций:
    инструкция
    список-инструкций инструкция

инструкция-выбора:
```

```
    if ( выражение ) инструкция
    if ( выражение ) инструкция else инструкция
    switch ( выражение ) инструкция

циклическая-инструкция:
    while ( выражение ) инструкция
    do инструкция while ( выражение )
    return выражение_необ;

выражение:
    выражение-присваивания
    выражение , выражение-присваивания

выражение-присваивания:
    условное-выражение
    унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из
    =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=

условное-выражение:
    логическое-ИЛИ-выражение
    логическое-ИЛИ-выражение ? выражение : условное-выражение

константное-выражение:
    условное-выражение

логическое-ИЛИ-выражение:
    логическое-И-выражение
    логическое-ИЛИ-выражение || логическое-И-выражение

логическое-И-выражение:
    ИЛИ-выражение
    логическое-И-выражение && ИЛИ-выражение

ИЛИ-выражение:
    исключающее-ИЛИ-выражение
    ИЛИ-выражение | исключающее-ИЛИ-выражение

исключающее-ИЛИ-выражение:
    И-выражение
    исключающее-ИЛИ-выражение ^ И-выражение

И-выражение:
    выражение-равенства
    И-выражение & выражение-равенства

выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения

выражение-отношения:
    сдвиговое-выражение
    выражение-отношения < сдвиговое-выражение
    выражение-отношения > сдвиговое-выражение
    выражение-отношения <= сдвиговое-выражение
    выражение-отношения >= сдвиговое-выражение

сдвиговое-выражение:
    аддитивное-выражение
    сдвиговое-выражение >> аддитивное-выражение
    сдвиговое-выражение << аддитивное-выражение

аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение - мультипликативное-выражение

мультипликативное-выражение:
    выражение-приведенное-к-типу
    мультипликативное-выражение * выражение-приведенное-к-типу
    мультипликативное-выражение / выражение-приведенное-к-типу
    мультипликативное-выражение % выражение-приведенное-к-типу

выражение-приведенное-к-типу:
    унарное-выражение
    ( имя-типа ) выражение-приведенное-к-типу

унарное-выражение:
    постфиксное –выражение
    ++ унарное-выражение
    -- унарное-выражение
    унарный-оператор выражение-приведенное-к-типу
    sizeof унарное-выражение
    sizeof( имя-типа )
```

```
унарный-оператор: один из
    &   *   +   -   ~   !

постфиксное-выражение:
    первичное-выражение
    постфиксное-выражение [ выражение ]
    постфиксное-выражение ( список-аргументов-выражений_необ )
    постфиксное-выражение , идентификатор
    постфиксное-выражение -> идентификатор
    постфиксное-выражение ++
    постфиксное-выражение —

первичное -выражение:
    идентификатор
    константа
    строка
    ( выражение )

список-аргументов-выражений:
    выражение-присваивания
    список-аргументов-выражений , выражение-присваивания

константа:
    целая-константа
    символьная-константа
    константа-с-плавающей-точкой
    константа-перечисление
```

Ниже приводится грамматика языка препроцессора в виде перечня структур управляющих строк. Для механического получения программы грамматического разбора она не годится. Грамматика включает символ *текст*, который означает текст обычной программы, безусловные управляющие строки препроцессора и его законченные условные конструкции.

```
управляющая-строка:
    #define идентификатор последовательность-лексем
    #define идентификатор ( идентификатор, ..., идентификатор) последовательность-лексем
    #undef идентификатор
    #include <имя-файла>
    #include "имя-файла"
    #include последовательность-лексем
    #line константа "идентификатор"
    #line константа
    #error последовательность-лексем_необ
    #pragma последовательность-лексем_необ
    #
    условная-конструкция-препроцессора

условная-конструкция-препроцессора:
    if-строка текст elif-части else-часть_необ #endif

if-строка:
    #if константное-выражение
    #ifdef идентификатор
    #ifndef идентификатор

elif-части:
    elif-строка текст
    elif-части_необ

elif-строка:
    #elif константное-выражение

else-часть:
    else-строка текст

else-строка:
    #else
```

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]

TOP mail ТОП Разработка игр