1. Введение

Язык программирования C++ - это C[*], расширенный введением классов, inline-функций, перегруженных операций, перегруженных имен функций, константных типов, ссылок, операций управления свободной памятью, проверки параметров функций. Коротко различия между C++ и "старым C" приведены в #15. В этом руководстве описывается язык по состоянию на Июнь 1985.

2. Договоренности о лексике

Есть шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операторы и прочие разделители. Символы пробела, табуляции и новой строки, а также комментарии (собирательно - "белые места"), как описано ниже, игнорируются, за исключением тех случаев, когда они служат разделителями лексем. Некое пустое место необходимо для разделения идентификаторов, ключевых слов и констант, которые в противном случае окажутся соприкасающимися.

Если входной поток разобран на лексемы до данного символа, принимается, что следующая лексема содержит наиболее длинную строку символов из тех, что могут составить лексему.

2.1 Комментарии

Символы /* задают начало комментария, заканчивающегося символами */. Комментарии не могут быть вложенными. Символы // начинают комментарий, который заканчивается в конце строки, на которой они появились.

2.2 Идентификаторы (имена)

Идентификатор - последовательность букв и цифр произвольной длины; первый символ обязан быть буквой; подчерк '_' считается за букву; буквы в верхнем и нижнем регистрах являются различными.

2.3 Ключевые слова

Следующие идентификаторы зарезервированы для использования в качестве ключевых слов и не могут использоваться иным образом:

```
asm       auto      break     case      char
class     const     continue  default   delete
do        double    else      enum      extern
float     for       friend    goto      if
inline    int       long      new       operator
overload  public    register  return    short
sizeof    static    struct    switch    this
typedef   union     unsigned  virtual   void
while
```

Идентификаторы signed и volatile зарезервированы для применения в будущем.

2.4 Константы

Как описано ниже, есть несколько видов констант. В #2.6 приводится краткая сводка аппаратных характеристик, которые влияют на их размеры.

2.4.1 Целые константы

Целая константа, состоящая из последовательности цифр, считается восьмеричной, если она начинается с 0 (цифры ноль), и десятичной в противном случае. Цифры 8 и 9 не являются восьмеричными цифрами. Последовательность цифр, которой предшествует 0x или 0X, воспринимается как шестнадцатеричное целое. В шестнадцатеричные цифры входят буквы от a или A до f или F, имеющие значения от 10 до 15. Десятичная константа, значение которой превышает наибольшее машинное целое со знаком, считается длинной (long); восьмеричная и шестнадцатеричная константа, значение которой превышает наибольшее машинное целое со знаком, считается long; в остальных случаях целые константы считаются int.

2.4.2 Явно заданные длинные константы

Десятичная, восьмеричная или шестнадцатеричная константа, за которой непосредственно стоит l (латинская буква "эль") или L, считается длинной константой.

2.4.3 Символьные константы

Символьная константа состоит из символа, заключенного в одиночные кавычки (апострофы), как, например, 'x'. Значением символьной константы является численное значение символа в машинном наборе символов (алфавите). Символьные константы считаются данными типа int.

Некоторые неграфические символы, одиночная кавычка ' и обратная косая \, могут быть представлены в соответствие со следующей таблицей escape-последовательностей:

символ новой строки	NL(LF)	\n
горизонтальная табуляция	NT	\t
вертикальная табуляция	VT	\v
возврат на шаг	BS	\b
возврат каретки	CR	\r
перевод формата	FF	\f
обратная косая	\	\\
одиночная кавычка (апостроф)	'	\'
набор битов	0ddd	\ddd
набор битов	0xddd	\xddd

Escape-последовательность \ddd состоит из обратной косой, за которой следуют 1, 2 или 3 восьмеричных цифры, задающие значение требуемого символа. Специальным случаем такой конструкции является \0 (не следует ни одной цифры), задающая пустой символ NULL. Escape-последовательность \xddd состоит из обратной косой, за которой следуют 1, 2 или 3 шестнадцатеричных цифры, задающие значение требуемого символа. Если следующий за обратной косой символ не является одним из перечисленных, то обратная косая игнорируется.

2.4.4 Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, мантиссы, e или E и целого показателя степени (возможно, но не обязательно, со знаком). Целая часть и мантисса обе состоят из последовательности цифр. Целая часть или мантисса (но не обе сразу) может быть опущена; или десятичная точка, или e(E) вместе с целым показателем степени (но не обе части одновременно) может быть опущена. Константа с плавающей точкой имеет тип double.

2.4.5 Перечислимые константы

Имена, описанные как перечислители, (см. #8.5) являются константами типа int.

2.4.6 Описанные константы

Объект (#5) любого типа может быть определен как имеющий постоянное значение во всей области видимости (#4.1) его имени. В случае указателей для достижения этого используется декларатор *const; для объектов, не являющихся указателями, используется описатель const (#8.2).

2.5 Строки

Строка есть последовательность символов, заключенная в двойные кавычки: "...". Строка имеет тип "массив символов" и класс памяти static (см. #4 ниже), она инициализируется заданными символами. Все строки, даже если они записаны одинаково, различны. Компилятор располагает в конце каждой строки нулевой (пустой) байт \0 с тем, чтобы сканирующая строку программа могла найти ее конец. В строке перед символом двойной кавычки " обязательно должен стоять \; кроме того, могут использоваться те же escape-последовательности, что были описаны для символьных констант. И, наконец, символ новой строки может появляться только сразу после \; тогда оба, - \ и символ новой строки, - игнорируются.

2.6 Харктеристики аппаратного обеспечения

В нижеследующей таблице собраны некоторые харктеристики аппаратного обеспечения, различающиеся от машины к машине.

	DEC VAX-11 ASCII	Motorola 68000 ASCII	IBM 370 EBCDIC	AT&T 3B ASCII
char	8 бит	8 бит	8 бит	8 бит
int	32 бит	16 бит	32 бит	16 бит
short	16 бит	16 бит	16 бит	16 бит
long	32 бит	32 бит	32 бит	32 бит
float	32 бит	32 бит	32 бит	32 бит
double	64 бит	64 бит	64 бит	64 бит
указатель	32 бит	32 бит	24 бит	32 бит
диапазон float	+_10E+_38	+_10E+_38	+_10E+_76	+_10E+_38
диапазон double	+_10E+_38	+_10E+_38	+_10E+_76	+_10E+_308
тип char	знаковый	без знака	без знака	без знака
тип поля	знаковый	без знака	без знака	без знака
порядок	справа	слева	слева	слева
полей	налево	направо	направо	направо

3. Запись синтаксиса

По используемым в данном руководстве синтаксическим правилам записи синтаксические категории выделяются курсивом а литеральные слова и символы шрифтом постоянной ширины[*]. Альтернативные категории записываются на разных строках. Необязательный терминальный или нетерминальный символ обозначается нижним индексом "opt", так что

```
{ выражение opt }
```

указывает на необязательность выражения в фигурных скобках. Синтаксис кратко изложен в #14.

4. Имена и типы

Имя обозначает (денотирует) объект, функцию, тип, значение или метку. Имя вводится в программе описанием (#8). Имя может использоваться только внутри области текста программы, называемой его областью видимости. Имя имеет тип, определяющий его использование. Объект - это область памяти. Объект имеет класс памяти, определяющий его время жизни. Смысл значения, обнаруженного в объекте, определяется типом имени, использованного для доступа к нему.

4.1 Область видимости

Есть четыре вида областей видимости: локальная, файл, программа и класс.

Локальная:

Имя, описанное в блоке (#9.2), локально в этом блоке и может использоваться только в нем после места описания и в охватываемых блоках. Исключение составляют метки (#9.12), которые могут использоваться в любом месте функции, в которой они описаны. Имена формальных параметров функции рассматриваются так, как если бы они были описаны в самом внешнем блоке этой функции.

Файл:

Имя, описанное вне любого блока (#9.2) или класса (#8.5), может использоваться в файле, где оно описано, после места описания.

Класс:

Имя члена класса локально для его класса и может использоваться только в функции члене этого класса (#8.5.2), после примененной к объекту его класса (#7.1) операции . или после примененной к указателю на объект его класса (#7.1) операции ->. На статические члены класса (#8.5.1) и функции

члены можно также ссылаться с помощью операции :: там, где имя их класса находится в области видимости. Класс, описанный внутри класса (#8.5.15), не считается членом, и его имя принадлежит охватывающей области видимости.

Имя может быть скрыто посредством явного описания того же имени в блоке или классе. Имя в блоке или классе может быть скрыто только именем, описанным в охватываемом блоке или классе. Скрытое нелокальное имя также может использоваться, когда его область видимости указана операцией :: (#7.1). Имя класса, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит class, struct или union (#8.2). Имя перечисления enum, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит enum (#8.2).

4.2 Определения

Описание (#8) является определением, за исключением тех случаев, когда оно описывает функции, не задавая тела функции (#10), когда оно содержит спецификатор extern (1) и в нем нет инициализатора или тела функции, или когда оно является описанием класса (#8.8).

4.3 Компоновка

Имя в файловой области видимости, не описанное явно как static, является общим для каждого файла многофайловой программы. Таковым же является имя функции. О таких именах говорится, что они внешние. Каждое описание внешнего имени в программе относится к тому же объекту (#5), функции (#8.7), классу (#8.5), перечислению (#8.10) или значению перечислителя (#8.10). Типы, специфицированные во всех описаниях внешнего имени должны быть идентичны. Может быть больше одного определения типа, перечисления, inline-функции (#8.1) или несоставного const (#8.2), при условии, что определения идентичны, появляются в разных файлах и все инициализаторы являются константными выражениями (#12). Во всех остальных случаях должно быть ровно одно определение для внешнего имени в программе. Реализация может потребовать, чтобы составное const, использованное там, где не встречено никакого определения const, должно быть явно описано extern и иметь в программе ровно одно определение. Это же ограничение может налагаться на inline-функции.

4.4 Классы памяти

Есть два описываемых класса памяти: автоматический и статический.

Автоматические объекты локальны для каждого вызова блока и сбрасываются по выходе из него.

Статические объекты существуют и сохраняют свое значение в течение выполнения всей программы.

Некоторые объекты не связаны с именами и их времена жизни явно управляются операторами new и delete ; см. #7.2 и #9.14

4.5 Основные типы

Объекты, описанные как символы (char), достаточны для хранения любого элемента машинного набора символов, и если принадлежащий этому набору символ хранится в символьной переменной, то ее значение равно целому коду этого символа.

В настоящий момент имеются целые трех размеров, описываемые как short int, int и long int. Более длинные целые (long int) предоставляют не меньше памяти, чем более короткие целые (short int), но при реализации или длинные, или короткие, или и те и другие могут стать эквивалентными обычным целым. "Обычные" целые имеют естественный размер, задаваемый архитектурой центральной машины; остальные размеры делаются такими, чтобы они отвечали специальным потребностям.

Каждое перечисление (#8.9) является набором именованных констант. Свойства enum идентичны свойствам int.

Целые без знака, описываемые как unsigned, подчиняются правилам арифметики по модулю 2n, где n - число бит в их представлении.

Числа с плавающей точкой одинарной (float) и двойной (double) точности в некоторых машинных реализациях могут быть синонимами.

Поскольку объекты перечисленных выше типов вполне можно интерпретировать как числа, мы будем говорить о них как об арифметических типах. Типы char, int всех размеров и enum будут собирательно называться целыми типами. Типы float и double будут собирательно называться плавающими типами.

Тип данных void (пустой) определяет пустое множество значений. Значение (несуществующее) объекта void нельзя использовать никаким образом, не могут применяться ни явное, ни неявное преобразования. Поскольку пустое выражение обозначает несуществующее значение, такое выражение такое выражение может использоваться только как оператор выражение (#9.1) или как левый операнд в выражении с запятой (#7.15). Выражение может явно преобразовываться к типу void (#7.2).

4.6 Производные типы

Кроме основных арифметических типов концептуально существует бесконечно много производных типов, сконструированных из основных типов следующим образом:

- массивы объектов данного типа;
- функции, получающие аргументы данного типа и возвращающие объекты данного типа;
- указатели на объекты данного типа;
- ссылки на объекты данного типа;
- константы, являющиеся значениями данного типа;
- классы, содержащие последовательность объектов различных типов, множество функций для работы с этими объектами и набор ограничений на доступ к этим объектам и функциям; структуры, являющиеся классами без ограничений доступа;
- объединения, являющиеся структурами, которые могут в разное время содержать объекты разных типов.

В целом эти способы конструирования объектов могут применяться рекурсивно.

Объект типа void* (указатель на void) можно использовать для указания на объекты неизвестного типа.

7. Выражения

Приоритет операций в выражениях такой же, как и порядок главных подразделов в этом разделе, наибольший приоритет у первого. Так например, выражения, о которых говорится как об операндах операции + (#7.4) - это те выражения, которые определены в ##7.1-7.4. Внутри каждого подраздела операции имеют одинаковый приоритет. В каждом подразделе для рассматриваемых в нем операций определяется их левая или правая ассоциативность (порядок обработки операндов). Приоритет и ассоциативность всех операций собран вместе в описании грамматики в #14 .

В остальных случаях порядок вычисления выражения не определен. Точнее, компилятор волен вычислять подвыражения в том порядке, который он считает более эффективным, даже если подвыражения вызывают побочные эффекты. Порядок возникновения побочных эффектов не определен. Выражения, включающие в себя коммутативные и ассоциативные операции (*, +, &, |, ^), могут быть реорганизованы произвольным образом, даже при наличии скобок; для задания определенного порядка вычисления выражения необходимо использовать явную временную переменную.

Обработка переполнения и контроль деления при вычислении выражения машинно-зависимы. В большинстве существующих реализаций C++ переполнение целого игнорируется; обработка деления на 0 и всех исключительных ситуаций с числами с плавающей точкой различаются от машины к машине и обычно могут регулироваться библиотечными функциями.

Кроме стандартного значения, описанного в #7.2-7.15, операции могут быть перегружены[*], то есть, могут быть заданы их значения для случая их применения к типам, определяемым пользователем; см. #7.1.6.

7.1 Основные выражения

Основные выражения, включающие в себя . , -> , индексирование и вызовы функций, группируются слева направо.

```
список_выражений:
     выражение
     список_выражений , выражение
id:
     идентификатор
     имя_функции_операции
     typedef-имя                ::                идентификатор
     typedef-имя :: имя_функции_операции
первичное_выражение:
     id
     ::                                           идентификатор
     константа
     строка
     this
     (                        выражение                        )
     первичное_выражение                  [      выражение     ]
     первичное_выражение         (   список_выражений   opt   )
     первичное_выражение                        .              id
     первичное_выражение    -> id
```

Идентификатор есть первичное выражение, причем соответственно описанное (#8). Имя_функции_операции есть идентификатор со специальным значением; см. #7.1.6 и #8.5.1.

Операция ::, за которой следует идентификатор из файловой области видимости, есть то же, что и идентификатор. Это позволяет ссылаться на объект даже в том случае, когда его идентификатор скрыт (#4.1).

Typedef-имя (#8.8) , за которым следует ::, после чего следует идентификатор, является первичным выражением. Typedef-имя должно обозначать класс (#8.5), и идентификатор должен обозначать член этого класса. Его тип специфицируется описанием идентификатора. Typedef-имя может быть скрыто именем, которое не является именем типа. В этом случае typedef-имя все равно может быть найдено и его можно использовать.

Константа является первичным выражением. Ее тип должен быть int, long или double в зависимости от ее формы.

Строка является первичным выражением. Ее тип - "массив символов". Обычно он сразу же преобразуется в указатель на ее первый символ (#6.7).

Ключевое слово this является локальной переменной в теле функции члена (см. #8.5) . Оно является указателем на объект, для которого функция была вызвана.

Выражение, заключенное в круглые скобки, является первичным выражением, чей тип и значение те же, что и у незаключенного в скобки выражения. Наличие скобок не влияет на то, является выражение lvalue или нет.

Первичное выражение, за которым следует выражение в квадратных скобках, является первичным выражением. Интуитивный смысл - индекс. Обычно первичное выражение имеет тип "указатель на ...", индексирующее выражение имеет тип int и тип результата есть "...". Выражение E1[E2] идентично (по определению) выражению *((E1)+(E2)). Все тонкие места, необходимые для понимания этой записи, содержатся в этом разделе вместе с обсуждением в ## 7.1, 7.2 и 7.4, соответственно, идентификаторов, * и + ; ниже, в #8.4.2 приводятся следствия из этого.

Вызов функции является первичным выражением, за которым следуют скобки, содержащие список (возможно, пустой) разделенных запятыми выражений, составляющих фактические параметры для функции. Первичное выражение должно иметь тип "функция, возвращающая ..." или "указатель на функцию, возвращающую ...", и результат вызова функции имеет тип "...".

Каждый формальный параметр инициализируется фактическим параметром (#8.6). Выполняются стандартные (#6.6-8) и определяемые пользователем преобразования (#8.5.6). Функция может изменять значения своих формальных параметров, но эти изменения не могут повлиять на значения фактических параметров за исключением случая, когда формальный параметр имеет ссылочный тип.

Функция может быть описана как получающая меньше или больше параметров, чем специфицировано в описании функции (#8.4). Каждый фактический параметр типа float, для которого нет формального параметра, преобразуются к типу double; и, как обычно, имена массивов преобразуются к указателям. Порядок вычисления параметров не определен языком; имейте в виду различия между компиляторами.

Допустимы рекурсивные вызовы любых функций.

Первичное выражение, после которого стоит точка, за которой следует идентификатор (или идентификатор, уточненный typedef-именем с помощью операции ::) является выражением. Первое выражение должно быть объектом класса, а идентификатор должен именовать член этого класса. Значением является именованный член объекта, и оно является адресным, если первое выражение является адресным. Следует отметить, что "классовые объекты" могут быть структурами (#8.5.12) или объединениями (#8.5.13).

Первичное выражение, после которого стоит стрелка (->), за которой следует идентификатор (или идентификатор, уточненный typedef-именем с помощью операции ::) является выражением. Первое выражение должно быть указателем на объект класса, а идентификатор должен именовать член этого класса. Значение является адресом, ссылающимся на именованный член класса, на который указывает указательное выражение. Так, выражение E1->MOS есть то же, что и (*E1).MOS. Классы обсуждаются в #8.5.

Если первичное выражение дает значение типа "указатель на ..." (см. #8.4 and #8.6.3), значением выражения был объект, обозначаемый ссылкой. Ссылку можно считать именем объекта; см. #8.6.3.

7.2 Унарные операции

Выражения с унарными операциями группируют справа налево:

```
унарное_выражение:
     унарная_операция                                  выражение
     выражение                                                ++
     выражение                                                --
     sizeof                                            выражение
     sizeof              (              имя_типа               )
     (             имя_типа             )              выражение
     простое_имя_типа        (        список_выражений        )
     new                имя_типа       инициализатор        opt
     new               (               имя_типа               )
     delete                                            выражение
     delete [ выражение ] выражение
унарная_операция:                   одна                    из
     *  &  -  !  ~  ++  --
```

Унарная операция * означает косвенное обращение: выражение должно быть указателем и результатом будет lvalue, ссылающееся на объект, на который указывает выражение. Если выражение имеет тип "указатель на ...", то тип результата есть "...".

Результатом унарной операции & является указатель на объект, на который ссылается операнд. Операнд должен быть lvalue. Если выражение имеет тип "...", то тип результата есть "указатель на ...".

Результатом унарной операции + является значение ее операнда после выполнения обычных арифметических преобразований. Операнд должен быть арифметического типа.

Результатом унарной операции - является отрицательное значение ее операнда. Операнд должен иметь целый тип. Выполняются обычные арифметические преобразования. Отрицательное значение беззнаковой величины вычислятся посредством вычитания ее значения из 2n, где n -число битов в целом типа int.

Результатом операции логического отрицания ! является 1, если значение операнда 0, и 0, если значение операнда не 0. Результат имеет тип int. Применима к любому арифметическому типу или к указателям.

Операция ~ дает дополнение значения операнда до единицы. Выполняются обычные арифметические преобразования. Операнд должен иметь интегральный тип.

7.2.1 Увеличение и Уменьшение

Операнд префиксного ++ получает приращение. Операнд должен быть адресным . Значением является новое значение операнда, но оно не адресное. Выражение ++x эквивалентно x+=1. По поводу данных о преобразованиях см. обсуждение операций сложения (#7.4) и присваивания (#7.14).

Операнд префиксного -- уменьшается аналогично действию префиксной операции ++.

Значение, получаемое при использовании постфиксного ++, есть значение операнда. Операнд должен быть адресным. После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Значение, получаемое при использовании постфиксной --, есть значение операнда. Операнд должен быть адресным. После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

7.2.2 Sizeof

Операция sizeof дает размер операнда в байтах. (Байт не определяется языком иначе, чем через значение sizeof. Однако, во всех существующих реализациях байт есть пространство, необходимое для хранения char.) При применении к массиву результатом является полное количество байтов в массиве. Размер определяется из описаний объектов, входящих в выражение. Семантически это выражение является беззнаковой константой и может быть использовано в любом месте, где требуется константа.

Операцию sizeof можно также применять к заключенному в скобки имени типа. В этом случае она дает размер, в байтах, объекта указанного типа.

7.2.3 Явное Преобразование Типа

Простое_имя_типа (#8.2), возможно, заключенное в скобки, за которым идет заключенное в скобки выражение (или список_выражений, если тип является классом с соответствующим образом описанным конструктором #8.5.5) влечет преобразование значения выражения в названный тип. Чтобы записать преобразование в тип, не имеющий простого имени, имя_типа (#8.7) должно быть заключено в скобки. Если имя типа заключено в скобки, выражение заключать в скобки необязательно. Такая запись называется приведением к типу.

Указатель может быть явно преобразован к любому из интегральных типов, достаточно по величине для его хранения. То, какой из int и long требуется, является машинно-зависимым. Отображающая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине. Подробности для некоторых конкретных машин были приведены в #2.6.

Объект интегрального типа может быть явно преобразован в указатель. Отображающая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно зависимой.

Указатель на один тип может быть явно преобразован в указатель на другой тип. Использование полученного в результате указателя может привести к исключительной ситуации адресации, если исходный указатель не указывает на объект, соответствующим образом выравненный в памяти. Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений. Различные машины могут различаться по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Объект может преобразовываться в объект класса только если был описан соответствующий конструктор или операция преобразования (#8.5.6).

Объект может явно преобразовываться в ссылочный тип &X, если указатель на этот объект может явно преобразовываться в X*.

7.2.4 Свободная Память

Операция new создает объект типа имя_типа (см. #8.7), к которому он применен. Время жизни объекта, созданного с помощью new, не ограничено областью видимости, в которой он создан. Операция new возвращает указатель на созданный ей объект. Когда объект является массивом, возвращается указатель на его первый элемент. Например, и new int и new int[10] возвращают int*. Для объектов некоторых классов надо предоставлять инициализатор (#8.6.2). Операция new (#7.2) для получения памяти вызывает функцию

```
void* operator new (long);
```

Параметр задает требуемое число байтов. Память будет инициализирована. Если operator new() не может найти требуемое количество памяти, то она возвращает ноль.

Операция delete уничтожает объект, созданный операцией new. Ее результат является void. Операнд delete должен быть указателем, возвращенным new. Результат применения delete к указателю, который не был получен с помощью операции new. Однако уничтожение с помощью delete указателя со значением ноль безвредно.

Чтобы освободить указанную память, операция delete вызывает функцию

```
void operator delete (void*);
```

В форме

```
delete [ выражение ] выражение
```

второй параметр указывает на вектор, а первое выражение задает число элементов этого вектора. Задание числа элементов является избыточным за исключением случаев уничтожения векторов некоторых классов; см. #8.5.8.

7.3 Мультипликативные операции

Мультипликативные операции *, / и % группируют слева направо. Выполняются обычные арифметические преобразования.

```
мультипликативное_выражение:
    выражение * выражение
    выражение  /  выражение
    выражение  % выражение
```

Бинарная операция * определяет умножение. Операция * ассоциативна и выражения с несколькими умножениями на одном уровне могут быть реорганизованы компилятором.

Бинарная операция / определяет деление. При делении положительных целых округление осуществляется в сторону 0, но если какой-либо из операндов отрицателен, то форма округления является машинно-зависимой. На всех машинах, охватываемых данным руководством, остаток имеет тот же знак, что и делимое. Всегда истинно, что (a/b)*b + a%b равно a (если b не 0).

Бинарная операция % дает остаток от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть числами с плавающей точкой.

7.4 Аддитивные операции

Аддитивные операции + и - группируют слева направо. Выполняюься обычные арифметические преобразования. Каждая операция имеет некоторые дополнительные возможности, связанные с типами.

```
аддитивное_выражение:
    выражение + выражение
```

```
выражение - выражение
```

Результатом операции + является сумма операндов. Можно суммировать указатель на объект массива и значение целого типа. Последнее во всех случаях преобразуется к смещению адреса с помощью умножения его на длину объекта, на который указывает указатель. Результатом является указатель того же типа, что и исходный указатель, указывающий на другой объект того же массива и соответствующим образом смещенный от первоначального объекта. Так, если P есть указатель на объект массива, то выражение P+1 есть указатель на следующий объект массива.

Никакие другие комбинации типов для указателей не допустимы.

Операция + ассоциативна и выражение с несколькими умножениями на одном уровне может быть реорганизовано компилятором.

Результатом операции - является разность операндов. Выполняются обычные арифметические преобразования. Кроме того, значение любого целого типа может вычитаться из указателя, в этом случае применяются те же преобразования, что и к сложению.

Если вычитаются указатели на объекты одного типа, то результат преобразуется (посредством деления на длину объекта) к целому, представляющему собой число объектов, разделяющих объекты, указанные указателями. В зависимости от машины результирующее целое может быть или типа int, или типа long; см. #2.6. Вообще говоря, это преобразование будет давать неопределенный результат кроме тех случаев, когда указатели указывают на объекты одного массива, поскольку указатели, даже на объекты одинакового типа, не обязательно различаются на величину, кратную длине объекта.

7.5 Операции сдвига

Операции сдвига << и >> группируют слева направо. Обе выполняют одно обычное арифметическое преобразование над своими операндами, каждый из которых должен быть целым. В этом случае правый операнд преобразуется к типу int; тип результата совпадает с типом левого операнда. Результат не определен, если правый операнд отрицателен или больше или равен длине объекта в битах.

```
сдвиговое_выражение:
     выражение                  <<                  выражение
     выражение >> выражение
```

Значением E1 << E2 является E1 (рассматриваемое как битовое представление), сдвинутое влево на E2 битов; освободившиеся биты заполняются нулями. Значением E1 >> E2 является E1 , сдвинутое вправо на E2 битовых позиций. Гарантируется, что сдвиг вправо является логическим (заполнение нулями), если E1 является unsigned; в противном случае он может быть арифметическим (заполнение копией знакового бита).

7.6 Операции отношения

Операции отношения (сравнения) группируют слева направо, но этот факт не очень-то полезен: a < b < c не означает то, чем кажется.

```
выражение_отношения:
     выражение                   <                  выражение
     выражение                   >                  выражение
     выражение                  <=                  выражение
     выражение >= выражение
```

Операции < (меньше чем), > (больше чем), <= и >= все дают 0, если заданное соотношение ложно, и 1, если оно истинно. Тип результата int. Выполняются обычные арифметические преобразования. Могут сравниваться два указателя; результат зависит от относительного положения объектов, на которые указывают указатели, в адресном пространстве. Сравнение указателей переносимо только если указатели указывают на объекты одного массива.

7.7 Операции равенства

```
выражение_равенства:
     выражение                    ==                    выражение
     выражение   !=   выражение
```

Операции == и != в точности аналогичны операциям сравнения за исключением их низкого приоритета. (Так, a < b == c < d есть 1 всегда, когда a < b и c < d имеют одинаковое истинностное значение.)

Указатель может сравниваться с 0.

7.8 Операция побитовое И

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

Операция & ассоциативна, и выражения, содержащие &, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция И операндов. Операция применяется только к целым операндам.

7.9 Операция побитовое исключающее ИЛИ

```
исключающее_ИЛИ_выражение:
     выражение ^ выражение
```

Операция ^ ассоциативна, и выражения, содержащие ^, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция исключающее ИЛИ операндов. Операция применяется только к целым операндам.

7.10 Операция побитовое включающее ИЛИ

```
включающее_ИЛИ_выражение:
     выражение | выражение
```

Операция | ассоциативна, и выражения, содержащие |, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция включающее ИЛИ операндов. Операция применяется только к целым операндам.

7.11 Операция логическое И

```
логическое_И_выражение:
     выражение && выражение
```

Операция && группирует слева направо. Она возвращает 1, если оба операнда ненулевые, и 0 в противном случае. В противоположность операции & операция && гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд есть 0.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип int.

7.12 Операция логическое ИЛИ

```
логическое_ИЛИ_выражение:
     выражение || выражение
```

Операция || группирует слева направо. Она возвращает 1, если хотя бы один из ее операндов ненулевой, и 0 в противном случае. В противоположность операции | операция || гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд не есть 0.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип int.

7.13 Условная операция

```
условное_выражение:
     выражение ? выражение : выражение
```

Условная операция группирует слева направо. Вычисляется первое выражение, и если оно не 0, то результатом является значение второго выражения, в противном случае значение третьего выражения. Если это возможно, то выполняются обычные арифметические преобразования для приведения второго и третьего выражения к общему типу. Если это возможно, то выполняются преобразования указателей для приведения второго и третьего выражения к общему типу. Вычисляется только одно из второго и третьего выражений.

7.14 Операции присваивания

Есть много операций присваивания, все группируют слева направо. Все в качестве левого операнда требуют lvalue, и тип выражения присваивания тот же, что и у его левого операнда. Это lvalue не может ссылаться на константу (имя массива, имя функции или const). Значением является значение, хранящееся в левом операнде после выполнения присваивания.

```
выражение_присваивания:
    выражение  операция_присваивания  выражение
операция_присваивания: одна из
    =  +=  -=  *=  /=  %=  >>=  <<=  &=  ~=  |=
```

В простом присваивании с = значение выражения замещает собой значение объекта, на который ссылается операнд в левой части. Если оба операнда имеют арифметический тип, то при подготовке к присваиванию правый операнд преобразуется к типу левого. Если аргумент в левой части имеет указательный тип, аргумент в правой части должен быть того же типа или типа, который может быть преобразован к нему, см. #6.7. Оба операнда могут быть объектами одного класса. Могут присваиваться объекты некоторых производных классов; см. #8.5.3.

Присваивание объекту типа "указатель на ..." выполнит присваивание объекту, денотируемому ссылкой.

Выполнение выражения вида E1 op= E2 можно представить себе как эквивалентное E1 = E1 op (E2); но E1 вычисляется только один раз. В += и -= левый операнд может быть указателем, и в этом случае (интегральный) правый операнд преобразуется так, как объяснялось в #7.4; все правые операнды и не являющиеся указателями левые должны иметь арифметический тип.

7.15 Операция запятая

```
запятая_выражение:
    выражение , выражение
```

Пара выражений, разделенных запятой, вычисляется слева направо, значение левого выражения теряется. Тип и значение результата являются типом и значением правого операнда. Эта операция группирует слева направо. В контексте, где запятая имеет специальное значение, как например в списке фактических параметров функции (#7.1) и в списке инициализаторов (#8.6), операция запятая, как она описана в этом разделе, может появляться только в скобках; например,

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

имеет три параметра, вторым из которых является значение 5.

7.16 Перегруженные операции

Большинство операций может быть перегружено, то есть, описано так, чтобы они получали в качестве операндов объекты классов (см. #8.5.11). Изменить приоритет операций невозможно. Невозможно изменить смысл операций при применении их к неклассовым объектам. Предопределенный смысл операций = и & (унарной) при применении их к объектам классов может быть изменен.

Эквивалентность операций, применяемых к основным типам (например, ++a эквивалентно a+=1), не обязательно выполняется для операций, применяемых к классовым типам. Некоторые операции, например, присваивание, в случае применения к основным типам требуют, чтобы операнд был lvalue; это не требуется для операций, описанных для классовых типов.

7.16.1 Унарные операции

Унарная операция, префиксная или постфиксная, может быть определена или с помощью функции члена (см. #8.5.4), не получающей параметров, или с помощью функции друга (см. #8.5.10), получающей один параметр, но не двумя способами одновременно. Так, для любой унарной операции @, x@ и @x могут интерпретироваться как x.операция@() или операция@(x). При перегрузке операций ++ и -- невозможно различить префиксное и постфиксное использование.

7.16.2 Бинарные операции

Бинарная операция может быть определена или с помощью функции члена (см. #8.5.4), получающей один параметр, или с помощью функции друга (см. #8.5.9), получающей два параметра, но не двумя способами одновременно. Так, для любой бинарной операции @, x@y может быть проинтерпретировано как x.операция@(y) или операция@(x,y).

7.16.3 Особые операции

Вызов функции

```
первичное_выражение ( список_выражений opt )
```

и индексирование

```
первичное_выражение [ выражение ]
```

считаются бинарными операциями. Именами определяющей функции являются соответственно operator() и operator[]. Обращение x(arg) интерпретируется как x.operator()(arg) для классового объекта x. Индексирование x[y] интерпретируется как x.operator[](y).

* Этот термин применяется для описания использования в языке одной и той же лексемы для обозначения различных процедур; вид процедуры выбирается компилятором на основании дополнительной информации в виде числа и типа аргументов и т.п.

8. Описания

Описания используются для определения интерпретации, даваемой каждому идентификатору; они не обязательно резервируют память, связанную с идентификатором. Описания имеют вид:

```
описание:
    спецификаторы_описания opt список_описателей opt ;
    описание_имени
    asm_описание
```

Описатели в списке_описателей содержат идентификаторы, подлежащие описанию. Спецификаторы_описания могут быть опущены только в определениях внешних функций (#10) или в описаниях внешних функций. Список описателей может быть пустым только при описании класса (#8.5) или перечисления (#8.10), то есть, когда спецификаторы_описания - это class_спецификатор или enum_спецификатор. Описания имен описываются в #8.8; описания asm описаны в #8.11.

```
спецификатор_описания:
    sc_спецификатор
    спецификатор_типа
    фнк_спецификатор
    friend
    typedef
спецификаторы_описания:
    спецификатор_описания спецификатор_описания opt
```

Список должен быть внутренне непротиворечив в описываемом ниже смысле.

8.1 Спецификаторы класса памяти

Спецификаторы "класса памяти" (sc-спецификатор) это:

```
sc-спецификатор:
    auto
    static
    extern
    register
```

Описания, использующие спецификаторы auto, static и register также служат определениями тем, что они вызывают резервирование соответствующего объема памяти. Если описание extern не является определением (#4.2), то где-то еще должно быть определение для данных идентификаторов.

Описание register лучше всего представить как описание auto (автоматический) с подсказкой компилятору, что описанные переменные усиленно используются. Подсказка может быть проигнорирована. К ним не может применяться операция получения адреса &.

Спецификаторы auto или register могут применяться только к именам, описанным в блоке, или к формальным параметрам. Внутри блока не может быть описаний ни статических функций, ни статических формальных параметров.

В описании может быть задан максимум один sc_спецификатор. Если в описании отсутсвует sc_спецификатор, то класс памяти принимается автоматическим внутри функции и статическим вне. Исключение: функции не могут быть автоматическими.

Спецификаторы static и extern могут использоваться только для имен объектов и функций.

Некоторые спецификаторы могут использоваться только в описаниях функций:

```
фнк-спецификатор:
    overload
    inline
    virtual
```

Спецификатор перегрузки overload делает возможным использование одного имени для обозначения нескольких функций; см. #8.9.

Спецификатор inline является только подсказкой компилятору, не влияет на смысл программы и может быть проигнорирован. Он используется, чтобы указать на то, что при вызове функции inline- подстановка тела функции предпочтительнее обычной реализации вызова функции. Функция (#8.5.2 и #8.5.10), определенная внутри описания класса, является inline по умолчанию.

Спецификатор virtual может использоваться только в описаниях членов класса; см. #8.5.4.

Спецификатор friend используется для отмены правил скрытия имени для членов класса и может использоваться только внутри описаний классов; см. #8.5.9.

С помощью спецификатора typedef вводится имя для типа; см. #8.8.

8.2 Спецификаторы Типа

Спецификаторами типов (спецификатор_типа) являются:

```
спецификатор_типа:
    простое_имя_типа
    class_спецификатор
```

```
    enum-спецификатор
    сложный_спецификатор_типа
    const
```

Слово const можно добавлять к любому допустимому спецификатору_типа. В остальных случаях в описании может быть дано не более одного спецификатора_типа. Объект типа const не является lvalue. Если в описании опущен спецификатор типа, он принимается int.

```
простое_имя_типа:
    char
    short
    int
    long
    unsigned
    float
    double
    const
    void
```

Слова long, short и unsigned можно рассматривать как прилагательные. Они могут применяться к типу int; unsigned может также применяться к типам char, short и long.

Спецификаторы класса и перечисления обсуждаются в #8.5 и #8.10 соответственно.

```
сложный_спецификатор_типа:
    ключ typedef-имя
    ключ идентификатор
ключ:
    class
    struct
    union
    enum
```

Сложный спецификатор типа можно использовать для ссылки на имя класса или перечисления там, где имя может быть скрыто локальным именем. Например:

```
class x { ... };
void f(int x)
{
    class x a;
    // ...
}
```

Если имя класса или перечисления ранее описано не было, сложный_спецификатор_типа работает как описание_имени; см. #8.8.

8.3 Описатели

Список_описателей, появляющийся в описании, есть разделенная запятыми последовательность описателей, каждый из которых может иметь инициализатор.

```
список_описателей:
    иниц_описатель
    иниц_описатель  ,  список_описателей
иниц_описатель:
    описатель  инициализатор opt
```

Инициализаторы обсуждаются в #8.6. Спецификатор в описании указывает тип и класс памяти объектов, к которым относятся описатели. Описатели имеют синтаксис:

```
описатель:
```

```
        оп_имя
        ( описатель )
        * const opt  описатель
        & const opt  описатель
        описатель    ( список_описаний_параметров )
        описатель      [ константное_выражение opt ]
    оп-имя:
        простое_оп_имя
        typedef-имя  ::  простое_оп_имя
    простое_оп_имя:
        идентификатор
        typedef-имя
        ~  typedef-имя
        имя_функции_операции
        имя_функции_преобразования
```

Группировка та же, что и в выражениях.

8.4 Смысл описателей

Каждый описатель считается утверждением того, что если в выражении возникает конструкция, имеющая ту же форму, что и описатель, то она дает объект указанного типа и класса памяти. Каждый описатель содержит ровно одно оп_имя; оно определяет описываемый идентификатор. За исключением описаний некоторых специальных функций (см. #8.5.2) , оп_имя будет простым идентификатором.

Если в качестве описателя возникает ничем не снабженный идентификатор, то он имеет тип, указанный спецификатором, возглавляющим описание.

Описатель в скобках эквивалентен описателю без скобок, но связку сложных описателей скобки могут изменять.

Теперь представим себе описание

```
T D1
```

где T - спецификатор типа (как int и т.д.), а D1 - описатель. Допустим, что это описание заставляет идентификатор иметь тип "... T", где "..." пусто, если идентификатор D1 есть просто обычый идентификатор (так что тип x в "int x" есть просто int). Тогда, если D1 имеет вид

```
*D
```

то тип содержащегося идентификатора есть "... указатель на T."

Если D1 имеет вид

```
* const D
```

то тип содержащегося идентификатора есть "... константный указатель на T", то есть, того же типа, что и *D, но не lvalue.

Если D1 имеет вид

```
&D
```

или

```
& const D
```

то тип содержащегося идентификатора есть "... ссылка на T." Поскольку ссылка по определению не может быть lvalue, использование const излишне. Невозможно иметь ссылку на void (void&).

Если D1 имеет вид

```
D (список_описаний_параметров)
```

то содержащийся идентификатор имеет тип "... функция, принимающая параметр типа список_описаний_параметров и возвращающая T."

```
список_описаний_параметров:
    список_описаний_парам opt ... opt
список_описаний_парам:
    список_описаний_парам  ,  описание_параметра
    описание_параметра
описание_параметра:
    спецификаторы_описания  описатель
    спецификаторы_описания  описатель  = выражение
    спецификаторы_описания  абстракт_описатель
    спецификаторы_описания  абстракт_описатель  =  выражение
```

Если список_описаний_параметров заканчивается многоточием, то о числе параметров известно лишь, что оно равно или больше числа специфицированных типов параметров; если он пуст, то функция не получает ни одного параметра. Все описания для функции должны согласовываться и в типе возвращаемого значения, а также в числе и типе параметров.

Список_описаний_параметров используется для проверки и преобразования фактических параметров и для контроля присваивания указателю на функцию. Если в описании параметра специфицировано выражение, то это выражение используется как параметр по умолчанию. Параметры по умолчанию будут использоваться в вызовах, где опущены стоящие в хвосте параметры. Параметр по умолчанию не может переопределяться более поздними описаниями. Однако, описание может добавлять параметры по умолчанию, не заданные в предыдущих описаниях.

Идентификатор может по желанию быть задан как имя параметра. Если он присутствует в описании функции, его использовать нельзя, поскольку он сразу выходит из области видимости. Если он присутствует в определении функции (#10), то он именует фармальный параметр.

Если D1 имеет вид

```
D[ константное_выражение]
```

или

```
D[]
```

то тип содержащегося идентификатора есть "... массив объектов типа T". В первом случае константное_выражение есть выражение, значение которого может быть определено во время компиляции, и тип которого int. (Константные выражения определены в #12.) Если подряд идут несколько спецификаций "массив из", то создается многомерный массив; константное выражение, определяющее границы массива, может быть опущено только для первого члена последовательности. Этот пропуск полезен, когда массив является внешним, и настоящее определение, которое резервирует память, находится в другом месте. Первое константное выражение может также быть опущено, когда за описателем следует инициализация. В этом случае используется размер, вычисленный исходя из числа начальных элементов.

Массив может быть построен из одного из основных типов, из указателей, из структуры или объединения или из другого массива (для получения многомерного массива).

Не все возможности, которые позволяет приведенный выше синтаксис, допустимы. Ограничения следующие: функция не может возвращать массив или функцию, хотя она может возвращать указатели на эти объекты; не существует массивов функций, хотя могут быть массивы указателей на функции.

8.4.1 Примеры

В качестве примера, описание

```
int i;
int *ip;
int f ();
int *fip ();
int (*pfi) ();
```

описывает целое i, указатель ip на целое, функцию f, возвращающую целое, функцию fip , возвращающую указатель на целое, и указатель pfi на функцию, возвращающую целое. Особенно полезно сравнить последние две. Цепочка *fip() есть *(fip()), как предполагается в описании, и та же конструкция требуется в

выражении, вызов функции fip, и затем косвенное использование результата через (указатель) для получения целого. В описателе (*pfi)() внешние скобки необходимы, поскольку они также входят в выражение, для указания того, что функция получается косвенно через указатель на фунцию, которая затем вызывается; это возвращает целое. Функции f и fip описаны как не получающие параметров, и fip как указывающая на функцию, не получающую параметров.

Описание

```
const a = 10, *pc = &a, *const cpc = pc;
int b, *const cp = &b;
```

описывает a: целую константу, pc: указатель на целую константу, cpc: константный указатель на целую константу, b: целое и cp: константный указатель на целое. Значения a, cpc и cp не могут быть изменены после инициализации. Значение pc может быть изменено, как и объект, указываемый cp. Примеры недопустимых выражений :

```
a = 1;
a++;
*pc = 2;
cp = &a;
cpc++;
```

Примеры допустимых выражений :

```
b = a;
*cp = a;
pc++;
pc = cpc;
```

Описание

```
fseek (FILE*,long,int);
```

описывает функцию, получающую три параметра специальных типов. Поскольку тип возвращаемого значения не определен, принимается, что он int (#8.2). Описание

```
point (int = 0,int = 0);
```

описывает функцию, которая может быть вызвана без параметров, с одним или двумя параметрами типа int. Например

```
point (1,2);
point (1)             /* имеет смысл point (1,0); */
point ()              /* имеет смысл point (0,0); */
```

Описание

```
printf (char* ... );
```

описывает функцию, которая может быть вызываться с различными числом и типами параметров. Например

```
printf ("hello, world");
printf ("a=%d b=%d",a,b);
printf ("string=%s",st);
```

Однако, она всегда должна иметь своим первым параметром char*.

В качестве другого примера,

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

описывает массив чисел с плавающей точкой и массив указателей на числа с плавающей точкой. И, наконец,

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

описывает массив целых, размером 3x6x7. Совсем подробно: x3d является массивом из трех элементов; каждый из элементов является массивом из пяти элементов; каждый из последних элементов является массивом из семи целых. Появление каждое из выражений x3d, x3d[i], x3d[i][j], x3d[i][j][k] может быть приемлемо. Первые три имеют тип "массив", последний имеет тип int.

8.5 Описания классов

Класс специфицирует тип. Его имя становится typedef-имя (см. #8.8), которое может быть использовано даже внутри самого спецификатора класса. Объекты класса состоят из последовательности членов.

```
спецификатор_класса:
     заголовок_класса     {      список_членов      opt      }
     заголовок_класса         {  список_членов  opt  public  :
список_членов                     opt                        }
заголовок_класса:
     агрег                  идентификатор                  opt
     агрег идентификатор opt : public opt typedef-имя
агрег:
     class
     struct
     union
```

Структура является классом, все члены которого общие; см. #8.5.8. Объединение является классом, содержащим в каждый момент только один член; см. #8.5.12. Список членов может описывать члены вида: данные, функция, класс, определение типа, перечисление и поле. Поля обсуждаются в #8.5.13. Список членов может также содержать описания, регулирующие видимость имен членов; см. #8.5.8.

```
список_членов:
     описание_члена  список_членов opt
описание_члена:
     спецификаторы_описания        opt        описатель_члена;
описатель_члена:
     описатель                          идентификатор   opt   :
константное_выражение
```

Члены, являющиеся классовыми объектами, должны быть объектами предварительно полностью описанных классов. В частности, класс cl не может содержать объект класса cl, но он может содержать указатель на объект класса cl.

Имена объектов в различных классах не конфликтуют между собой и с обычными переменными.

Вот простой пример описания структуры:

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

содержащей массив из 20 символов, целое и два указателя на такие же структуры. Если было дано такое описание, то описание

```
tnode s, *sp
```

описывает s как структуру данного сорта и sp как указатель на структуру данного сорта. При наличии этих описаний выражение

```
sp->count
```

ссылается на поле count структуры, на которую указывает sp;

```
s.left
```

ссылается на указатель левого поддерева структуры s; а

```
 s.right->tword[0]
```

ссылается на первый символ члена tword правого поддерева структуры s.

8.5.1 Статические члены

Член-данные класса может быть static; члены-функции не могут. Члены не могут быть auto, register или extern. Есть единственная копия статического члена, совместно используемая всеми членами класса в программе. На статический член mem класса cl можно ссылаться cl:mem, то есть без ссылки на объект. Он существует, даже если не было создано ни одного объекта класса cl.

8.5.2 Функции члены

Функция, описанная как член, (без спецификатора friend (#8.5.9)) называется функцией членом и вызывается с помощью синтаксиса члена класса (#7.1). Например:

```
struct tnode
  {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
    void set (char* w,tnode* l,tnode* r);
  };
tnode n1, n2;
n1.set ("asdf",&n2,0);
n2.set ("ghjk",0,0);
```

Определение функции члена рассматривается как находящееся в области видимости ее класса. Это значит, что она может непосредственно использовать имена ее класса. Если определение функции члена находится вне описания класса, то имя функции члена должно быть уточнено именем класса с помощью записи

```
typedef-имя . простое_оп_имя
```

см. 3.3. Определения функций обсуждаются в #10.1. Например:

```
void tnode.set (char* w,tnode* l,tnode* r)
  {
    count = strlen (w);
    if (sizeof (tword) <= count) error ("tnode string too long");
    strcpy (tword,w);
    left = l;
    right = r;
  }
```

Имя функции tnode.set определяет то, что множество функций является членом класса tnode. Это позволяет использовать имена членов word, count, left и right. В функции члене имя члена ссылается на объект, для которого была вызвана функция. Так, в вызове n1.set(...) tword ссылается на n1.tword, а в вызове n2.set(...) он ссылается на n2.tword. В этом примере предполагается, что функции strlen, error и strcpy описаны где-то в другом месте как внешние функции (см. #10.1).

В члене функции ключевое слово this указывает на объект, для которого вызвана функция. Типом this в функции, которая является членом класса cl, является cl*. Если mem - член класса cl,то mem и this->mem - синонимы в функции члене класса cl (если mem не был использован в качестве имени локальной переменной в промежуточной области видимости).

Функция член может быть определена (#10.1) в описании класса. Помещение определения функции члена в описание класса является кратким видом записи описания ее в описании класса и затем определения ее как inline (#8.1) сразу после описания класса. Например:

```
int b;
struct x
  {
    int f () { return b; }
    int f () { return b; }
    int b;
  };
```

означает

```
int b;
struct x
  {
    int f ();
    int b;
  };
inline x.f () { return b; }
```

Для функций членов не нужно использование спецификатора overload (#8.2): если имя описывается как означающее несколько имен в классе, то оно перегружено (см. #8.9).

Применение операции получения адреса к функциям членам допустимо. Тип параметра результирующей функции указатель на есть (...), то есть, неизвестен (#8.4). Любое использование его является зависимым от реализации, поскольку способ инициализации указателя для вызова функции члена не определен.

8.5.3 Производные классы

В конструкции

```
аргег идентификатор:public opt typedef-имя
```

typedef-имя должно означать ранее описанный класс, называемый базовым классом для класса, подлежащего описанию. Говорится, что последний выводится из предшествующего. На члены базового класса можно ссылаться, как если бы они были членами производного класса, за исключением тех случаев, когда имя базового члена было переопределено в производном классе; в этом случае для ссылки на скрытое имя может использоваться такая запись (#7.1):

```
typedef-имя :: идентификатор
```

Например:

```
struct base
  {
    int a;
    int b;
  };
struct derived : public base
  {
    int b;
    int c;
  };
derived d;
d.a = 1;
d.base::b = 2;
d.b = 3;
d.c = 4;
```

осуществляет присваивание четырем членам d.

Производный тип сам может использоваться как базовый.

8.5.4 Виртуальные функции

Если базовый класс base содержит (виртуальную) virtual (#8.1) функцию vf, а производный класс derived также содержит функцию vf, то вызов vf для объекта класса derived вызывает derived::vf. Например:

```
struct base
  {
    virtual void vf ();
    void f ();
  };
 struct derived : public base
  {
    void vf ();
    void f ();
  };
derived d;
base* bp = &d;
bp->vf ();
bp->f ();
```

Вызовы вызывают, соответственно, derived::vf и base::f для объекта класса derived, именованного d. Так что интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызвана, в то время как интерпретация вызова невиртуальной функции зависит только от типа указателя, обозначающего объект.

Из этого следует, что тип объектов классов с виртуальными функциями и объектов классов, выведенных из таких классов, могут быть определены во время выполнения.

Если производный класс имеет член с тем же именем, что и у виртуальной функции в базовом классе, то оба члена должны иметь одинаковый тип. Виртуальная функция не может быть другом (friend) (#8.5.9). Функция f в классе, выведенном из класса, который имеет виртуальную функцию f, сама рассматривается как виртуальная. Виртуальная функция в базовом классе должна быть определена. Виртуальная функция, которая была определена в базовом классе, не нуждается в определении в производном классе. В этом случае функция, определенная для базового класса, используется во всех вызовах.

8.5.5 Конструкторы

Член функция с именем, совпадающим с именем ее класса, называется конструктором. Конструктор не имеет типа возвращаемого значения; он используется для конструирования значений с типом его класса. С помощью конструктора можно создавать новые объекты его типа, используя синтаксис

```
typedef-имя ( список_параметров opt )
```

Например,

```
complex zz = complex (1,2.3);
cprint (complex (7.8,1.2));
```

Объекты, созданные таким образом, не имеют имени (если конструктор не использован как инициализатор, как это было с zz выше), и их время жизни ограничено областью видимости, в которой они созданы. Они не могут рассматриваться как константы их типа. Если класс имеет конструктор, то он вызывается для каждого объекта этого класса перед тем, как этот объект будет как-либо использован; см. #8.6.

Конструктор может быть overload, но не virtual или friend.

Если класс имеет базовый класс с конструктором, то конструктор для базового класса вызывается до вызова конструктора для производного класса. Конструкторы для объектов членов, если таковые есть, выполняются после конструктора базового класса и до конструктора объекта, содержащего их. Объяснение того, как могут быть специфицированы параметры для базового класса, см. в #8.6.2, а того, как конструкторы могут использоваться для управления свободной памятью, см. в #17.

8.5.6 Преобразования

Конструктор, получающий один параметр, определяет преобразование из типа своего параметра в тип своего класса. Такие преобразования неявно применяются дополнительно к обычным арифметическим преобразованиям. Поэтому присваивание объекту из класса X допустимо, если или присваиваемое значение является X, или если X имеет конструктор, который получает присваиваемое значение как свой единственный параметр. Аналогично конструкторы используются для преобразования параметров функции (#7.1) и инициализаторов (#8.6). Например:

```
class X { ... X (int); };
f (X arg)
  {
    X a = 1;          /* a = X (1) */
    a = 2;            /* a = X (2) */
    f (3);            /* f (X (3)) */
  }
```

Если для класса X не найден ниодин конструктор, принимающий присваиваемый тип, то не делается никаких попыток отыскать конструктор для преобразования присваиваемого типа в тип, который мог бы быть приемлем для конструкторов класса X. Например:

```
class X { ... X (int); };
class X { ... Y (X); };
Y a = 1;              /* недопустимо: Y (X (1)) не пробуется */
```

8.5.7 Деструкторы

Функция член класса cl с именем ~cl называется деструктором. Деструктор не возвращает никакого значения и не получает никаких параметров; он используется для уничтожения значений типа cl непосредственно перед уничтожением содержащего их объекта. Деструктор не может быть overload, virtual или friend.

Деструктор для базового класса выполняется после деструктора производного от него класса. Как деструкторы используются для управления свободной памятью, см. объяснение в #17.

8.5.8 Видимость имен членов

Члены класса, описанные с ключевым словом class, являются закрытыми, это значит, что их имена могут использоваться только функциями членами (#8.5.2) и друзьями (см. #8.5.10), пока они не появятся после метки public: . В этом случае они являются общими. Общий член может использоваться любой функцией. Структура является классом, все члены которого общие; см. #8.5.11.

Если перед именем базового класса в описании производного класса стоит ключевое слово public, то общие члены базового класса являются общими для производного класса; если нет, то они являются закрытыми. Общий член mem закрытого базового класса base может быть описан как общий для производного класса с помощью описания вида

```
typedef-имя . идентификатор;
```

в котором typedef-имя означает базовый класс, а идентификатор есть имя члена базового класса. Такое описание может появляться в общей части производного класса.

Рассмотрим

```
class base
  {
    int a;
public:
    int b,c;
    int bf ();
  };
class derived : base
  {
    int d;
public:
    base.c;
    int e;
    int df ();
  };
int ef (derived&);
```

Внешняя функция ef может использовать только имена c, e и df. Являясь членом derived, функция df может использовать имена b, c, bf, d, e и df, но не a. Являясь членом base, функция bf может использовать члены a, b, c и bf.

8.5.9 Друзья (friends)

Другом класса является функция не-член, которая может использовать имена закрытых членов. Следующий пример иллюстрирует различия между членами и друзьями:

```
class private
  {
    int a;
    friend void friend_set (private*,int);
public:
    void member_set (int);
  };
void friend_set (private* p,int i) { p->a=i; }
void private.member_set (int i) { a = i; }
private obj;
friend_set (&obj,10);
obj.member_set (10);
```

Если описание friend относится к перегруженному имени или операции, то другом становится только функция с описанными типами параметров. Все функции класса cl1 могут быть сделаны друзьями класса cl2 с помощью одного описания

```
class cl2
  {
    friend cl1;
    . . .
  };
```

8.5.10 Функция операция

Большинство операций могут быть перегружены с тем, чтобы они могли получать в качестве операндов объекты класса.

```
  имя_функции_операции:   operator  op
op:             +   -   *   /   %   ^   &   |   ~
                !   =   <   >   +=  -=  *=  /=  %=
                ^=  &=  |=  <<  >>  <<= >>= ==  !=
                <=  >=  &&  ||  ++  --  ()  []
```

Последние две операции - это вызов функции и индексирование. Функция операция может или быть функцией членом, или получать по меньшей мере один параметр класса. См. также #7.1.6.

8.5.11 Структуры

Структура есть класс, все члены которого общие. Это значит, что

```
struct s { ... };
```

эквивалентно

```
class s { public: ... };
```

Структура может иметь функции члены (включая конструкторы и деструкторы).

8.5.12 Объединения

Объединение можно считать структурой, все объекты члены которой начинаются со смещения 0, и размер которой достаточен для содержания любого из ее объектов членов. В каждый момент времени в объединении может храниться не больше одного из объектов членов. Объединение может иметь функции члены (включая конструкторы и деструкторы).

8.5.13 Поля бит

Описатель члена вида

```
        идентификатор opt:  константное_выражение
```
определяет поле; его длина отделяется от имени поля двоеточием. Поля упаковываются в машинные целые; они не являются альтернативой слов. Поле , не влезающее в оставшееся в целом место, помещается в следующее слово. Поле не может быть шире слова. На некоторых машинах они размещаются справа налево, а на некоторых слева направо; см. #2.6.

Неименованные поля полезны при заполнении для согласования внешне предписанных размещений (форматов). В особых случаях неименованные поля длины 0 задают выравнивание следующего поля по границе слова. Не требуется аппаратной поддержки любых полей, кроме целых. Более того, даже целые поля могут рассматриваться как unsigned. По этим причинам рекомендуется описывать поля как unsigned. К полям не может применяться операция получения адреса &, поэтому нет указателей на поля.

Поля не могут быть членами объединения.

8.5.14 Вложенные классы

Класс может быть описан внутри другого класса. В этом случае область видимости имен внутреннего класса его и общих имен ограничивается охватывающим классом. За исключением этого ограничения допустимо, чтобы внутренний класс уже был описан вне охватывающего класса. Описание одного класса внутри другого не влияет на правила доступа к закрытым членам и не помещает функции члены внутреннего класса в область видимости охватывающего класса. Например:

```
int x;
class enclose   /* охватывающий */
  {
    int x;
    class inner
      {
        int y;
        f () { x=1 }
        ...
      };
    g (inner*);
    ...
  };
int inner;  /* вложенный */
enclose.g (inner* p) { ... }
```

В этом примере x в f ссылается на x, описанный перед классом enclose. Поскольку y является закрытым членом inner, g не может его использовать. Поскольку g является членом enclose, имена, использованные в g, считаются находящимися в области видимости класса enclose. Поэтому inner в описании параметров g относится к охваченному типу inner, а не к int.

8.6 Инициализация

Описание может задавать начальное значение описываемого идентификатора. Инициализатору предшествует =, и он состоит из выражения или списка значений, заключенного в фигурные скобки.

```
инициализатор:
     =                                              expression
     =          {          список_инициализаторов          }
     =       {         список_инициализаторов       ,       }
     (                   список_выражений                   )
список_инициализаторов                                      :
     выражение
     список_инициализаторов      ,      список_инициализаторов
     { список_инициализаторов }
```

Все выражения в инициализаторе статической или внешней переменной должны быть константными выражениями, которые описаны в #15, или выражениями, которые сводятся к адресам ранее описанных переменных, возможно со смещением на константное выражение. Автоматические и регистровые

переменные могут инициализироваться любыми выражениями, включающими константы, ранее описанные переменные и функции.

Гарантируется, что неинициализированные статические и внешние переменные получают в качестве начального значения "пустое значение"[*]. Когда инициализатор применяется к скаляру (указатель или объект арифметического типа), он состоит из одного выражения, возможно, заключенного в фигурные скобки. Начальное значение объекта находится из выражения; выполняются те же преобразования, что и при присваивании.

Заметьте, что поскольку () не является инициализатором, то "X a();" является не описанием объекта класса X, а описанием функции, не получающей значений и возвращающей X.

8.6.1 Список инициализаторов

Когда описанная переменная является составной (класс или массив), то инициализатор может состоять из заключенного в фигурные скобки, разделенного запятыми списка инициализаторов для членов составного объекта, в порядке возрастания индекса или по порядку членов. Если массив содержит составные подобъекты, то это правило рекурсивно применяется к членам составного подобъекта. Если инициализаторов в списке меньше, чем членов в составном подобъекте, то составной подобъект дополняется нулями.

Фигурные скобки могут опускаться следующим образом. Если инициализатор начинается с левой фигурной скобки, то следующий за ней список инициализаторов инициализирует члены составного объекта; наличие числа инициализаторов, большего, чем число членов, считается ошибочным. Если, однако, инициализатор не начинается с левой фигурной скобки, то из списка берутся только элементы, достаточные для сопоставления членам составного объекта, частью которого является текущий составной объект.

Например,

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

описывает и инициализирует x как одномерный массив, имеющий три члена, поскольку размер не был указан и дано три инициализатора.

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

является полностью снабженной квадратными скобками инициализацией: 1,3 и 5 инициализируют первый ряд массива y[0], а именно, y[0][2]. Аналогично, следующие две строки инициализируют y[1] и y[2]. Инициализатор заканчивается раньше, поэтому y[3] инициализируется значением 0. В точности тот же эффект может быть достигнут с помощью

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

Инициализатор для y начинается с левой фигурной скобки, но не начинается с нее инициализатор для y[0], поэтому используется три значения из списка. Аналогично, следующие три успешно используются для y[1] и следующие три для y[2].

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

инициализирует первый столбец y (рассматриваемого как двумерный массив) и оставляет остальные элементы нулями.

8.6.2 Классовые объекты

Объект с закрытыми членами не может быть инициализирован с помощью простого присваивания, как это описывалось выше; это же относится к объекту объединение. Если класс имеет конструктор, не получающий значений, то этот конструктор используется для объектов, которые явно не инициализированы.

Параметры для конструктора могут также быть представлены в виде заключенного в круглые скобки списка. Например:

```
struct complex
  {
    float re;
    float im;
    complex (float r,float i) { re=r; im=i; }
    complex (float r) { re=r; im=0; }
  };
complex zz (1,2.3);
complex* zp = new complex (1,2.3);
```

Инициализация может быть также выполнена с помощью явного присваивания; преобразования производятся. Например,

```
complex zz1 = complex (1,2.3);
complex zz2 = complex (123);
complex zz3 = 123;
complex zz4 = zz3;
```

Если конструктор ссылается на объект своего собственного класса, то он будет вызываться при инициализации объекта другим объектом этого класса, но не при инициализации объекта конструктором.

Объект класса, имеющего конструкторы, может быть членом составного объекта только если он сам не имеет конструктора или если его конструкторы не имеют параметров. В последнем случае конструктор вызывается при создании составного объекта. Если член составного объекта является членом класса с деструкторами, то этот деструктор вызывается при уничтожении составного объекта.

8.6.3 Ссылки

Когда переменная описана как T&, что есть "ссылка на тип T", она может быть инициализирована или указателем на тип T, или объектом типа T. В последнем случае будет неявно применена операция взятия адреса &. Например:

```
int i;
int& r1 = i;
int& r2 = &i;
```

И r1 и r2 будут указывать на i.

Обработка инициализации ссылки очень сильно зависит от того, что ей присваивается. Как описывалось в #7.1, ссылка неявно переадресуется при ее использовании. Например

```
r1 = r2;
```

означает копирование целого, на которое указывает r2, в целое, на которое указывает r1.

Ссылка должна быть инициализирована. Таким образом, ссылку можно считать именем объекта.

Чтобы получить указатель pp, обозначающий тот объект, что и ссылка rr, можно написать pp=&rr. Это будет проинтерпретировано как pp=&*rr.

Если инициализатор для ссылки на тип T не является адресным выражением, то будет создан и инициализирован с помощью правил инициализации объект типа T. Тогда значением ссылки станет адрес объекта. Время жизни объекта, созданного таким способом, будет в той области видимости, в которой он создан. Например:

```
double& rr = 1;
```
допустимо, и rr будет указывать на объект типа double, в котором хранится значение 1.0.

Ссылки особенно полезны в качестве типов параметров.

8.6.4 Массивы символов

Последняя сокращенная запись позволяет инициализировать строкой массив данных типа char. В этом случае последовательные символы строки инициализируют члены массива. Например:

```
char msg[] = "Syntax error on line %d\n";
```
демонстрирует массив символов, члены которого инициализированы строкой.

8.7 Имена типов

Иногда (для неявного задания преобразования типов и в качестве параметра sizeof или new) нужно использовать имя типа данных. Это выполняется при помощи "имени типа" которое по сути является описанием для объекта этого типа, в котором опущено имя объекта.

```
имя_типа:
     спецификатор_типа абстрактный_описатель
абстрактный_описатель                                          :
     пустой
     *                                    абстрактный_описатель
     абстрактный_описатель    (  списоко_писателей_параметров)
     абстрактный_описатель    [  константное_выражение  opt  ]
     ( абстрактный_описатель )
```

Является возможным идентифицировать положение в абстрактном_описателе, где должен был бы появляться идентификатор в случае, если бы конструкция была описателем в описании. Тогда именованный тип является тем же, что и тип предполагаемого идентификатора. Например:

```
int
int *
int *[3]
int *()
int (*)()
```
именует, соответственно, типы "целое", "указатель на целое", "указатель на массив из трех целых", "функция, возвращающая указатель на функцию, возвращающую целое" и "указатель на целое".

Простое имя типа есть имя типа, состоящее из одного идентификатора или ключевого слова.

```
простое_имя_типа:
     typedef-имя
     char
     short
     int
     long
     unsigned
     float
     double
```

Они используются в альтернативном синтаксисе для преобразования типов. Например:

```
(double) a
```
может быть также записано как
```
double (a)
```

8.8 Определение типа typedef

Описания, содержащие спецификатор_описания typedef, определяют идентификаторы, которые позднее могут использоваться так, как если бы они были ключевыми словами типа, именующее основные или производные типы.

```
typedef-имя:
     идентификатор
```

Внутри области видимости описания, содержащего typedef, каждый идентификатор, возникающий как часть какого-либо описателя, становится в этом месте синтаксически эквивалентным ключевому слову типа, которое именует тип, ассоциированный с идентификатором таким образом, как описывается в #8.4. Имя класса или перечисления также является typedef-именем. Например, после

```
typedef int MILES, *KLICKSP;
struct complex { double re, im; };
```

каждая из конструкций

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

является допустимым описанием; distance имеет тип int, metricp имеет тип "указатель на int".

typedef не вводит новых типов, но только синонимы для типов, которые могли бы быть определены другим путем. Так в приведенном выше примере distance рассматривается как имеющая в точности тот же тип, что и любой другой int объект.

Но описание класса вводит новый тип. Например:

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

описывает три переменных трех различных типов.

Описание вида

```
описание_имени:
     агрег                    идентификатор                    ;
     enum  идентификатор ;
```

определяет то, что идентификатор является именем некоторого (возможно, еще не определенного) класса или перечисления. Такие описания позволяют описывать классы, ссылающихся друг на друга. Например:

```
class vector;
class matrix
  {
    ...
    friend matrix operator* (matrix&,vector&);
  };
class vector
  {
    ...
    friend matrix operator* (matrix&,vector&);
  };
```

8.9 Перегруженные имена функций

В тех случаях, когда для одного имени определено несколько (различных) описаний функций, это имя называется перегруженным. При использовании этого имени правильная функция выбирается с помощью сравнения типов фактических параметров с типами параметров в описаниях функций. К перегруженным именам неприменима операция получения адреса &.

Из обычных арифметических преобразований, определенных в #6.6, для вызова перегруженной функции выполняются только char->short->int, int->double, int->long и float->double. Для того, чтобы перегрузить имя функции не-члена описание overload должно предшествовать любому описанию функции; см. #8.2.

Например:

```
overload abs;
int abs (int);
double abs (double);
```

Когда вызывается перегруженное имя, по порядку производится сканирование списка функций для нахождения той, которая может быть вызвана. Например, abs(12) вызывает abs(int), а abs(12.0) будет вызывать abs(double). Если бы был зарезервирован порядок вызова, то оба обращения вызвали бы abs(double).

Если в случае вызова перегруженного имени с помощью вышеуказанного метода не найдено ни одной функции, и если функция получает параметр типа класса, то конструкторы классов параметров (в этом случае существует единственный набор преобразований, делающий вызов допустимым) применяются неявным образом. Например:

```
class X { ... X (int); };
class Y { ... Y (int); };
class Z { ... Z (char*); };
overload int f (X), f (Y);
overload int g (X), g (Y);
f (1);        /* неверно: неоднозначность f(X(1)) или f(Y(1)) */
g (1);        /* g(X(1))                                      */
g ("asdf");   /* g(Z("asdf"))                                 */
```

Все имена функций операций являются автоматически перегруженными.

8.10 Описание перечисления

Перечисления являются int с именованными константами.

```
enum_спецификатор:
     enum    идентификатор     opt     {     enum_список     }
enum_список:
     перечислитель
     enum_список,                                  перечислитель
перечислитель:
     идентификатор
     идентификатор = константное_выражение
```

Идентификаторы в enum-списке описаны как константы и могут появляться во всех местах, где требуются константы. Если не появляется ни одного перечислителя с =, то значения всех соответствующих констант начинаются с 0 и возрастают на 1 по мере чтения описания слева направо. Перечислитель с = дает ассоциированному с ним идентификатору указанное значение; последующие идентификаторы продолжают прогрессию от присвоеннного значения.

Имена перечислителей должны быть отличными от имен обычных переменных. Значения перечислителей не обязательно должны быть различными.

Роль идентификатора в спецификаторе перечисления enum_спецификатор полностью аналогична роли имени класса; он именует определенный нутератор. Например:

```
enum color { chartreuse, burgundy, claret=20, winedark };
  ...
  color *cp, col;
  ...
```

```
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```
делает color именем типа, описывающего различные цвета, и затем описывает cp как указатель на объект этого типа. Возможные значения извлекаются из множества { 0, 1, 20, 21 }.

8.11 Описание Asm

Описание Asm имеет вид <ырге> asm (строка);

Смысл описания asm не определен. Обычно оно используется для передачи информации ассемблеру через компилятор.

9. Операторы

Операторы выполняются последовательно во всех случаях кроме особо оговоренных.

9.1 Оператор выражение

Большинство операторов является операторами выражение, которые имеют вид

```
выражение ;
```

Обычно операторы выражение являются присваиваниями и вызовами функций.

9.2 Составной оператор, или блок

Составной оператор (называемый также "блок", что эквивалентно) дает возможность использовать несколько операторов в том месте, где предполагается использование одного:

```
составной_оператор:
     { список_описаний opt список_операторов opt }
список_описаний:
     описание
     описание список_описаний
список_операторов:
     оператор
     оператор список_операторов
```

Если какой-либо из идентификаторов в списке_описаний был ранее описан, то внешнее описание выталкивается на время выполнения блока, и снова входит в силу по его окончании.

Каждая инициализация auto или register переменных производится всякий раз при входе в голову блока. В блок делать передачу; в этом случае инициализации не выполняются. Инициализации переменных, имеющих класс памяти static (#4.2) осуществляются только один раз в начале выполнения программы.

9.3 Условный оператор

Есть два вида условных операторов

```
if ( выражение ) оператор
if ( выражение ) оператор else оператор
```

В обоих случаях вычисляется выражение, и если оно не ноль, то выполняется первый подоператор. Во втором случае второй подоператор выполняется, если выражение есть 0. Как обычно, неоднозначность "else" разрешается посредством того, что else связывается с последним встреченным if, не имеющим else.

9.4 Оператор while

Оператор while имеет вид

```
while ( выражение ) оператор
```

Выполнение подоператора повторяется, пока значение выражения остается ненулевым. Проверка выполняется перед каждым выполнением оператора.

9.5 Оператор do

Оператор do имеет вид

```
do оператор while (выражение);
```

Выполнение подоператора повторяется до тех пор, пока значение выражения не станет нулем. Проверка выполняется после каждого выполнения оператора.

9.6 Оператор for

Оператор for имеет вид

```
for (  выражение_1 opt  ; выражение_2  opt ; выражение_3 opt )
     оператор
```

Этот оператор эквивалентен следующему:

```
выражение_1;
while                                                (выражение_2)
                                                                {
                                                          оператор
                                                       выражение_3;
  }
```

Первое выражение задает инициализацию цикла; второе выражение задает осуществляемую перед каждой итерацией проверку, по которой производится выход из цикла, если выражение становится нулем; третье выражение часто задает приращение, выполняемое после каждой итерации.

Каждое или все выражения могут быть опущены. Отсутствие выражения_2 делает подразумеваемое while-предложение эквивалентным while(1); остальные опущенные выражения просто пропускаются в описанном выше расширении.

9.7 Оператор switch

Оператор switch вызывает передачу управления на один из нескольких операторов в зависимости от значения выражения. Он имеет вид

```
switch ( выражение ) оператор
```

Выражение должно быть целого типа или типа указателя. Любой оператор внутри оператора может быть помечен одним или более префиксом case следующим образом:

```
case      константное_выражение :
```

где константное выражение должно иметь тот же тип что и выражение- переключатель; производятся обычные арифметические преобразования. В одном операторе switch никакие две константы, помеченные case, не могут иметь одинаковое значение. Константные выражения точно определяются в #15.

Может также быть не более чем один префикс оператора вида

```
default :
```

Когда выполнен оператор switch, проведено вычисление его выражения и сравнение его с каждой case константой. Если одна из констант равна значению выражения, то управление передается на выражение,

следующее за подошедшим префиксом case. Если никакая case константа не соответствует выражению, и есть префикс default, то управление передается на выражение, которому он предшествует. Если нет соответствующих вариантов case и default отсутствует, то никакой из операторов в операторе switch не выполняется.

Префиксы case и default сами по себе не изменяют поток управления, который после задержки идет дальше, перескакивая через эти префиксы. Для выхода из switch см. break, #9.8.

Обычно зависящий от switch оператор является составным. В голове этого оператора могут стоять описания, но инициализации автоматических и регистровых переменных являются безрезультатными.

9.8 Оператор break

Оператор

```
break ;
```

прекращает выполнение ближайшего охватывающего while, do, for или switch оператора; управление передается на оператор, следующий за законченным.

9.9 Оператор continue

Оператор

```
continue ;
```

вызывает передачу управления на управляющую продолжением цикла часть наименьшего охватывающего оператора while, do или for; то есть на конец петли цикла. Точнее, в каждом из операторов

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

continue эквивалентно goto contin. (За contin: следует пустой оператор, #9.13.)

9.10 Оператор return

Возврат из функции в вызывающую программу осуществляется с помощью оператора return, имеющего один из двух видов:

```
return ;
return выражение ;
```

Первый может использоваться только в функциях, не возвращающих значения, т.е. в функциях с типом возвращаемого значения void. Вторая форма может использоваться только в функциях, не возвращающих значение; вызывающей функцию программе возвращается значение выражения. Если необходимо, то выражение преобразуется, как это делается при присваивании, к типу функции, в которой оно возникло. Обход конца функции эквивалентен возврату return без возвращаемого значения.

9.11 Оператор goto

Можно осуществлять безусловную передачу управления с помощью оператора

```
goto идентификатор ;
```

Идентификатор должен быть меткой (#9.12), расположенной в текущей функции.

9.12 Помеченные операторы

Перед любым оператором может стоять префикс метка, имеющий вид

```
идентификатор :
```
который служит для описания идентификатора как метки. Метка используется только как объект для goto. Областью видимости метки является текущая функция, исключая любой подблок, в котором был переописан такой же идентификатор. См. #4.1.

9.13 Пустой оператор

Пустой оператор имеет вид

```
;
```

Пустой оператор используется для помещения метки непосредственно перед } составного оператора или того, чтобы снабдить такие операторы, как while, пустым телом.

9.14 Оператор delete

Оператор delete имеет вид

```
delete выражение ;
```

Результатом выражения должен быть указатель. Объект, на который он указывает, уничтожается. Это значит, что после оператора уничтожения delete нельзя гарантировать, что объект имеет определенное значение; см. #17. Эффект от применения delete к указателю, не полученному из операции new (#7.1), не определен. Однако, уничтожение указателя с нулевым значением безопасно.

9.15 Оператор asm

Оператор asm имеет вид

```
asm ( строка) ;
```

Смысл оператора asm не определен. Обычно он используется для передачи информации через компилятор ассемблеру.

10. Внешние определения

Программа на C++ состоит из последовательности внешних определений. Внешнее определение описывает идентификатор как имеющий класс памяти static и определяет его тип. Спецификатор типа (#8.2) может также быть пустым, и в этом случае принимается тип int. Область видимости внешних определений простирается до конца файла, в котором они описаны, так же, как действие описаний сохраняется до конца блока. Синтаксис внешних определений тот же, что и у описаний, за исключением того, что только на этом уровне и внутри описаний классов может быть задан код (текст программы) функции.

10.1 Определения функций

Определения функций имеют вид

```
определение_функции:
     спецификаторы_описания описатель_функции  opt инициализа-
тор_базового_класса                                        opt
     тело_функции
```

Единственными спецификаторами класса памяти (sc-спецификаторами), допустимыми среди спецификаторов описания, являются extern, static, overload, inline и virtual. Описатель функции похож на описатель "функции, возвращающей ...", за исключением того, что он включает в себя имена формальных параметров определяемой функции. Описатель функции имеет вид

```
описатель_функции:
     описатель ( список_описаний_параметров )
```

Форма списка описаний параметров определена в #8.4. Единственный класс памяти, который может быть задан, это тот, при котором соответствующий фактический параметр будет скопирован, если это возможно, в регистр при входе в функцию. Если в качестве инициализатора для параметра задано константное выражение, то это значение используется как значение параметра по умолчанию.

Тело функции имеет вид

```
тело_функции:
     составной_оператор
```

Вот простой пример полного определения функции:

```
int max (int a,int b,int c)
  {
    int m = (a > b) ? a : b;
    return  (m > c) ? m : c;
  }
```

Здесь int является спецификатором типа ; max (int a, int b, int c) является описателем функции ; { ... } - блок, задающий текст программы (код) оператора.

Поскольку в контексте выражения имя (точнее, имя как формальный параметр) считается означающим указатель на первый элемент массива, то описания формальных параметров, описанных как "массив из ...", корректируются так, чтобы читалось "указатель на ...".

Инициализатор базового класса имеет вид

```
инициализатор_базового_класса:
     : ( список_параметров opt )
```

Он используется для задания параметров конструктора базового класса в конструкторе производного класса. Например:

```
struct base { base (int); ... };
struct derived : base { derived (int); ... };
derived.derived (int a) : (a+1) { ... }
derived d (10);
```

Конструктор базового класса вызывается для объекта d с параметром 11.

10.2 Определения внешних данных

Определения внешних данных имеют вид

```
определение_данных:
     описание
```

Класс памяти таких данных статический.

Если есть более одного определения внешних данных одного имени, то определения должны точно согласовываться по типу и классу памяти, и инициализаторы (если они есть), должны иметь одинаковое значение.

11. Правила области видимости

См. #4.1.

12. Командные строки компилятора

Компилятор языка C++ содержит препроцессор, способный выполнять макроподстановки, условную компиляцию и включение именованных файлов. Строки, начинающиеся с #, относятся к препроцессору. Эти строки имеют независимый от остального языка синтаксис; они могут появляться в любом месте оказывать влияние, которое распространяется (независимо от области видимости) до конца исходного файла программы.

Заметьте, что определения const и inline дают альтернативы для большинства использований #define.

12.1 Замена идентификаторов

Командная строка компилятора имеет вид

```
#define идент строка_символов
```

вызывает замену препроцессором последующих вхождений идентификатора, заданного строкой символов. Точка с запятой внутри (или в конце) строки символов является частью этой строки.

Строка вида

```
#define идент( идент , ..., идент ) строка_символов
```

где отсутсвует пробел между первым идентификатором и (, является макроопределением с параметрами. Последующие вхождения первого идентификатора с идущими за ним (, последовательностью символов, разграниченной запятыми, и), заменяются строкой символов, заданной в определении. Каждое местоположение идентификатора, замеченного в списке параметров определения, заменяется соответствующей строкой из вызова. Фактическими параметрами вызова являются строки символов, разделенные запятыми; однако запятые в строке, заключенной в кавычки, или в круглых скобках не являются разделителями параметров. Число формальных и фактических параметров должно совпадать. Строки и символьные константы в символьной строке сканируются в поисках формальных параметров, но строки и символьные константы в остальной программе не сканируются в поисках определенных (с помощью define) идентификаторов.

В обоих случаях строка замещения еще раз сканируется в поисках других определенных идентификаторов. В обоих случаях длинное определение может быть продолжено на другой строке с помощью записи \ в конце продолжаемой строки.

Командная строка вида

```
#undef идент
```

влечет отмену препроцессорного определения идентификатора.

12.2 Включение файлов

Командная строка компилятора вида

```
#include "имя_файла"
```

вызывает замену этой строки полным содержимым файла имя_файла. Сначала именованный файл ищется в директории первоначального исходного файла, а затем в стандартных или заданных местах. Альтернативный вариант, командная строка вида

```
#include <имя_файла>
```

производит поиск только в стандартном или заданном месте, и не ищет в директории первоначального исходного файла. (То, как эти места задаются, не является частью языка.)

Включения с помощью #include могут быть вложенными.

12.3 Условная компиляция

Командная строка компилятора вида

```
#if выражение
```

проверяет, является ли результатом вычисления выражения не-ноль. Выражение должно быть константным выражением, которые обсуждаются в #15; применительно к использованию данной директивы есть дополнительные ограничения: константное выражение не может содержать sizeof или перечислимые константы. Кроме обычных операций С может использоваться унарная операция defined. В случае применения к идентификатору она дает значение не-ноль, если этот идентификатор был ранее определен с помощью #define и после этого не было отмены определения с помощью #undef; иначе ее значение 0.

Командная строка вида

```
#ifdef идент
```

проверяет, определен ли идентификатор в препроцессоре в данный момент; то есть, был ли он объектом командной строки #define.

Командная строка вида

```
#ifndef идент
```

проверяет, является ли идентификатор неопределенным в препроцессоре в данный момент.

После строки каждого из трех видов может стоять произвольное количество строк, возможно, содержащих командную строку

```
#else
```

и далее до командной строки

```
#endif
```

Если проверенное условие истинно, то все строки между #else и #endif игнорируются. Если проверенное условие ложно, то все строки между проверкой и #else или, в случае отсутствия #else, #endif, игнорируются.

Эти конструкции могут быть вложенными.

12.4 Управление строкой

Для помощи другим препроцессорам, генерирующим программы на C, строка вида

```
#line константа "имя_файла"
```

заставляет компилятор считать, например, в целях диагностики ошибок, что константа задает номер следующей строки исходного файла, и текущий входной файл именуется идентификатором. Если идентификатор отсутствует, то запомненное имя файла не изменяется.

13. Неявные описания

См. #8.1.

14. Обзор типов

В этом разделе кратко собрано описание действий, которые могут совершаться над объектами различных типов.

14.1 Классы

Классовые объекты могут присваиваться, передаваться функциям как параметры и возвращаться функциями. Другие возможные операции, как, например, проверка равенства, могут быть определены пользователем; см. #8.5.10.

14.2 Функции

Есть только две вещи, которые можно проделывать с функцией: вызывать ее и брать ее адрес. Если в выражении имя функции возникает не в положении имени функции в вызове, то генерируется указатель на функцию. Так, для передачи одной функции другой можно написать

```
typedef int (*PF) ();
extern g (PF);
extern f ();
...
```

```
g (f);
```
Тогда определение g может иметь следующий вид:
```
g (PF funcp)
  {
    ...
    (*funcp) ();
    ...
  }
```

Заметьте, что f должна быть описана явно в вызывающей программе, поскольку ее появление в g(f) не сопровождалось (.

14.3 Массивы, указатели и индексирование

Всякий раз, когда в выражении появляется идентификатор типа массива, он преобразуется в указатель на первый член массива. Из-за преобразований массивы не являются адресами. По определению операция индексирования [] интерпретируется таким образом, что E1[E2] идентично *((E1)+(E2)). В силу правил преобразования, применяемых к +, если E1 массив и E2 целое, то E1[E2] относится к E2-ому члену E1. Поэтому, несмотря на такое проявление асимметрии, индексирование является коммутативной операцией.

Это правило сообразным образом применяется в случае многомерного массива. Если E является n-мерным массивом ранга i*j*...*k, то возникающее в выражении E преобразуется в указатель на (n-1)-мерный массив ранга j*...*k. Если к этому указателю, явно или неявно, как результат индексирования, применяется операция *, ее результатом является (n-1)-мерный массив, на который указывалось, который сам тут же преобразуется в указатель.

Рассмотрим, например,

```
int x[3][5];
```

Здесь x - массив целых размером 3*5. Когда x возникает в выражении, он преобразуется в указатель на (первый из трех) массив из 5 целых. В выражении x[i], которое эквивалентно *(x+1), x сначала преобразуется, как описано, в указатель, затем 1 преобразуется к типу x, что включает в себя умножение 1 на длину объекта, на который указывает указатель, а именно объект из 5 целых. Результаты складываются, и используется косвенная адресация для получения массива (из 5 целых), который в свою очередь преобразуется в указатель на первое из целых. Если есть еще один индекс, снова используется тот же параметр; на этот раз результат является целым.

Именно из всего этого проистекает то, что массивы в C хранятся по строкам (быстрее всего изменяется последний индекс), и что в описании первый индекс помогает определить объем памяти, поглощаемый массивом, но не играет никакой другой роли в вычислениях индекса.

14.4 Явные преобразования указателей

Определенные преобразования, включающие массивы, выполняются, но имеют зависящие от реализации аспекты. Все они задаются с помощью явной операции преобразования типов, см. #7.2 и #8.7.

Указатель иожет быть преобразован к любому из целых типов, достаточно больших для его хранения. То, какой из int и long требуется, является машинно-зависимым. Преобразующая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине. Подробности для некоторых конкретных машин были даны в #2.6.

Объект целого типа может быть явно преобразован в указатель. Преобразующая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно-зависимой.

Указатель на один тип может быть преобразован в указатель на другой тип. Использование результирующего указателя может вызывать особые ситуации, если исходный указатель не указывает на объект, соответствующим образом выравненный в памяти. Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений.

Например, программа, выделяющая память, может получать размер (в байтах) размещаемого объекта и возвращать указатель на char; это можно использовать следующим образом.

```
extern void* alloc ();
double* dp;
dp = (double*) alloc (sizeof (double));
*dp= 22.0 / 7.0;
```

alloc должна обеспечивать (машинно-зависимым образом) то, что возвращаемое ею значение подходит для преобразования в указатель на double; в этом случае использование функции мобильно. Различные машины различаются по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

15. Константные выражения

В нескольких местах C++ требует выражения, вычисление которых дает константу: в качестве границы массива (#8.3), в case выражениях (#9.7), в качестве значений параметров функции, присваиваемых по умолчанию, (#8.3), и в инициализаторах (#8.6). В первом случае выражение может включать только целые константы, символьные константы, константы, описанные как имена, и sizeof выражения, возможно, связанные бинарными операциями

```
+ - * / % & | ^ << >> == != < > <= >= && ||
```

или унарными операциями

```
- ~ !
```

или тернарными операциями

```
? :
```

Скобки могут использоваться для группирования, но не для вызова функций.

Большая широта допустима для остальных трех случаев использования; помимо константных выражений, обсуждавшихся выше, допускаются константы с плавающей точкой, и можно также применять унарную операцию & к внешним или статическим объектам, или к внешним или статическим массивам, индексированным константным выражением. Унарная операция & может также быть применена неявно с помощью употребления неиндексированных массивов и функций. Основное правило состоит в том, что инициализаторы должны при вычислении давать константу или адрес ранее описанного внешнего или статического объекта плюс или минус константа.

Меньшая широта допустима для константных выражений после #if: константы, описанные как имена, sizeof выражения и перечислимые константы недопустимы.

16. Соображения мобильности

Определенные части C++ являются машинно-зависимыми по своей сути. Следующий ниже список мест возможных затруднений не претендует на полноту, но может указать на основные из них.

Как показала практика, характеристики аппаратуры в чистом виде, такие, как размер слова, свойства плавающей арифметики и целого деления, не создают особых проблем. Другие аппаратные аспекты отражаются на различных программных разработках. Некоторые из них, особенно знаковое расширение (преобразование отрицательного символа в отрицательное целое) и порядок расположения байтов в слове, являются досадными помехами, за которыми надо тщательно следить. Большинство других являются всего лишь мелкими сложностями.

Число регистровых переменных, которые фактически могут быть помещены в регистры, различается от машины к машине, как и множество фактических типов. Тем не менее, все компиляторы на "своей" машине все делают правильно; избыточные или недействующие описания register игнорируются.

Некоторые сложности возникают при использовании двусмысленной манеры программирования. Писать программы, зависящие от какой-либо из этих особенностей, районе неблагоразумно.

В языке не определен порядок вычисления параметров функции. На некоторых машинах он слева направо, а на некоторых справа налево. Порядок появления некоторых побочных эффектов также недетерминирован.

Поскольку символьные константы в действительности являются объектами типа int, то могут быть допустимы многосимвольные константы. Однако конкретная реализация очень сильно зависит от машины, поскольку порядок, в котором символы присваиваются слову, различается от машины к машине. На некоторых машинах поля в слове присваиваются слева направо, на других справа налево.

Эти различия невидны для отдельных программ, не позволяющих себе каламбуров с типами (например, преобразования int указателя в char указатель и просмотр памяти, на которую указывает указатель), но должны приниматься во внимание при согласовании внешне предписанных форматов памяти

17. Свободная память

Операция new (#7.2) вызывает функцию

```
extern void* _new (long);
```

для получения памяти. Параметр задает число требуемых байтов. Память будет инициализирована. Если _new не может найти требуемое количество памяти, то она возвращает ноль.

Операция delete вызывает функцию

```
extern void _delete (void*);
```

чтобы освободить память, указанную указателем, для повторного использования. Результат вызова _delete() для указателя, который не был получен из _new(), не определен, это же относится и к повторному вызову _delete() для одного и того же указателя. Однако уничтожение с помощью delete указателя со значением ноль безвредно.

Предоставляются стандартные версии _new() и _delete(), но пользователь может применять другие, более подходящие для конкретных приложений.

Когда с помощью операции new создается классовый объект, то для получения необходимой памяти конструктор будет (неявно) использовать new. Конструктор может осуществить свое собственное резервирование памяти посредством присваивания указателю this до каких-либо использований. С помощью присваивания this значения ноль деструктор может избежать стандартной операции дерезервирования памяти для объекта его класса. Например:

```
class cl
  {
    int v[10];
    cl () { this = my_own_allocator (sizeof (cl)); }
    ~cl () { my_own_deallocator (this); this = 0; }
  }
```

На входе в конструктор this является не-нулем, если резервирование памяти уже имело место (как это имеет место для автоматических объектов), и нулем в остальных случаях.

Если производный класс осуществляет присваивание this, то вызов конструктора (если он есть) базового класса будет иметь место после присваивания, так что конструктор базового класса ссылаться на объект посредством конструктора производного класса. Если конструктор базового класса осуществляет присваивание this, то значение также будет использоваться конструктором (если таковой есть) производного класса.

18. Краткое изложение синтаксиса

Мы надеемся, что эта краткая сводка синтаксиса C++ поможет пониманию. Она не является точным изложением языка.

18.1 Выражения

```
выражение:
  терм
  выражение            бинарная_операция            выражение
```

```
        выражение       ?       выражение       :       выражение
        список_выражений
    терм:
        первичный
        *                                                       терм
        &                                                       терм
        -                                                       терм
        !                                                       терм
        ~                                                       терм
        ++терм
        --терм
        терм++
        терм--
        (                   имя_типа)                   выражение
        имя_простого_типа           (           список_выражений)
        sizeof                                          выражение
        sizeof              (               имя_типа            )
        new                                              имя_типа
        new ( имя_типа )
    первичный:
        id
        ::                                          идентификатор
        константа
        строка
        this
        (                       выражение                       )
        первичный[                  выражение                   ]
        первичный       (       список_выражений       opt      )
        первичный.id
        первичный->id
    id:
        идентификатор
        typedef-имя :: идентификатор
    список_выражений:
        выражение
        список_выражений, выражение
    операция:
        унарная_операция
        бинарная_операция
        специальная_операция
```

Бинарные операции имеют приоритет, убывающий в указанном порядке:

```
    бинарная_операция:
        *                           /                           %
        +                                                       -
        <<                                                      >>
        <                                                       >
        ==                                                      !=
        &
        ^
        |
        &&
        ||
        =   +=  -=  *=  /=  %=  ^=  &=  |=  >>=  <<=
    унарная_операция:
          *   &   -   ~   !   ++   --
    специальная_операция:
        ()      []
    имя_типа:
        спецификаторы_описания абстрактный_описатель
    абстрактный_описатель:
        пустой
        *                                   абстрактный_описатель
        абстрактный_описатель  (   список_описаний_параметров   )
        абстрактный_описатель  [   константное_выражение  opt   ]
```

```
     ( абстрактный_описатель )
простое_имя_типа:
     typedef-имя
     char
     short
     int
     long
     unsigned
     float
     double
typedef-имя:
     идентификатор
```

18.2 Описания

```
описание:
     спецификаторы_описания  opt   список_описателей   opt   ;
     описание_имени
     asm-описание
описание_имени:
     агрег                   идентификатор                 ;
     enum  идентификатор ;
агрег:
     class
     struct
     union
asm-описание:
     asm ( строка );
спецификаторы_описания:
     спецификатор_описания спецификатор_описания opt
спецификатор_описания:
     имя_простого_типа
     спецификатор_класса
     enum_спецификатор
     sc_спецификатор
     фнк_спецификатор
     typedef
     friend
     const
     void
sc_спецификатор:
     auto
     extern
     register
     static
фнк-спецификатор:
     inline
     overload
     virtual
список_описателей:
     иниц-описатель
     иниц-описатель , список_описателей
иниц-описатель:
     описатель инициализатор opt
описатель:
     оп_имя
     (                       описатель                       )
     *             const             opt             описатель
     &             const             opt             описатель
     описатель      (       список_описаний_параметров       )
     описатель [ константное_выражение opt ]
оп_имя:
     простое_оп_имя
```

```
     typedef-имя . простое_оп_имя
простое_оп_имя:
     идентификатор
     typedef-имя
     -                                                  typedef-имя
     имя_функции_операции
имя_функции_операции:
     операция операция
список_описаний_параметров:
     список_описаний_прм opt ... opt
список_описаний_прм                                             :
     список_описаний_прм         ,          описание_параметра
     описание_параметра
описание_параметра:
     спецификаторы_описания                              описатель
     спецификаторы_описания описатель = константное_выражение
спецификатор_класса:
     заголовок_класса       {список_членов        opt        }
     заголовок_класса    {список_членов     opt    public    :
список_членов opt }
заголовок_класса                                                :
     агрег                   идентификатор                   opt
     агрег идентификатор opt : public opt typedef-имя
список_членов                                                   :
     описание_члена список_членов opt
описание_члена:
     спецификаторы_описания opt описатель_члена ;
описатель_члена:
     описатель
     идентификатор opt : константное_выражение
инициализатор:
     =                                                  выражение
     =                {                 список_инициализаторов}
     =       {        список_инициализаторов,                  }
     (список_выражений )
список_инициализаторов                                          :
     выражение
     список_инициализаторов        ,    список_инициализаторов
     { список_инициализаторов }
enum-спецификатор:
     enum идентификатор opt { enum-список }
enum-список:
     перечислитель
     enum-список , перечислитель
перечислитель:
     идентификатор
     идентификатор = константное_выражение
```

18.3 Операторы

```
составной_оператор:
     { список_описаний opt список_операторов opt }
список_описаний:
     описание
     описание список_описаний
список_операторов:
     оператор
     оператор список_операторов
оператор:
     выражение                                                  ;
     if         (         выражение         )         оператор
     if    (    выражение    )    оператор    else    оператор
     while        (         выражение        )        оператор
```

```
do     оператор     while     (     выражение     )     ;
for (  выражение opt  ; выражение  opt ;  выражение opt )
     оператор
switch        (        выражение        )        оператор
case      константное      выражение      :      оператор
default                    :                    оператор
break;
continue;
return            выражение             opt             ;
goto                  идентификатор                    ;
идентификатор               :                   оператор
delete                    выражение                     ;
asm           (            строка           )           ;
;
```

18.4 Внешние определения

```
программа:
     внешнее_определение
     внешнее_определение программа
внешнее_определение:
     определение_функции
     описание
определение_функции:
     спецификаторы_описания       opt        описатель_функции
инициализатор_базового_класса opt тело_функции
описатель_функции:
     описатель ( список_описаний_параметров)
тело_функции:
     составной_оператор
инициализатор_базового_класса:
     : ( список_параметров opt )
```

18.5 Препроцессор

```
#define идент строка_символов
#define идент( идент,...,идент ) строка символов
#else
#endif
#if выражение
#ifdef идент
#ifndef идент
#include "имя_файла"
#include <имя_файла>
#line константа "имя_файла"
#undef идент
```

19. Отличия от "старого C"

19.1 Расширения

Типы параметров функции могут быть заданы (#8.4) и будут проверяться (#7.1). Могут выполняться преобразования типов.

Для выражений с числами с плавающей точкой может использоваться плавающая арифметика одинарной точности; #6.2.

Имена функций могут быть перегружены; #8.6

Операции могут быть перегружены; #7.16, #8.5.10.

Может осуществляться inline-подстановка функций; #8.1.

Объекты данных могут быть константными (const); #8.3.

Могут быть описаны объекты ссылочного типа; #8.3, #8.6.3

Операции new и delete обеспечивают свободное хранение в памяти; #17.

Класс может обеспечивать скрытые данные (#8.5.8), гарантированную инициализацию (#8.6.2), определяемые пользователем преобразования (#8.5.6), и динамическое задание типов через использование виртуальных функций (#8.5.4).

Имя класса является именем типа; #8.5.

Любой указатель может присваиваться [указателю] void* без приведения типов; #7.14.