С.О. Бочков
Д.М. Субботин

Язык программирования СИ для персонального компьютера

Под общей редакцией канд. техн. наук, доцента П.И.Садчикова

ВВЕДЕНИЕ

Среди современных языков программирования язык Си является одним из наиболее распространенных. Язык Си универсален, однако наиболее эффективно его применение в задачах системного программирования — разработке трансляторов, операционных систем, экранных интерфейсов, инструментальных средств. Язык Си хорошо зарекомендовал себя эффективностью, лаконичностью записи алгоритмов, логической стройностью программ. Во многих случаях программы, написанные на языке Си, сравнимы по скорости с программами, написанными на языке ассемблера; при этом они более наглядны и просты в сопровождении.

Одним из основных достоинств языка Си считается высокая переносимость написанных на нем программ между компьютерами с различной архитектурой, между различными операционными средами. Трансляторы языка Си существуют практически для всех используемых в настоящее время персональных компьютеров.

Язык Си имеет ряд существенных особенностей, которые выделяют его среди других языков программирования. В значительной степени на формирование идеологии языка повлияла цель, которую ставили перед собой его создатели, — обеспечение системного программиста удобным инструментальным языком, который мог бы заменить язык ассемблера. В результате появился язык программирования высокого уровня, обеспечивающий необычайно легкий доступ к аппаратным средствам компьютера. Иногда Си называют языком программирования "среднего" уровня. С одной стороны, как и другие современные языки высокого уровня, язык Си поддерживает полный набор конструкций структурного программирования, модульность, блочную структуру программ, раздельную компиляцию. С другой стороны, язык Си имеет ряд низкоуровневых черт.

Перечислим некоторые особенности языка Си:

В языке Си реализованы некоторые операции низкого уровня (в частности, операции над битами). Некоторые из таких операций напрямую соответствуют машинным командам.

Базовые типы данных языка Си отражают те же объекты, с которыми приходится иметь дело в программе на языке ассемблера, — байты, машинные слова, символы, строки. Несмотря на наличие в языке Си развитых средств построения составных объектов (массивов и структур), в нем практически отсутствуют средства для работы с ними как с единым целым (нельзя, например, сложить две структуры).

Язык Си поддерживает механизм указателей на переменные и функции. Указатель — это переменная, предназначенная для хранения машинного адреса некоторой переменной или функции. Поддерживается арифметика указателей, что позволяет осуществлять непосредственный доступ и работу с адресами памяти практически так же легко, как на языке ассемблера. Использование указателей позволяет создавать высокоэффективные программы, однако требует от программиста особой осторожности.

Как никакой другой язык программирования высокого уровня, язык Си "доверяет" программисту. Даже в таком существенном вопросе, как преобразование типов данных, налагаются лишь незначительные ограничения. Предшественники языка Си — языки BCPL и Би — вообще имели бестиповую структуру. Во многих случаях слабый контроль типов может помочь в повышении эффективности программ, однако программист должен очень хорошо знать используемый язык программирования и четко представлять его идеологию, иначе отладка будет крайне тяжела, а надежность создаваемых программ низка.

Несмотря на эффективность и мощность конструкций языка Си, он относительно мал по объему. В нем отсутствуют встроенные операторы для выполнения ввода-вывода, динамического распределения памяти, управления процессами и т.п., однако в системное окружение языка Си входит библиотека стандартных функций, в которой реализованы подобные действия. Вынос этих функций в библиотеку позволяет отделить особенности архитектуры конкретного компьютера и соглашений операционной системы от реализации языка, сделать программу максимально независимой от деталей реализации операционной среды. В то же время программисты могут пользоваться системными библиотечными программами, чтобы более эффективно использовать особенности конкретных операционных сред.

Язык Си был разработан в США сотрудниками фирмы Bell Laboratories на рубеже 70-х годов. Примерно в то же время он был использован для разработки операционной системы UNIX. Вопросы повышения переносимости играли большую роль с самого начала разработки

как языка Си, так и операционной системы UNIX, поэтому их распространение на новые компьютеры протекало очень быстро.

Первое описание языка Си было дано его авторами, Б. Керниганом и Д. Ритчи (имеется русский перевод — [1]). К сожалению, оно не было строгим и полным и содержало ряд моментов, допускающих неоднозначное толкование. Это привело к тому, что в последующем различные разработчики систем программирования трактовали язык Си по-разному. В течение долгого времени фактическим стандартом языка Си служила его реализация в седьмой версии операционной системы UNIX, однако заявления о совместимости некоторой системы программирования с реализацией языка Си в операционной системе UNIX, как правило, не означали точного соответствия. В настоящее время число только наиболее распространенных реализаций языка Си исчисляется десятками, и все они поддерживают, по существу, разные диалекты языка. Ситуация усугубляется тем, что обычно в документации особенности реализации языка освещаются неполно и нестрого.

Для исправления этой ситуации в 1983 г. при Американском Национальном Институте Стандартов (ANSI) был образован комитет по стандартизации языка Си. В октябре 1986 г. разработанный этим комитетом проект был опубликован для общественного обсуждения. В 1989 г. окончательный вариант проекта был утвержден в качестве стандарта ANSI. Подавляющее большинство используемых в настоящее время реализаций языка Си не поддерживает стандарт ANSI в полном объеме. Тем не менее, отчетливо просматривается тенденция к созданию новых реализаций языка Си, соответствующих стандарту ANSI.

В последние годы язык Си получил в СССР широкое распространение. Вышло немало книг, посвященных ему. Некоторые из них носят учебный характер [6, 8, 10, 11], в других рассматриваются конкретные реализации [4, 8, 13], третьи дают краткую, справочную информацию по языку [2, 3, 5, 9, 14] либо дают его сравнительную характеристику [7, 12].

Кроме того, вышло большое количество книг, посвященных операционной системе UNIX, в которых, как правило, дается представление о языке Си.

Ни одна из перечисленных книг, однако, не содержит полного и точного описания языка Си и не может служить справочным пособием, особенно в трудных ситуациях. Отсутствие книги такого рода привело к тому, что основная масса отечественных пользователей не понимает идеологии и концепций построения языка и предпочитает пользоваться только традиционными, проверенными методами, избегая многих мощных и эффективных конструкций. Такие черты языка, как использование синтаксиса выражений при построении объявлений данных, или концепция области действия переменных и функций, делают его трудным для изучения. Даже квалифицированные программисты зачастую нечетко представляют себе некоторые аспекты языка. Для многих программистов единственным источником знаний по языку Си служит документация на используемую ими систему программирования. Однако не всегда эта документация оказывается полной и понятной; кроме того, она обычно написана на английском языке или является плохим переводом с английского.

Настоящая книга должна, по замыслу авторов, восполнить этот пробел и дать современное, полное и по возможности строгое описание языка Си. Книга ориентирована на широкий круг программистов, однако не является учебником по языку, а носит в основном справочный характер. Предполагается, что читатель имеет опыт работы с современными языками программирования.

В СССР наибольшее распространение на персональных компьютерах типа IBM PC/XT/AT и совместимых с ними получили система программирования фирмы Microsoft версий 4.0 и 5.0 (далее по тексту называемая СП MSC) и система программирования Турбо Си фирмы Borland International версий 1.5 и 2.0 (далее по тексту называемая СП TC). Версия 4.0 СП MSC является наименее мощной реализацией языка Си из вышеперечисленных, поэтому описание базируется на ней. В тех случаях, когда имеются отличия для версии 5.0 СП MSC либо для СП TC, в тексте делаются соответствующие отступления.

Описание библиотечных функций языка Си затрудняется тем, что расхождений в составе библиотек в различных системах программирования (даже в пределах разных версии одной и той же системы программирования) значительно больше, чем отличий в реализации языка. В данной книге описаны стандартные библиотечные функции версии 4.0 СП MSC и версии 1.5 и 2.0 СП TC.

Материал книги организован следующим образом:

В разделе 1 "Элементы языка Си" описываются алфавит, лексические конструкции и правила языка Си.

В разделе 2 "Структура программы" обсуждаются структура и компоненты Си-программы, организация исходных файлов и правила доступа к программным объектам.

В разделе 3 "Объявления" описывается, каким образом объявлять переменные, функции, а также типы, определяемые пользователем. Помимо простых переменных, язык Си позволяет объявлять указатели и составные объекты—массивы, структуры, объединения.

В разделе 4 "Выражения" описываются операнды и операции, а также обсуждаются вопросы преобразования типов и побочные эффекты, которые могут возникнуть при этом.

В разделе 5 "Операторы" описываются управляющие конструкции.

В разделе 6 "Функции" обсуждаются правила построения и вызова модулей, которые в языке Си называются функциями. В частности, в этом разделе объясняется, как определять, объявлять и вызывать функции, как описывать параметры функции и возвращаемые значения.

В разделе 7 "Директивы препроцессора и указания компилятору" описываются директивы, распознаваемые препроцессором языка Си. Препроцессор представляет собой макропроцессор, автоматически вызываемый в качестве нулевого прохода компилятора языка Си.

В разделе 8 описаны модели памяти для процессора с сегментной архитектурой памяти (типа Intel 8086/8088) и правила работы с ними в программах, написанных на языке Си.

Соглашения о нотации

В книге приняты следующие соглашения о нотации:

Обозначение	Смысл
Угловые скобки	Угловые скобки выделяют нетерминальные символы в синтаксических конструкциях. Например, в записи goto <*имя*> <имя> представлено в угловых скобках, чтобы обратить внимание на то, что определяется общая форма оператора перехода goto. В своей программе пользователь подставит конкретный идентификатор вместо нетерминального символа <*имя*>
Квадратные скобки	Квадратные скобки, ограничивающие синтаксическую конструкцию, означают ее необязательность. Например, в операторе возврата return выражение необязательно: return [<*выражение*>];

Многоточие	Многоточие может быть вертикальным или горизонтальным. В следующем примере вертикальные многоточия означают, что нуль или более объявлений может следовать перед одним или более операторами внутри фигурных скобок.
	{
	[<*объявление*>]
	.
	.
	.
	<*оператор*>
	[<*оператор*>]
	.
	.
	.
	}
	Вертикальные многоточия также используются в примерах программ для обозначения части программы, которая пропущена.
	Горизонтальное многоточие, следующее после некоторой синтаксической конструкции, обозначает последовательность конструкций той же самой формы, что и предшествующая многоточию конструкция. Например, запись ={<*выражение*>[,<*выражение*>]…} означает, что одно или более выражений, разделенных запятыми, может появиться между фигурными скобками. В целях экономии места в некоторых случаях вместо вертикальных многоточий используются горизонтальные.

ЧАСТЬ 1 ОПИСАНИЕ ЯЗЫКА СИ

1 ЭЛЕМЕНТЫ ЯЗЫКА СИ

Под элементами языка понимаются его базовые конструкции, используемые при написании программ. В этом разделе описываются следующие элементы языка Си:

- алфавит;
- константы;
- идентификаторы;
- ключевые слова;
- комментарии.

Компилятор языка Си воспринимает исходный файл, содержащий программу на языке Си, как последовательность текстовых строк. Каждая строка завершена символом новой строки. Этот символ вставляется текстовым редактором при нажатии клавиши ENTER (ВВОД).

Компилятор языка Си последовательно считывает строки программы и разбивает каждую из считанных строк на группы символов, называемые лексемами. Лексема—это единица текста программы, которая имеет самостоятельный смысл для компилятора языка Си и которая не содержит в себе других лексем. Никакие лексемы, кроме символьных строк, не могут продолжаться на последующих строках текста программы. Знаки операций, константы, идентификаторы и ключевые слова, описанные в этом разделе, являются примерами лексем. Разделители, например квадратные скобки [], фигурные скобки {}, круглые скобки (), угловые скобки < > и запятые, также являются лексемами. Внутри идентификаторов, ключевых слов, а также знаков операций, состоящих из нескольких символов, пробельные символы недопустимы.

Когда компилятор языка Си выделяет отдельную лексему, он пытается включить в нее последовательно столько символов, сколько возможно, прежде чем перейти к выделению следующей лексемы. Рассмотрим, например, следующее выражение:

```
i+++j
```

В этом примере компилятор языка Си вначале сформирует из первых двух знаков "плюс" операцию инкремента (++), а из оставшегося знака плюс — операцию сложения. Выражение проинтерпретируется как (i++)+(j), а не как (i)+(++j). В подобных случаях рекомендуется для ясности разделять лексемы пробельными символами или круглыми скобками.

1.1 Алфавит

В программах на языке Си используются два множества символов: множество символов языка Си и множество представимых символов. Множество символов языка Си содержит буквы, цифры и знаки пунктуации, которые имеют определенный смысл для компилятора языка Си. Программы на языке Си строятся путем комбинирования в осмысленные синтаксические конструкции символов из множества символов языка Си.

Множество символов языка Си является подмножеством множества представимых символов. Множество представимых символов состоит из всех букв, цифр и символов, которые могут быть представлены как отдельный символ на клавиатуре данного персонального компьютера.

Программа на языке Си может содержать только символы из множества символов языка Си, однако внутри символьных строк, символьных констант и комментариев может быть использован любой представимый символ. Компилятор языка Си выдает сообщение об ошибке при обнаружении неверно использованных символов.

В последующих разделах описываются символы из множества символов языка Си и объясняются правила их использования.

1.1.1 Буквы и цифры

Множество символов языка Си включает прописные и строчные буквы латинского алфавита и арабские цифры:

прописные латинские буквы: ABCDEFGHIJKLMNOPQRSTUVWXYZ;

строчные латинские буквы: abcdefghijkimnopqrstuvwxyz;

десятичные цифры: 0123456789.

Буквы и цифры используются при формировании констант, идентификаторов и ключевых слов (эти конструкции описаны ниже).

Компилятор языка Си рассматривает одну и ту же прописную и строчную буквы как различные символы.

1.1.2 Пробельные символы

Символы *пробел, табуляция, перевод строки, возврат каретки, новая страница, вертикальная табуляция* и *новая строка* называются пробельными, поскольку они имеют то же самое назначение, что и пробелы между словами и строками в тексте на естественном языке. Эти символы отделяют друг от друга лексемы, например константы и идентификаторы.

Символ CONTROL-Z (шестнадцатеричный код 1A) рассматривается как индикатор конца файла. Он автоматически вставляется текстовым редактором при создании файла в его конец. Компилятор языка Си завершает обработку файла с исходным текстом программы при обнаружении символа CONTROL-Z.

Компилятор языка Си игнорирует пробельные символы, если они используются не как компоненты символьных констант или символьных строк. Это позволяет использовать столько пробельных символов, сколько нужно для повышения наглядности программы.

Комментарии компилятор языка Си также рассматривает как пробельные символы.

1.1.3 Разделители

Разделители из множества символов языка Си используются для различных целей, от организации текста программы до определения указаний компилятору языка Си. Разделители перечислены в таблице 1.1.

Таблица 1.1.

Символ	Наименование	Символ	Наименование
,	Запятая	!	Восклицательный знак
.	Точка	|	Вертикальная черта
;	Точка с запятой	/	Наклонная черта вправо (слэш)
:	Двоеточие	\	Наклонная черта влево(обратный слэш)
?	Знак вопроса	~	Тильда
`	Одиночная кавычка (апостроф)	_	Подчеркивание
(	Левая круглая скобка	#	Знак номера
)	Правая круглая скобка	%	Процент
{	Левая фигурная скобка	&	Амперсанд
}	Правая фигурная скобка	^	Стрелка вверх
<	Знак "меньше"	-	Знак минус
>	Знак "больше"	=	Знак равенства
[	Левая квадратная скобка	+	Знак плюс
]	Правая квадратная скобка	*	Знак умножения (звездочка)

Эти символы имеют специальный смысл для компилятора языка Си. Правила их использования описываются в дальнейших разделах руководства. Элементы множества представимых символов, которые не представлены в данном списке (в частности, русские буквы), могут быть использованы только в символьных строках, символьных константах и комментариях.

1.1.4 Специальные символы

Специальные символы предназначены для представления пробельных и неграфических символов в строках и символьных константах. Обычно они используются для спецификации таких действий, как возврат каретки и табуляция для терминалов и принтеров, а также для представления символов, имеющих особый смысл (например, двойная кавычка). Специальный символ состоит из обратного слэша, за которым следует либо буква, либо знаки пунктуации, либо комбинация цифр. В таблице 1.2 приведен список специальных символов языка Си.

В СП ТС шестнадцатеричное значение байта может задаваться не только как \x, но и как \X.

В СП ТС, помимо перечисленных специальных символов, имеется еще один: \?—знак вопроса (код 0x3F). Он введен в состав языка Си для совместимости со стандартом ANSI на язык Си. Стандарт ANSI предусматривает использование пары знаков вопроса (??) в качестве признака последовательности, представляющей какой-либо символ, который может не иметь представления на клавиатуре терминала. Если же необходимо просто записать подряд два знака вопроса (например, в символьной строке), следует записать их так: ?\?. В СП ТС, однако, не реализованы последовательности, начинающиеся знаками ??, поэтому использование специального символа \? необязательно.

Таблица 1.2.

Специальный символ	Шестнадцатеричное значение в коде ASCII	Наименование
\n	0A	Новая строка
\t	09	Горизонтальная табуляция
\v	0B	Вертикальная табуляция
\b	08	Забой
\r	0D	Возврат каретки
\f	0C	Новая страница
\a	07	Звуковой сигнал
\'	2C	Апостроф
\"	22	Двойная кавычка
\\	5C	Обратный слэш
\ddd		Байтовое значение в восьмеричном представлении
\xdd		Байтовое значение в шестнадцатеричном представлении

Примечание. При работе с текстовым редактором ввод каждой строки завершается нажатием клавиши ENTER (ВВОД). Фактически при этом в текст вставляются два символа: возврат каретки и новая строка (с шестнадцатеричными значениями 0D и 0A в коде ASCII). Однако стандартные библиотечные функции ввода и вывода текстовой информации рассматривают эту пару символов как один символ — символ новой строки с шестнадцатеричным значением 0A. Этот символ представляется в символьных константах и символьных строках как \n. При чтении текстовой строки стандартные библиотечные функции заменяют упомянутую пару символов единственным символом новой строки, а при записи символа новой строки добавляют перед ним символ возврат каретки.

Если обратный слэш предшествует символу, не входящему в приведенный список, то обратный слэш игнорируется, а символ представляется обычным образом. Например, сочетание \h в строковой или символьной константе представляет символ h.

Конструкция \ddd позволяет задать произвольное байтовое значение как последовательность от одной до трёх восьмеричных цифр. Конструкция \xdd позволяет задать произвольное байтовое значение как последовательность от одной до двух шестнадцатеричных цифр, а для версии 5.0 СП MSC – до трех шестнадцатеричных цифр. Например, символ забой в коде ASCII может быть задан как \010 или \x08. Нулевой код может быть задан как \0 или \x0. В восьмеричном представлении байта могут быть заданы только восьмеричные цифры, причем по крайней мере одна цифра должна быть задана. Например, символ забой может быть задан как \10. Аналогично, в шестнадцатеричном представлении байта должна быть задана по крайней мере одна шестнадцатеричная цифра. Так, шестнадцатеричное представление символа забой может быть задано и как \x08, и как \x8.

Примечание. Если восьмеричное или шестнадцатеричное представление байта используется в составе строки, то рекомендуется полностью задавать все цифры представления. В противном случае, если символ, непосредственно следующий за представлением, случайно окажется восьмеричной или шестнадцатеричной цифрой, он будет интерпретироваться как часть этого представления. Например, в версии 4.0 СП MSC строка **\x7Bell** при выводе на печать будет выглядеть как **{ell**, поскольку **\x7B** проинтерпретируется как код левой фигурной скобки. Строка **\x07Be11** будет правильным представлением кода звукового сигнала с последующим словом **Bell**.

В СП TC разбор конструкций, представляющих байтовое значение, реализован не вполне корректно; так, запись "\1234" считается ошибочной, хотя она представляет восьмеричное значение 123 и символ '4'.

Специальные символы позволяют посылать неграфические управляющие последовательности на внешние устройства. Например, код **\033** (символ ESC в коде ASCII) часто используется как первый символ команд управления терминалом и принтером.

Помимо специальных символов, обратный слэш (\) используется также в качестве признака продолжения символьных строк и препроцессорных макроопределений. Если символ новой строки непосредственно следует за обратным слэшем, то комбинация "обратный слэш—символ новой строки" игнорируется и следующая строка рассматривается как продолжение предыдущей строки.

1.1.5 Операции

Операции — это комбинации символов, специфицирующие действия по преобразованию значений. Компилятор языка Си интерпретирует каждую из этих комбинаций как самостоятельную лексему.

В таблице 1.3. представлен список операций. Операции должны использоваться точно так, как они представлены в таблице,' без пробельных символов между символами в тех операциях, которые представлены несколькими символами.

Операция **sizeof** не включена в эту таблицу, поскольку задается ключевым словом, а не символом.

Таблица 1.3.

Операция	Наименование	Операция	Наименование
!	Логическое НЕ	^	Поразрядное исключающее ИЛИ
~	Обратный код	&&	Логическое И
+	Сложение; унарный плюс	||	Логическое ИЛИ
-	Вычитание; унарный минус	?:	Условная операция
*	Умножение; косвенная адресация	++	Инкремент
/	Деление	--	Декремент
%	Остаток от деления	=	Простое присваивание
<<	Сдвиг влево	+=	Присваивание со сложением
>>	Сдвиг вправо	-=	Присваивание с вычитанием
<	Меньше	*=	Присваивание с умножением
<=	Меньше или равно	/=	Присваивание с делением
>	Больше	%=	Присваивание с остатком от деления
>=	Больше или равно	>>=	Присваивание со сдвигом вправо
==	Равно	<<=	Присваивание со сдвигом влево
!=	Не равно	&=	Присваивание с поразрядным И
&	Поразрядное И; адресация	|=	Присваивание с поразрядным включающим ИЛИ
|	Поразрядное включающее ИЛИ	^=	Присваивание с поразрядным исключающим ИЛИ
,	Последовательное выполнение (запятая)		

Примечание. Условная операция ?: является не двухсимвольной, а тернарной (трехоперандной) операцией. Она имеет следующий формат: <операнд1> ? <операнд2> : <операнд3>

1.2 Константы

Константа — это число, символ или строка символов. Константы используются в программе для задания постоянных величин. В языке Си различают четыре типа константы: целые, с плавающей точкой, символьные константы и символьные строки.

1.2.1 Целые константы

Целая константа — это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целое значение. Десятичная константа имеет следующий формат представления:

<*цифры*>

<*цифры*> — последовательность из одной или более десятичных цифр от 0 до 9.

Восьмеричная константа имеет следующий формат представления:

0<*в-цифры*>

<*в-цифры*>—это одна или более восьмеричных цифр от 0 до 7. Запись нуля впереди обязательна.

Шестнадцатеричная константа имеет следующий формат представления:

0*x*<*ш-цифры*> или 0X<*ш-цифры*>

<*ш-цифры*> —одна или более шестнадцатеричных цифр. Шестнадцатеричная цифра может быть цифрой от 0 до 9 или буквой (большой или малой) от A до F. Допускается "смесь" больших и малых букв. Запись нуля впереди и следующего за ним символа **x** или **X** обязательна.

Между цифрами целой константы пробельные символы недопустимы. В таблице 1.4 приведены примеры целых констант. Константы, записанные в одной строке таблицы, используются для представления одного и того же значения.

Таблица 1.4.

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы
10	012	0xa или 0xA
132	0204	0x84
32179	076663	0x7dB3 или 0x7DB3

Целые константы всегда специфицируют положительные значения. Если требуется отрицательное значение, то необходимо сформировать константное выражение из знака

минус и следующей за ним константы. Знак минус рассматривается при этом как арифметическая операция.

Каждая целая константа имеет тип, определяющий ее представление в памяти (описание типов приведено в разделе 3.1 "Базовые типы данных"). Десятичные константы могут иметь тип **int** (целый тип) или **long** (длинный целый тип).

Восьмеричные и шестнадцатеричные константы в зависимости от размера могут иметь тип **int**, **unsigned int**, **long** или **unsigned long**. Если константа может быть представлена типом **int**, то компилятор языка Си присваивает ей тип **int**. Если ее значение больше, чем максимальное положительное значение, которое может быть представлено типом **int**, но может быть представлено тем же числом битов, что и **int**, ей присваивается тип **unsigned int**. Наконец, константа, значение которой больше, чем максимальное значение, представляемое типом **unsigned int**, задается типом **long** или, если размер этого типа также оказывается недостаточен, типом **unsigned long**. В таблице 1.5 показаны диапазоны значений констант различных типов для компьютера, на котором тип **int** имеет длину 16 битов и тип **long** имеет длину 32 бита.

Таблица1.5.

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы	Тип
0—32767	0—077777	0x0—0x7FFF	int
	0100000—0177777	0x8000—0xFFFF	unsigned int
32767—2147483647	02000001—017777777777	0x10000—0x7FFFFFFF	long
	020000000000—030000000000	0x80000000—0xFFFFFFFF	unsigned long

Из рассмотренных правил следует, что при преобразовании восьмеричных и шестнадцатеричных констант к более длинным типам не производится расширения знака (поскольку старший, знаковый бит всегда равен нулю).

Программист может явно определить для любой целой константы тип **long**, записав букву **"l"** или **"L"** в конец константы. Это позволяет расширить нижнюю границу диапазона значений констант любого типа до нуля. Например, константа со значением 10 будет иметь тип **long** только в том случае, если она будет записана с суффиксом **L**, т. е. 10L. В таблице 1.6 приведены примеры длинных целых констант.

Таблица 1.6.

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы
12L	012L	0xaL или 0xAL
0l	0115l	0x4fl или 0x4Fl

В СП ТС реализован также суффикс **U** (или **u**), означающий, что константа имеет тип **unsigned**. Можно использовать одновременно оба суффикса — **L** и **U** — для одной и той же константы. Кроме того, в СП ТС константе присваивается тип **unsigned long**, если ее значение превышает **65535**, независимо от наличия или отсутствия суффикса **U** (в СП MSC в этом случае константе был бы присвоен тип **long**).

1.2.2 Константы с плавающей точкой

Константа с плавающей точкой — это действительное десятичное положительное число. Оно включает целую часть, дробную часть и экспоненту. Константы с плавающей точкой имеют следующий формат представления:

[<*цифры*>][.<*цифры*>][<э>[-]<*цифры*>]

<*цифры*> —одна или более десятичных цифр (от 0 до 9); <э>—признак экспоненты, задаваемый символом **E** или **e**. Либо целая, либо дробная часть константы может быть опущена, но не обе сразу. Либо десятичная точка с дробной частью, либо экспонента может быть опущена, но не обе сразу.

Экспонента состоит из символа экспоненты, за которым следует целочисленное значение экспоненты, возможно со знаком плюс или минус.

Между цифрами или символами константы пробельные символы недопустимы.

```
Примеры констант с плавающей точкой:
15.75
1.575E1
1575e-2
25.
Примеры констант с плавающей точкой с опущенной целой частью:
.75
.0075e2
```

Константы с плавающей точкой всегда специфицируют положительные значения. Если требуются отрицательные значения, то необходимо сформировать константное выражение и?

знака минус и следующей за ним константы. Знак минус рассматривается при этом как арифметическая операция.

```
Примеры:
-0.0025
-2.5e-3
-.125
-.175E-2
```

Все константы с плавающей точкой имеют тип **double**. В СП ТС можно явно присвоить константе тип **float**, добавив к ней суффикс **f** или **F**.

1.2.3 Символьные константы

Символьная константа — это буква, цифра, знак пунктуации или специальный символ, заключенный в апострофы. Значение символьной константы равно коду представляемого ею символа. Символьная константа имеет следующую форму представления:

'<*символ*>'

<*Символ*> может быть любым символом из множества представимых символов (в том числе любым специальным символом), за исключением символов апостроф ('), обратный слэш (\) и новая строка.

Для представления символов апостроф и обратный слэш в качестве символьной константы необходимо вставить перед ними символ обратный слэш — '\'' и '\\'. Для представления символа новой строки используется запись '\n' (см. раздел 1.1.4).

Примеры символьных констант приведены в таблице 1.7.

Таблица 1.7.

Константа	Значение
'a'	Малая буква a
'?'	Знак вопроса
'\b'	Символ забой
'\x1B'	Символ ESC в коде ASCII

Символьные константы имеют тип **int**. Младший байт хранит код символа, а старший байт — знаковое расширение младшего байта.

Помимо односимвольных констант, в СП ТС реализованы двухсимвольные константы, например 'An', '\n\t', '\007\007'. Они представляются 16-битовым значением типа **int**, причем первый символ заносится в младший байт, а второй — в старший. Односимвольные константы также представляются 16-битовыми значениями типа **int**, и в старший байт, как и в СП MSC, заносится знаковое расширение младшего байта.

Компилятор языка Си имеет опцию, позволяющую определить тип **char** по умолчанию как беззнаковый тип — **unsigned char**. В этом случае старший байт любой односимвольной константы будет нулевым.

1.2.4 Символьные строки

Символьная строка — это последовательность символов, заключенная в двойные кавычки. Символьная строка рассматривается как массив символов, каждый элемент которого представляет отдельный символ. Символьная строка имеет следующую форму представления:

"**<***символы***>**"

<символы> — это произвольное (в том числе нулевое) количество символов из множества представимых символов, за исключением символов двойная кавычка ("), обратный слэш (\) и новая строка. Чтобы использовать эти символы внутри символьной строки, нужно представить их с помощью соответствующих специальных символов, как показано на следующих примерах:

```
"Это символьная строка\n"
"Первый \\ Второй"
"\"Да, конечно,\"— сказала она."
"Следующая строка — пустая:"
""
```

Для формирования символьных строк, занимающих несколько строк текста программы, используется комбинация символов — обратный слэш и новая строка. Компилятор языка Си проигнорирует эту комбинацию символов, а символьные строки объединит и представит в памяти как одну строку. Например, символьная строка:

```
"Длинные строки могут быть раз\
биты на части."
```

идентична строке:

```
"Длинные строки могут быть разбиты на части."
```

В СП MSC версии 5.0 и в СП ТС для формирования символьных строк, занимающих несколько строк текста программы, не требуется применения комбинации символов

обратный слэш и новая строка. Символьные строки, следующие друг за другом и не разделенные ничем, кроме пробельных символов, объединяются компилятором языка Си в одну строку.

Например, программа

```
main()
{
char *p;
p = "Данная программа — пример того, как можно"
" автоматически\nосуществлять объединение"
" строк в очень длинную строку;\n"
" такая форма записи может повысить"
" наглядность программ.\n";
printf("%s", p);
}
```

напечатает следующий текст:

```
Данная программа—пример того, как можно автоматически
осуществлять объединение строк в очень длинную строку;
такая форма записи может повысить наглядность программ.
```

Каждый символ символьной строки (в том числе каждый специальный символ) хранится в отдельном байте оперативной памяти. Нулевой символ ('\0') автоматически добавляется в качестве последнего байта символьной строки и служит признаком ее конца. Каждая символьная строка в программе рассматривается как отдельный объект; если в программе содержатся две идентичные символьные строки, то они будут занимать две различные области оперативной памяти.

В СП ТС реализована опция компиляции, позволяющая хранить в памяти только одну из идентичных строк.

Тип символьной строки—массив элементов типа **char**. Число элементов в массиве равно числу символов в символьной строке плюс один, поскольку нулевой символ (признак конца символьной строки) тоже является элементом массива.

1.3 Идентификаторы

Идентификаторы — это имена переменных, функций и меток, используемых в программе. Идентификатор вводится в объявлении переменной или функции, либо в качестве метки оператора. После этого его можно использовать в последующих операторах программы. Идентификатор — это последовательность из одной или более латинских букв, цифр и символов подчеркивания, которая начинается с буквы или символа подчеркивания. Допускается любое число символов в идентификаторе, однако только первые 32 символа рассматриваются компилятором языка Си как значащие. Если первые 32 символа у двух идентификаторов совпадают, компилятор языка Си рассматривает их как один и тот же идентификатор. Компоновщик также распознает 32 символа в именах глобальных переменных.

В идентификаторах версии 1.5 СП ТС допускается знак **$**, однако, идентификатор не может с него начинаться.

Компиляторы языка Си в СП MSC и СП ТС имеют опцию, позволяющую изменять число значащих символов в идентификаторах.

При использовании символов подчеркивания в качестве первых символов идентификаторов необходимо соблюдать осторожность, поскольку такие идентификаторы могут совпасть (войти в конфликт) с именами "скрытых" библиотечных функций.

```
Примеры идентификаторов:
temp1
top_of_page
skip12
```

Компилятор языка Си рассматривает буквы верхнего и нижнего регистров как различные символы. Поэтому можно создавать идентификаторы, которые совпадают орфографически, но различаются регистром букв. Например, каждый из следующих идентификаторов является уникальным:

```
add
ADD
Add
aDD
```

В СП ТС, однако, существует опция компиляции, позволяющая рассматривать в именах внешних переменных буквы верхнего и нижнего регистров как совпадающие.

Компилятор языка Си не допускает использования идентификаторов, совпадающих по написанию с ключевыми словами.

Например, идентификатор **while** недопустим (однако идентификатор **While**—допустим).

1.4 Ключевые слова

Ключевые слова — это предопределенные идентификаторы, которые имеют специальное значение для компилятора языка Си. Их использование строго регламентировано. Имена объектов программы не могут совпадать с ключевыми словами.

список ключевых слов:

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

При необходимости можно с помощью директив препроцессора определить для ключевых слов другие имена. Например, при наличии в программе макроопределения

#define BOOL int

слово **BOOL** можно использовать в объявлениях вместо слова **int**. Смысл объявлений (спецификация целого типа данных) от этого не изменится, однако программа станет более читабельной, если речь идет не просто о целых переменных, а о переменных, предназначенных для хранения значений булевского типа (булевский тип не реализован в языке Си как самостоятельный тип данных).

Имеется также ряд специальных ключевых слов:

СП MSC:	cdecl	СП TC:	asm	_cs	_BX
	far		cdecl	_ds	_CH
	fortran		far	_es	_CL
	huge		huge	_ss	_CX
	near		interrupt	_AH	_DH
	pascal		near	_AL	_DI
	const		pascal	_AX	_DL
	volatile		const	_BH	_DX
	interrupt		volatile	_BL	_SI
				_BP	_SP

В версии 4.0 СП MSC ключевые слова const и volatile зарезервированы, но использовать их невозможно. В версии 5.0 СП MSC ключевое слово volatile реализовано лишь синтаксически, а const — полностью (как синтаксически, так и семантически). В СП TC и const, и volatile полностью реализованы. В версии 4.0 СП MSC ключевое слово interrupt не реализовано.

Ключевое слово fortran используется для организации связи программ, написанных на языках Си и Фортран. По действию оно аналогично ключевому слову pascal. Ключевое слово asm применяется для записи в программе на языке Си ассемблерных инструкций. Специальные ключевые слова, начинающиеся с подчеркивания, представляют собой имена псевдопеременных, соответствующих регистрам микропроцессора. Ключевые слова cdecl, pascal, interrupt, near, far, huge, const, volatile объясняются подробно в разделе 3.3.3 "Описатели с модификаторами".

1.5 Комментарии

Комментарий — это последовательность символов, которая воспринимается компилятором языка Си как отдельный пробельный символ и игнорируется. Комментарий имеет следующий вид:

/* ***<символы>*** */

<символы> должны принадлежать множеству представимых символов. Символ новой строки также допустим внутри комментария. Это означает, что комментарии могут занимать более одной строки программного текста. Внутри комментария недопустима комбинация символов */. Это означает, что комментарии не могут быть вложенными.

Компилятор языка Си рассматривает комментарий как пробельный символ, поэтому комментарии допускается использовать везде, где можно использовать пробельные символы (но нельзя, например, внутри лексем). Компилятор языка Си игнорирует все символы комментария, поэтому даже запись в комментариях ключевых слов не приведет к ошибке.

Следующие примеры иллюстрируют использование комментариев:

```
/* Комментарии помогают документировать программу.*/
/* Комментарии могут содержать ключевые слова, например while и for */
/********************
Комментарий может занимать
несколько строк.
********************/
```

Так как комментарии не могут содержать вложенных комментариев, то следующий пример будет ошибочным:

```
/* Недопустимы /* вложенные */ комментарии */
```

Компилятор языка Си распознает первую комбинацию символов */ после слова "вложенные" как конец комментария. Затем компилятор языка Си попытается обработать оставшийся текст и выявить в нем лексемы языка Си, что приведет к ошибке.

Во избежание случайного возникновения ситуации вложенности комментариев рекомендуется ограничивать участок программного текста, который должен быть закомментирован, директивами препроцессора **#if 0** и **#endif**.

В СП ТС существует опция компиляции, допускающая вложенные комментарии.

2 СТРУКТУРА ПРОГРАММЫ

2.1 Исходная программа

Исходная программа представляет собой совокупность следующих элементов: директив препроцессора, указаний компилятору, объявлений и определений. Директивы препроцессора специфицируют действия препроцессора по преобразованию текста программы перед компиляцией. Указания компилятору — это специальные инструкции, которым компилятор языка Си следует во время компиляции.

Объявление переменной задает имя и атрибуты переменной. Определение переменной, помимо задания ее имени и атрибутов, приводит к выделению для нее памяти. Кроме того, определение задает начальное значение переменной (явно или неявно).

Объявление функции задает ее имя, тип возвращаемого значения и может задавать атрибуты ее формальных параметров.

Определение функции специфицирует тело функции, которое представляет собой составной оператор (блок), содержащий объявления и операторы. Определение функции также задает имя функции, тип возвращаемого значения и атрибуты ее формальных параметров.

Объявление типа позволяет программисту создать собственный тип данных. Оно состоит в присвоении имени некоторому базовому или составному типу языка Си. Для типа понятия объявления и определения совпадают.

Исходная программа может содержать произвольное число директив, указаний компилятору, объявлений и определений. Их синтаксис описан в последующих разделах. Порядок появления этих элементов в программе весьма существен; в частности, он влияет на возможность использования переменных, функций и типов в различных частях программы (см. раздел 2.4 "Время жизни и область действия").

Для того чтобы программа на языке Си могла быть скомпилирована и выполнена, она должна содержать по крайней мере одно определение — определение функции. Эта функция определяет действия, выполняемые программой. Если же программа содержит несколько функций, то среди них выделяется одна главная функция, которая должна иметь имя **main**. С нее начинается выполнение программы; она определяет действия, выполняемые программой, и вызывает другие функции. Порядок следования определений функций в исходной программе несуществен.

Если программа содержит только одну функцию, то она и является главной (и должна иметь имя **main**). В следующем примере приведена простая программа на языке Си:

```
int x = 1; /* определения переменных. */
inl y = 2;
extern int printf(char *, ..,); /* объявление функции */
main () /* определение главной функции */
{
int z; /* объявления переменных */
int w;
z = y + x; /* выполняемые операторы */
w = y - x;
printf("z = %d\nw = %d\n", z, w);
}
```

Эта исходная программа определяет функцию с именем **main** и объявляет функцию **printf**. Переменные **x** и **y** определяются на внешнем уровне, а переменные **z** и **w** объявляются внутри функции.

2.2 Исходные файлы

Текст программы на языке Си может быть разделен на несколько исходных файлов. Исходный файл представляет собой текстовый файл, который содержит либо всю программу, либо ее часть. При компиляции исходной программы каждый из составляющих ее исходных файлов должен быть скомпилирован отдельно, а затем связан с другими файлами компоновщиком. Отдельные исходные файлы можно объединять в один исходный файл, компилируемый как единое целое, посредством директивы препроцессора **#include**.

Исходный файл может содержать любую целостную комбинацию директив, указаний компилятору, объявлений и определений. Под целостностью подразумевается, что такие объекты, как определения функций, структуры данных либо набор связанных между собой директив условной компиляции, должны целиком располагаться в одном файле, т. е. не могут начинаться в одном файле, а продолжаться в другом.

Исходный файл не обязательно должен содержать выполняемые операторы. Иногда удобно размещать определения переменных в одном файле, а в других файлах использовать эти переменные путем их объявления. В этом случае определения переменных становятся легко доступными для поиска и модификации. Из тех же соображений именованные константы и макроопределения обычно собирают в отдельные файлы и включают их посредством директивы препроцессора **#include** в те исходные файлы, в которых они требуются.

Указания компилятору обычно действуют только для отдельных участков исходного файла. Специфические действия компилятора, задаваемые указаниями, определяются конкретной реализацией компилятора языка Си.

В нижеследующем примере исходная программа состоит из двух исходных файлов. Функции **main** и **max** представлены в отдельных файлах. Функция **main** использует функцию **max** в процессе своего выполнения.

```
/* исходный файл 1 — функция main */
#define ONE 1
#define TWO 2
#define THREE 3
extern int max (int, int); /* объявление функции */
main () /* определение функции */
{
int w = ONE, x = TWO, y = THREE;
int z = 0;
z = max(x, y);
z = max(z, w);
}
/* исходный файл 2 — функция max */
int max (a, b) /* определение функции */
int a, b;
{
if ( a > b )
return (a);
else
return (b);
}
```

В первом исходном файле функция **max** объявлена, но не определена. Такое объявление функции называется предварительным; оно позволяет компилятору контролировать обращение к функции до того, как она определена. Определение функции **main** содержит вызовы функции **max**.

Строки, начинающиеся с символа #, являются директивами препроцессора. Директивы указывают препроцессору на необходимость замены в первом исходном файле идентификаторов ONE, TWO, THREE на соответствующие значения. Область действия директив не распространяется на второй исходный файл.

Второй исходный файл содержит определение функции **max**. Это определение соответствует объявлению **max** в первом исходном файле. После того как оба исходных файла скомпилированы, они могут быть объединены компоновщиком и выполнены как единая программа.

2.3 Выполнение программы

Каждая программа на языке Си содержит главную функцию. В языке Си главная функция программы должна иметь имя **main**. С функции **main** начинается выполнение программы; обычно она управляет выполнением программы, организуя вызовы других функций. Программа может завершить выполнение по достижению конца функции **main**, однако может завершиться и в других точках путем вызова стандартных библиотечных функций, предназначенных для выхода из программы (см. описание функций **exit** и **abort** в разделе 12).

Исходная программа обычно включает в себя несколько функций, каждая из которых предназначена для выполнения определенной задачи. Функция **main** может вызывать эти функции, с тем чтобы выполнить ту или иную задачу. Когда функция вызывается, выполнение начинается с ее первого оператора. Функция возвращает управление при выполнении оператора **return** либо когда выполнение доходит до конца тела функции.

Все функции, включая функцию **main**, могут иметь формальные параметры. Вызываемые функции получают значения формальных параметров из вызывающих функций. Значения формальных параметров функции **main** могут быть получены извне — из командной строки

при вызове программы и из таблицы контекста операционной системы. Таблица контекста заполняется системными командами SET и PATH.

Для передачи данных программе через командную строку необходимо при вызове вслед за именем выполняемого файла, содержащего программу, задать ее аргументы. Аргументы должны быть отделены друг от друга пробелами или символами горизонтальной табуляции. Если требуется передать программе аргумент, содержащий внутри себя пробелы или символы горизонтальной табуляции, следует заключить его в двойные кавычки.

Аргументы передаются программе (точнее, функции **main**) как символьные строки.

Пример:

```
PROG 25 "ab c" 100
```

Программе с именем PROG передаются три аргумента — символьные строки "25", "ab c" и "100".

В процессе компоновки программы на языке Си в ее состав включается модуль поддержки выполнения. Этот модуль получает управление непосредственно от операционной системы при вызове программы, разбирает командную строку и передает аргументы функции **main**.

Для получения аргументов из командной строки и таблицы контекста в функции **main** должно быть объявлено три формальных параметра. В языке Си по традиции параметры функции **main** именуются **argc**, **argv** и **envp**, однако это не является требованием языка.

Пример объявления формальных параметров функции main:

```
main (int argc, char *argv[], char *envp []){}
```

Если программа не требует аргументов, то функцию **main** можно объявить без формальных параметров:

```
main()
{
…
}
```

Если аргументы передаются программе только через командную строку, то достаточно объявить только параметры **argc** и **argv**. Однако порядок объявления параметров существен; например, если программа принимает аргументы из таблицы контекста, а из командной строки не принимает, необходимо объявить все три параметра.

Параметр **argv** представляет собой массив адресов, каждый элемент которого указывает на строковое представление соответствующего по порядку аргумента, передаваемого программе. Параметр **argc** определяет общее число передаваемых аргументов. Первый элемент массива **argv** (т. е. **argv[0]**) всегда содержит имя программы, по которому она была вызвана. Этот элемент всегда заполнен, поэтому значение **argc** всегда равно по крайней мере 1. Доступ к первому аргументу, переданному программе, можно осуществить с помощью выражения **argv[1]**, к последнему аргументу — **argv[argc-1]**.

Параметр **envp** представляет собой указатель на массив строк, определяющих контекст, т.е. среду выполнения программы. Стандартные библиотечные функции **getenv** и **putenv** позволяют организовать удобный доступ к таблице контекста.

Существует еще один способ передачи аргументов функции **main** — при запуске программы как независимого подпроцесса из другой программы, также написанной на языке Си. Подробное описание этого способа приведено в разделе 12 в описании групп стандартных библиотечных функций **exec** и **spawn**.

2.4 Время жизни и область действия

Понятия "время жизни" и "область действия" являются очень важными для понимания структуры программ на языке Си. Время жизни переменной может быть либо "глобальным", либо "локальным". Объект с глобальным временем жизни характеризуется тем, что в течение всего времени выполнения программы с ним ассоциирована ячейка оперативной памяти и значение. Объекту с локальным временем жизни выделяется новая ячейка памяти при каждом входе в блок, в котором он определен или объявлен. Когда выполнение блока завершается, память, выделенная под локальный объект, освобождается и, следовательно, локальный объект теряет значение.

Блок представляет собой составной оператор. Составные операторы могут содержать объявления и операторы (см. раздел 5.3 "Составной оператор").

Тело функции представляет собой блок. Блоки в свою очередь могут содержать внутри себя другие, вложенные блоки. Из этого следует, что функции имеют блочную структуру. Однако функции не могут быть вложенными, т.е. определение функции не может содержаться внутри определения другой функции.

Объявления и определения, записанные внутри какого-либо блока (т. е. на внутреннем уровне), называются внутренними. Объявления и определения, записанные за пределами всех блоков (т. е. на внешнем уровне), называются внешними. Переменные и

функции могут быть объявлены как на внешнем уровне, так и на внутреннем. Переменные могут быть также определены на внутреннем уровне, а функции определяются только на внешнем уровне.

Все функции имеют глобальное время жизни. Переменные, определенные на внешнем уровне, всегда имеют глобальное время жизни. Переменные, определенные на внутреннем уровне, имеют локальное время жизни, однако путем указания для них спецификации класса памяти **static** можно сделать их время жизни глобальным.

Область действия объекта определяет, в каких участках программы допустимо использование имени этого объекта. Так, объект с глобальным временем жизни существует в течение всего времени выполнения программы, однако он доступен только в тех частях программы, на которые распространяется его область действия. Область действия объекта распространяется на блок или исходный файл, если в этом блоке или исходном файле известны тип и имя объекта. Объект может иметь глобальную или локальную область действия. Глобальная область действия означает, что объект доступен, или может быть через соответствующие объявления сделан доступным в пределах всех исходных файлов, образующих программу. Этот вопрос рассматривается в разделе 3.6 "Классы памяти". Локальная область действия означает, что объект доступен только в том блоке или файле, в котором он объявлен или определен.

Область действия переменной, объявленной на внешнем уровне, распространяется от точки программы, в которой она объявлена, до конца исходного файла, на все функции и вложенные блоки, за исключением случаев локального переобъявления (см. ниже). Область действия этой переменной можно распространить и на другие исходные файлы путем ее объявления в этих файлах (см. раздел 3.6 "Классы памяти"). Однако область действия переменной, объявленной на внешнем уровне с классом памяти **static**, распространяется только до конца исходного файла, содержащего ее объявление.

Область действия переменной, объявленной на внутреннем уровне, распространяется от точки программы, в которой она объявлена, до конца блока, содержащего ее объявление. Такая переменная называется локальной.

Если переменная, объявленная внутри блока, имеет то же самое имя, что и переменная, объявленная на внешнем уровне, то внутреннее объявление переменной заменяет (вытесняет) в пределах блока внешнее объявление. Этот механизм называется локальным переобъявлением переменной. Область действия переменной внешнего уровня восстанавливается при завершении блока.

Блок, вложенный внутрь другого блока, может в свою очередь содержать локальные переобъявления переменных, объявленных в охватывающем блоке. Локальное переобъявление переменной имеет силу во внутреннем блоке, а действие ее первоначального объявления восстанавливается, когда управление возвращается в охватывающий блок. Область действия переменной из внешнего (охватывающего) блока распространяется на все внутренние (вложенные) блоки, за исключением тех блоков, в которых она локально переобъявляется.

Область действия типов, созданных программистом, подчиняется тем же правилам, что и область действия переменных.

Использование функций в языке Си имеет некоторые отличия от использования переменных. Во-первых, как уже говорилось, на внутреннем уровне функция может быть только объявлена, а на внешнем уровне — и объявлена, и определена. Во-вторых, для работы с переменной ее необходимо предварительно явно объявить, а для того, чтобы вызвать функцию, это необязательно. Вызов функции компилятор языка Си рассматривает как неявное объявление функции с типом возвращаемого значения **int** и классом памяти **extern**. Если далее в файле встретится объявление или определение этой функции с другими атрибутами, компилятор сообщит об ошибке.

Область действия функции, объявленной со спецификацией класса памяти **static**, распространяется на весь исходный файл, в котором она объявлена, т.е. она может быть вызвана из любой точки этого файла, за исключением тех блоков, в которых она локально переобъявляется. Например, в каком-то блоке может быть объявлена функция с тем же именем и классом памяти **extern**, определенная в другом файле.

Область действия функции, объявленной с классом памяти **extern**, распространяется на все исходные файлы программы, т.е. она может быть вызвана из любой точки любого файла, за исключением блоков, в которых она локально переобъявляется. Например, если в каком-то из файлов на внешнем уровне объявлена функция с тем же именем и классом памяти **static**, то именно она будет вызываться в этом файле.

Помимо вызова, существует еще одна операция, применимая к функции, — получение ее адреса. Для этой операции функция ничем не отличается от переменной, поэтому функция должна быть предварительно объявлена. Для операции получения адреса область действия функции не зависит от се класса памяти и распространяется от точки

объявления функции до конца исходного файла, за исключением случаев локального переобъявления.

В таблице 2.1 показана взаимосвязь основных факторов, которые определяют время жизни и область действия функций и переменных. При обсуждении области действия переменных мы использовали термин "объявление"; в таблице 2.1 конкретизировано для каждого случая, идет ли речь об объявлении или определении. Область действия функций в таблице 2.1 показана под углом зрения операции получения адреса, а не операции вызова функции. Более подробная информация о влиянии спецификаций класса памяти на область действия объекта приведена в разделе 3.6 "Классы памяти".

Таблица 2.1.

Уровень	Объект	Спецификация класса памяти	Время жизни	Область действия
Внешний	Определение переменной	static	Глобальное	Остаток исходного файла
	Объявление переменной	extern	Глобальное	Остаток исходного файла
	Объявление или определение функции	static или extern	Глобальное	Остаток исходного файла
Внутренний	Объявление переменной	extern	Глобальное	Блок
	Определение переменной	static	Глобальное	Блок
	Определение переменной	auto или register	Локальное	Блок
	Объявление функции	extern или static	Локальное	Остаток исходного файла

Следующий пример программы иллюстрирует понятия блочной структуры, времени жизни и области действия переменных.

```
/* i определяется на внешнем уровне */
int i = 1;
/* функция main определяется на внешнем уровне */
main()
{
/* печатается 1 (значение переменной i внешнего уровня) */
printf("%d\n", i);
/* первый вложенный блок */
   {
   /* i переопределяется */
   int i = 2, j = 3;
   /* печатается 2, 3 */
   printf("%d\n%d\n", i, j);
   /* второй вложенный блок */
      {
      /* i переопределяется */
      int i = 0;
      /* печатается 0, 3 */
      printf("%d\n%d\n, i, j);
      /* конец второго вложенного блока */
      }
   /* печатается 2 (восстановлено определение i в охватывающем блоке) */
   printf("%d\n", i);
   /* конец первого вложенного блока */
   }
печатается 1 (восстановлено определение внешнего уровня)*/
printf("%d\n", i);
/* конец определения функции main */
}
```

В этом примере показано четыре уровня области действия: самый внешний уровень и три уровня, образованных блоками. Функция **printf** определена в библиотеке стандартных функций (см. раздел 12). Функция **main** печатает значения 1, 2, 3, 0,3,2,1.

2.5 Пространства имен

В программе на языке Си имена (идентификаторы) используются для ссылок на различного рода объекты — функции, переменные, формальные параметры и т. п. При соблюдении определенных правил, описанных в данном разделе, допускается использование одного и того же идентификатора для более чем одного программного объекта.

Чтобы различать идентификаторы объектов различного рода, компилятор языка Си устанавливает так называемые "пространства имен". Во избежание противоречий имена внутри одного пространства должны быть уникальными, однако в различных пространствах могут содержаться идентичные имена. Это означает, что можно использовать один и тот же идентификатор для двух или более различных объектов, если имена объектов

принадлежат к различным пространствам. Однозначное разрешение вопроса о том, на какой объект ссылается идентификатор, компилятор языка Си осуществляет по контексту появления данного идентификатора в программе. Ниже перечисляются виды объектов, которые можно именовать в программе на языке Си, и соответствующие им четыре пространства имен.

Таблица 2.2.

Объекты	Пространство имен
Переменные, функции, формальные параметры, элементы списка перечисления, typedef	Уникальность имен в пределах этого пространства тесно связана с понятием области действия. Это выражается в том, что в данном пространстве могут содержаться совпадающие идентификаторы, если области действия именуемых ими объектов не пересекаются. Другими словами, совпадение идентификаторов возможно только при локальном переобъявлении (см. раздел 2.4). Обратите внимание на то, что имена формальных параметров функции сгруппированы в одном пространстве с именами локальных переменных. Поэтому переобъявление формальных параметров внутри любого из блоков функции недопустимо, typedef — это объявления имен типов (см. раздел 3.8.2).
Теги	Теги всех переменных перечислимого типа, структур и объединений (см. разделы 3.4.2 — 3.4.4) сгруппированы в одном пространстве имен. Каждый тег переменной перечислимого типа, структуры или объединения должен быть отличен от других тегов с той же самой областью действия. Ни с какими другими именами имена тегов не конфликтуют.
Элементы структур и объединений	Элементы каждой структуры или объединения) образуют свое пространство имен, поэтому имя каждого элемента должно быть уникальным внутри структуры или объединения, но не обязано отличаться от любого другого имени в программе, включая имена элементов других структур и объединений.
Метки операторов	Метки операторов образуют отдельное пространство имен. Каждая метка должна быть отлична от всех других меток операторов в той же самой функции. В разных функциях могут быть одинаковые метки.

```
Пример: struct student
{
char student [20]; /*массив из 20 элементов типа char*/
int class;
int id;
} student; /* структура из трех элементов */
```

В этом примере имя тега структуры, элемента структуры и самой структуры относится к трем различным пространствам имен, поэтому не возникает противоречия между тремя объектами с одинаковым именем **student**. Компилятор языка Си определит по контексту использования, на какой из объектов ссылается идентификатор в каждом конкретном случае. Например, когда идентификатор **student** появится после ключевого слова **struct**, это будет означать, что именуется тег структуры. Когда идентификатор **student** появится после операции выбора элемента (-> или .), то это будет означать, что именуется элемент структуры. В любом другом контексте идентификатор **student** будет рассматриваться как ссылка на переменную структурного типа.

3 ОБЪЯВЛЕНИЯ

В этом разделе описываются формат и составные части объявлений переменных, функций и типов. В разделе 2.1 были введены понятия объявления и определения. Далее по тексту будем для краткости называть и объявления, и определения "объявлениями", если явно не конкретизируется то или иное понятие.

Объявления в языке Си имеют следующий синтаксис:

<*спецификация КП***>**

<*спецификация типа*>

<*описатель***>** [=**<***инициализатор***>**] [,<*описатель*> [= <*инициализатор*>…]];

где:

<*спецификация КП*> — спецификация класса памяти;

<*спецификация типа*> — имя типа, присваиваемого объекту;

<*описатель*> — идентификатор простой переменной либо более сложная конструкция при объявлении переменной составного типа;

<*инициализатор*> — значение или последовательность значений, присваиваемых переменной при объявлении.

В некоторых случаях спецификация класса памяти и/или спецификация типа может быть опущена.

Все переменные в языке Си должны быть явно объявлены перед их использованием, за исключением формальных параметров, имеющих тип **int**. Функции могут быть объявлены явно, посредством задания объявления или определения функции, либо неявно, если их вызов следует до определения или объявления.

Спецификация класса памяти влияет на то, в какой области памяти хранится объявляемый объект, производится ли его неявная инициализация и на какие участки программы распространяется его область действия. Местоположение объявления в программе, а также наличие или отсутствие других объявлений этой же переменной также существенны при определении ее области действия. Классы памяти описаны в разделе 3.6.

В языке Си определен набор базовых типов данных. Новые типы данных можно добавлять к этому набору посредством их объявления на основе уже определенных типов данных. Спецификация типа позволяет задавать для объекта либо базовый тип данных (см. раздел 3.1), либо структурный тип (см. раздел 3.4.3), либо тип объединение (см. раздел 3.4.4).

Не считается ошибкой объявление внешнего уровня, в котором отсутствует и спецификация класса памяти, и спецификация типа. В этом случае предполагается тип **int**. Однако объявление, состоящее только из идентификатора, например

```
n;
```

недопустимо, т.е. простая переменная не может быть объявлена подобным образом; может быть объявлен указатель, массив или функция.

Объявление должно содержать один или более описателей. В простейшем случае, когда объявляется простая переменная, тип которой задан <*спецификацией типа*>, описатель представляет собой идентификатор. Для объявления массива значений специфицированного типа (см. раздел 3.4.5), либо функции, возвращающей значение специфицированного типа (см. раздел 3.5), либо указателя на значение специфицированного типа (см. раздел 3.4.6), идентификатор дополняется, соответственно, квадратными скобками, круглыми скобками или звездочкой. В одном объявлении может быть задано несколько описателей различных объектов, имеющих одинаковый класс памяти и тип.

Определения функций описаны в разделе 6.2, инициализаторы — в разделе 3.7.

3.1 Базовые типы данных

В языке Си реализован набор типов данных, называемых "базовыми" типами. Спецификации этих типов перечислены в таблице 3.1.

Таблица 3.1.

Базовые типы	Спецификация типов	
Целые	signed char	знаковый символьный
	signed int	знаковый целый
	signed short int	знаковый короткий целый
	signed long int	знаковый длинный целый
	unsigned char	беззнаковый символьный
	unsigned int	беззнаковый целый
	unsigned short int	беззнаковый короткий целый
	unsigned long int	беззнаковый длинный целый
Плавающие	float	плавающий одинарной точности
	double	плавающий двойной точности
	long float	длинный плавающий одинарной точности
	long double	длинный плавающий двойной точности
Прочие	void	пустой
	enum	перечислимый

Тип **long float** реализован только в версии 4.0 СП MSC и эквивалентен типу **double**. В версии 5.0 СП MSC и в СП TC реализован тип **long double**, причем в версии 5.0 СП MSC и версии 1.5 СП TC он эквивалентен типу **double**, а в версии 2.0 СП TC является самостоятельным типом размером 80 битов.

Типы **char**, **int**, **short** и **long** имеют две формы — знаковую (**signed**) и беззнаковую (**unsigned**). В совокупности они образуют целый тип. Перечислимый тип также служит для представления целых значений, однако, переменная перечислимого типа может принимать

значения только из набора, заданного в ее объявлении. Спецификации типов **float** и **double** относятся к плавающему типу.

Целый тип (включая перечислимый тип) и плавающий тип в совокупности образуют арифметический тип.

Тип **void** (пустой) имеет специальное назначение. Указание спецификации типа **void** в объявлении функции означает, что функция не возвращает значений. Указание типа **void** в списке объявлений аргументов в объявлении функции означает, что функция не принимает аргументов. Можно объявить указатель на тип **void**; он будет указывать на любой, т.е. неспецифицированный тип. Тип **void** может быть указан в операции приведения типа. Приведение значения выражения к типу **void** явно указывает на то, что это значение не используется. Нельзя объявить переменную типа **void**.

При записи спецификаций целого и плавающего типа допустимы сокращения, приведенные в таблице 3.2. Например, в целых типах ключевое слово **signed** может быть опущено. Если ключевое слово **unsigned** отсутствует в записи спецификации типа **short**, **int** или **long**, то тип целого будет знаковым, даже если опущено ключевое слово **signed**.

По умолчанию тип **char** всегда имеет знак. Однако существует опция компилятора языка Си, позволяющая изменить умолчание для **char** со знакового типа на беззнаковый. Если эта опция задана, то сокращение **char** имеет тот же смысл, что и **unsigned char**, и, следовательно, для объявления символьной переменной со знаком должно быть записано ключевое слово **signed**.

Таблица 3.2.

Спецификации типов и их сокращения

Спецификация типа	Сокращение
signed char	char
signed int	signed, int
signed short int	short, signed short
signed long int	long, signed long
unsigned char	-
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long
float	-
long float	double
long double	double (СП MSC 5.0, СП TC 1.5)
long double	—(СП TC 2.0)

Примечание. В данной книге в основном используются сокращенные формы записи спецификаций типов, перечисленные в таблице 3.2; при этом предполагается, что тип **char** по умолчанию имеет знак.

3.2 Области значений

Область значений — это интервал от минимального до максимального значения, которое может быть представлено в переменной данного типа. В таблице 3.3 приведен размер занимаемой памяти и области значений переменных для каждого типа. Поскольку переменных типа **void** не существует, он не включен в эту таблицу.

Таблица 3.3.

Размер памяти и область значений типов

Тип	Представление в памяти	Область значений
char	1 байт	от -128 до 127
int	зависит от реализации	
short	2 байта	от -32768 до 32767
long	4 байта	от -2.147.483.648 до 2.147.483.647
unsigned char	1 байт	от 0 до 255
unsigned	зависит от реализации	
unsigned short	2 байта	от 0 до 65535
unsigned long	4 байта	от 0 до 4.294.967.295
float	4 байта	стандартный формат IEEE
double	8 байтов	стандартный формат IEEE
long double	10 байтов	стандартный формат IEEE

Тип **char** может использоваться для хранения буквы, цифры или другого символа из множества представимых символов. Значением объекта типа **char** является код, соответствующий данному символу. Тип **char** интерпретируется как однобайтовое целое с

областью значений от -128 до 127. Тип **unsigned char** может содержать значения в интервале от 0 до 255. В частности, буквы русского алфавита имеют коды, соответствующие типу **unsigned char**.

Следует отметить, что представление в памяти и область значений для типов **int** и **unsigned int** не определены в языке Си. В большинстве систем программирования размер типа **int** (со знаком или без знака) соответствует реальному размеру целого машинного слова. Например, на 16-разрядном компьютере тип **int** занимает 16 разрядов, или 2 байта. На 32-разрядном компьютере тип **int** занимает 32 разряда, или 4 байта. Таким образом, тип **int** эквивалентен либо типу **short int** (короткое целое), либо типу **long int** (длинное целое), в зависимости от компьютера. Аналогично, тип **unsigned int** эквивалентен либо типу **unsigned short int**, либо типу **unsigned long int**. Однако рассматриваемые в данной книге компиляторы языка Си, разработанные для моделей IBM PC с 16-разрядным машинным словом, при работе на IBM PC/AT с процессором Intel 80386 (имеющим 32-разрядное машинное слово) отводят для типа **int** и **unsigned int** только 16 разрядов.

Спецификации типов **int** и **unsigned int** широко используются в программах на Си, поскольку они позволяют наиболее эффективно работать с целыми значениями на данном компьютере. Однако, поскольку размер типов **int** и **unsigned int** является машинно-зависимым, программы, зависящие от конкретного размера типа **int** или **unsigned int** на каком-либо компьютере, могут быть непереносимы на другой компьютер. Переносимость программ можно повысить, если использовать для ссылки на размер типа данных операцию **sizeof**.

Порядок размещения байтов в памяти для базовых целых типов следующий (по возрастанию адресов):

для типа **short** — b0, b1;

для типа **long** — b0, b1, b2, b3,

где b0—младший байт.

Архитектура процессора Intel 8086/88 позволяет размещать переменные различного размера в памяти, как с четного, так и с нечетного адреса. Однако в последнем случае обращение к переменным будет более медленным. В СП ТС существует опция компиляции, задающая выравнивание всех объектов, занимающих более одного байта, на границу четного адреса. Память при этом будет использоваться менее эффективно, но скорость обращения к переменным возрастет. В СП MSC по умолчанию производится выравнивание на границу четного адреса. В версии 5.0 СП MSC существует опция компиляции, обеспечивающая выравнивание на границу, заданную программистом. Вопросы выравнивания структур рассматриваются в разделе 3.4.3.

Согласно правилам преобразования типов в языке Си (см. раздел 5 "Выражения"), не всегда возможно использовать в выражении максимальное или минимальное значение для константы данного типа.

Допустим, требуется использовать в выражении значение -32768 типа **short**. Константное выражение -32768 состоит из арифметической операции отрицания (-), предшествующей значению константы 32768. Поскольку значение 32768 слишком велико для типа **short**, компилятор языка Си представляет его типом **long** и, следовательно, константа -32768 будет иметь тип **long**. Значение -32768 может быть представлено типом **short** только путем явного приведения его к типу **short** с помощью выражения (**short**) (-32768). Информация при этом не будет потеряна, поскольку значение -32768 может быть представлено двумя байтами памяти.

Восьмеричные и шестнадцатеричные константы могут иметь знаковый или беззнаковый тип, в зависимости от их значения (см. раздел 1.2.1). Однако метод присвоения компилятором языка Си типов восьмеричным и шестнадцатеричным константам гарантирует, что в выражениях они будут вести себя как беззнаковые целые (поскольку их знаковый бит всегда равен нулю).

СП ТС позволяет явно присваивать константам беззнаковый тип с помощью суффикса **u**.

Для представления значений с плавающей точкой используется стандартный формат IEEE (Institute of Electrical and Electronics Engineers, Inc.). Значения типа **float** занимают 4 байта, состоящих из бита знака, 7-битовой двоичной экспоненты и 24-битовой мантиссы. Мантисса представляет число в интервале от 1.0 до 2.0. Поскольку старший бит мантиссы всегда равен единице, он не хранится в памяти. Это представление дает область значений приблизительно от 3.4E-38 до 3.4E+38.

Значения типа **double** занимают 8 байтов. Их формат аналогичен формату **float**, за исключением того, что экспонента занимает 11 битов, а мантисса 52 бита плюс неявный старший бит, единичный. Это дает область значений приблизительно от 1.7E-308 до 1.7E+308.

Значения типа **long double** занимают 80 битов; их область значений—от 3.4E-4932 до 1.1E+4932. Формат их аналогичен формату **double**, однако, мантисса длиннее на 16 битов.

3.3 Описатели

3.3.1 Синтаксис описателей

Синтаксис описателей рекурсивными правилами:

<*идентификатор***>**

<*описатель***>** []

<*описатель*> [<*константное-выражение*>]

<описатель***>**

<*описатель***>**()

<*описатель***>**(**<***список типов аргументов***>**)

(**<***описатель***>**)

Описатели в языке Си позволяют объявить следующие объекты: простые переменные, массивы, указатели и функции. В простейшем случае, если объявляется простая переменная базового типа, либо структура, либо объединение, описатель представляет собой идентификатор. При этом объекту присваивается тип, заданный спецификацией типа.

Для объявления массива значений специфицированного типа, либо функции, возвращающей значение специфицированного типа, либо указателя на значение специфицированного типа, идентификатор дополняется, соответственно, квадратными скобками (справа), круглыми скобками (справа) или звездочкой (слева). В дальнейшем будем называть квадратные скобки, круглые скобки и звездочку признаками типа массив, функция и указатель, соответственно.

Следующие примеры иллюстрируют простейшие формы описателей:

```
int list(20)      —массив list значений целого типа;
char *cp          -указатель cp на значение типа char;
double func()     —функция func, возвращающая значение типа double.
```

3.3.2 Интерпретация составных описателей

Составной описатель — это идентификатор, дополненный более чем одним признаком типа массив, указатель или функция.

С одним идентификатором можно образовать множество различных комбинаций признаков типа массив, указатель или функция. Некоторые комбинации недопустимы. Например, массив не может содержать в качестве элементов функции, а функция не может возвращать массив или функцию.

При интерпретации составных описателей сначала рассматриваются квадратные скобки и круглые скобки, расположенные справа от идентификатора. Квадратные и круглые скобки имеют одинаковый приоритет. Они интерпретируются слева направо. После них справа налево рассматриваются звездочки, расположенные слева от идентификатора. Спецификация типа рассматривается на последнем шаге после того, как описатель уже полностью проинтерпретирован.

Для изменения действующего по умолчанию порядка интерпретации описателя можно использовать внутри него круглые скобки.

Правило интерпретации составных описателей может быть названо чтением "изнутри — наружу". Начать интерпретацию нужно с идентификатора и проверить, есть ли справа от него открывающие квадратные или круглые скобки. Если они есть, то проинтерпретировать правую часть описателя. Затем следует проверить, есть ли слева от идентификатора звездочки, и, если они есть, проинтерпретировать левую часть. Если на какой-либо стадии интерпретации справа встретится закрывающая круглая скобка (которая используется для изменения порядка интерпретации описателя), то необходимо сначала полностью провести интерпретацию внутри данной пары круглых скобок, а затем продолжить интерпретацию справа от закрывающей круглой скобки.

На последнем шаге интерпретируется спецификация типа. После этого тип объявленного объекта полностью известен.

Следующий пример иллюстрирует применение правила интерпретации составных описателей. Последовательность шагов интерпретации пронумерована.

```
char*(*(*var)())[10].
  7  6 4 2 1   3  5
1. Идентификатор var объявлен как
2. Указатель на
3. Функцию, возвращающую
4. Указатель на
5. Массив из 10 элементов, которые являются
6. Указателями на
```

7. Значения типа char.

В приведенных ниже примерах обратите внимание на то, как применение круглых скобок может изменять смысл объявлений.

1. int *var[5]; — массив **var** указателей на значения типа **int**.
2. int (*var)[5]; — указатель **var** на массив значений типа **int**.
3. long *var(); — функция **var**, возвращающая указатель на значение типа **long**.
4. long (*var)(); — указатель **var** на функцию, возвращающую значение типа **long**.
5. struct both {
   int a;
   char b;
   }(*var[5])(); - массив **var** указателей на функции, возвращающие структуры типа **both**.
6. double (*var())[3]; — функция **var**, возвращающая указатель на массив из трех значений типа **double**.
7. union sign {
   int x;
   unsigned y;
   } **var[5][5]; — массив **var**, элементы которого являются массивами указателей на указатели на объединения типа **sign**.
8. union sign *(*var[5])[5]; — массив **var**, элементы которого являются указателями на массив указателей на объединения типа **sign**.

Пояснения к примерам:

В первом примере **var** объявляется как массив, поскольку признак типа массив имеет более высокий приоритет, чем признак типа указатель. Элементами массива являются указатели на значения типа **int**.

Во втором примере скобки меняют смысл объявления из первого примера. Теперь признак типа указатель применяется раньше, чем признак типа массив, и переменная **var** объявляется как указатель на массив из пяти значений типа **int**.

В третьем примере, поскольку признак типа функция имеет более высокий приоритет, чем признак типа указатель, **var** объявляется как функция, возвращающая указатель на значение типа **long**.

Четвертый пример аналогичен второму. С помощью скобок обеспечивается применение признака типа указатель прежде, чем признака типа функция, поэтому переменная **var** объявляется как указатель на функцию, возвращающую значение типа **long**.

Элементы массива не могут быть функциями. Однако в пятом примере показано, как объявить массив указателей на функции. В этом примере переменная **var** объявлена как массив из пяти указателей на функции, возвращающие структуры типа **both**. Заметьте, что круглые скобки, в которые заключено выражение **var[5]**, обязательны. Без них объявление становится недопустимым, поскольку объявляется массив функций:

struct both *var[5] (struct both, struct both);

В шестом примере показано, как объявить функцию, возвращающую указатель на массив. Здесь объявлена функция **var**, возвращающая указатель на массив из трех значений типа **double**. Тип аргумента функции задан составным абстрактным описателем (см. раздел 3.8.3), также специфицирующим указатель на массив из трех значений типа **double**, В отсутствие круглых скобок, заключающих звездочку, типом аргумента был бы массив из трех указателей на значения типа **double**.

В седьмом примере показано, что указатель может указывать на другой указатель, а массив может содержать массивы. Здесь **var** является массивом из пяти элементов. Каждый элемент, в свою очередь, также является массивом из пяти элементов, каждый из которых является указателем на указатель на объединение типа **sign**. Массив массивов является аналогом двумерного массива в других языках программирования.

В восьмом примере показано, как круглые скобки изменили смысл объявления из седьмого примера. В этом примере **var** является массивом из пяти указателей на массив из пяти указателей на объединения типа **sign**.

3.3.3 Описатели с модификаторами

В разделе 1.4 "Ключевые слова" приведен перечень специальных ключевых слов, реализованных в СП MSC и СП TC. Использование специальных ключевых слов (называемых в дальнейшем модификаторами) в составе описателей позволяет придавать объявлениям специальный смысл. Информация, которую несут модификаторы, используется компилятором языка Си в процессе генерации кода.

Рассмотрим правила интерпретации объявлений, содержащих модификаторы **const**, **volatile**, **cdecl**, **pascal**, **near**, **far**, **huge**, **interrupt**.

3.3.3.1 Интерпретация описателей с модификаторами

Модификаторы **cdecl**, **pascal**, **interrupt** воздействуют на идентификатор и должны быть записаны непосредственно перед ним.

Модификаторы **const**, **volatile**, **near**, **far**, **huge** воздействуют либо на идентификатор, либо на звездочку, расположенную непосредственно справа от модификатора. Если справа расположен идентификатор, то модифицируется тип объекта, именуемого этим идентификатором. Если же справа расположена звездочка, то модифицируется тип объекта, на который указывает эта звездочка, т.е. эта звездочка представляет собой указатель на модифицированный тип. Таким образом, конструкция <*модификатор*>* читается как "указатель на модифицированный тип". Например,

int const *p; - это указатель на **const int**, а

int * const p; - это **const** указатель на **int**. Модификаторы **const** и **volatile** могут также записываться и перед спецификацией типа.

В СП TC использование модификаторов **near**, **far**, **huge** ограничено: они могут быть записаны только перед идентификатором функции или перед признаком указателя (звездочкой).

Допускается более одного модификатора для одного объекта (или элемента описателя). В следующем примере тип функции **func** модифицируется одновременно специальными ключевыми словами **far** и **pascal**. Порядок специальных ключевых слов не важен, т. е. комбинации **far pascal** и **pascal far** имеют один и тот же смысл.

```
int far * pascal far func();
```

Тип значения, возвращаемого функцией **func**, представляет собой указатель на значения типа **int**. Тип этих значений модифицирован специальным ключевым словом **far**.

Как обычно, в объявлении могут быть использованы круглые скобки для изменения порядка его интерпретации.

Пример:

```
char far *(far *getint)(int far *);
 7    6      2   1  3 5    4
```

В примере показано объявление с различными вариантами расположения модификатора **far**. Учитывая правило, согласно которому модификатор воздействует на элемент описателя, расположенный справа от него, можно проинтерпретировать это объявление следующим образом (шаги интерпретации пронумерованы):

1. Идентификатор getint объявляется как
2. Указатель на far
3. Функцию, требующую
4. Один аргумент, который является указателем на far
5. Значение типа int
6. И возвращающую указатель на far
7. Значение типа char

3.3.3.2 Модификаторы const и volatile

Модификатор **const** не допускает явного присваивания значения переменной либо других косвенных действий по изменению ее значения, таких как выполнение операций инкремента и декремента. Значение указателя, объявленного с модификатором **const**, не может быть изменено, в отличие от значения объекта, на который он указывает. В СП MSC, в отличие от СП TC, недопустима также инициализация **const** объектов, имеющих класс памяти **auto** (поскольку их инициализация должна выполняться каждый раз при входе в блок, содержащий их объявления).

Применение модификатора **const** помогает выявить нежелательные присваивания значений переменным. Переменные, объявленные с модификатором **const**, могут быть загружены в ячейки постоянной памяти (ПЗУ).

Модификатор **volatile** противоположен по смыслу модификатору **const**. Он указывает на то, что значение переменной может быть изменено; но не только непосредственно программой, а также и внешним воздействием, например программой обработки прерываний, либо, если переменная соответствует порту ввода/вывода, обменом с внешним устройством. Объявление объекта с модификатором **volatile** предупреждает компилятор языка Си, что не следует делать предположений относительно стабильности значения объекта в момент вычисления содержащего его выражения, т. к. значение может (теоретически) измениться в любой момент. Для выражений, содержащих объекты типа **volatile**, компилятор языка Си не будет применять методы оптимизации, а сами объекты не будет загружать в машинные регистры.

Возможно одновременное использование в объявлении модификаторов **const** и **volatile**. Это означает, что значение объявляемой переменной не может модифицироваться программой, но подвержено внешним воздействиям.

Если с модификатором **const** или **volatile** объявляется переменная составного типа, то действие модификатора распространяется на все ее составляющие элементы. Возможно применение модификаторов **const** и **volatile** в составе объявления **typedef**.

Примечание. При отсутствии в объявлении спецификации типа и наличии модификатора **const** или **volatile** подразумевается спецификация типа **int**.

Примеры:

```
float const pi = 3.1415926;
const maxint = 32767;
/* указатель с неизменяемым значением*/
char *const str = "Здравствуй, мир!";
/* указатель на неизменяемую строку */
char const *str2 = "Здравствуй, мир!";
С учетом приведенных объявлений следующие операторы недопустимы:
pi = 3.0;/* Присвоение значения константе */
i = maxini--; /* Уменьшение константы */
str = "Привет!"; /* Переназначение указателя */
```

Однако вызов функции *strcpy*(str,"Привет!") допустим, т. к. в данном случае осуществляется посимвольное копирование строки "Привет!" в область памяти, на которую указывает str. Поскольку компилятор "не знает", что делает функция *strcpy*, он не считает эту ситуацию недопустимой.

Аналогично, если указатель на тип **const** присвоить указателю на тип, отличный от **const**, то через полученный указатель можно присвоить значение. Если же с помощью операции приведения типа преобразовать указатель на **const** к указателю на тип, отличный от **const**, то СП MSC, в отличие от СП TC, не позволит выполнить присваивание через преобразованный указатель.

Пример:

```
volatile int ticks;
void interrupt timer()
{
ticks ++;
}
wait(int interval)
{
ticks = 0;
while ( ticks < interval );
}
```

Функция **wait** будет "ждать" в течение времени, заданного параметром **interval** при условии, что функция **timer** корректно связана с аппаратным прерыванием от таймера. Значение переменной **ticks** изменяется в функции **timer** каждый раз при наступлении прерывания от таймера. Модификатор **interrupt** описан в разделе 3.3.3.5.

Если бы переменная **ticks** была объявлена без модификатора **volatile**, то компилятор языка Си с высоким уровнем оптимизации вынес бы за пределы цикла **while** сравнение переменных **ticks** и **interval**, поскольку в теле цикла их значения не изменяются. Это привело бы к зацикливанию программы.

3.3.3.3Модификаторы cdecl и pascal

Рассматриваемые системы программирования в языке Си позволяют обращаться из программы на языке Си к программам, написанным на других языках, и обратно. При смешивании языков программирования приходится иметь дело с двумя важными проблемами: написанием внешних имен и передачей параметров.

Результатом работы компилятора языка Си является файл, содержащий объектный код программы. Файлы с объектным кодом, полученные при компиляции всех исходных файлов, составляющих программу, компоновщик объединяет в один выполнимый файл. При этом производится так называемое разрешение ссылок на глобальные объекты из разных исходных файлов программы.

При компиляции все глобальные идентификаторы программы, т. е. имена функций и глобальных переменных, сохраняются в объектном коде и используются компоновщиком в процессе работы. По умолчанию эти идентификаторы сохраняются в своем первоначальном виде (т. е. набранные прописными, строчными буквами либо и теми, и другими). Кроме того, в качестве первого символа каждого идентификатора компилятор языка Си добавляет символ подчеркивания.

Компоновщик по умолчанию различает прописные и строчные буквы, поэтому идентификаторы, используемые в различных исходных файлах программы для именования одного и того же объекта, должны полностью совпадать с точки зрения, как орфографии, так и регистров клавиатуры. Для обеспечения совпадения идентификаторов, используемых в разноязычных исходных файлах, применяются модификаторы **pascal** и **cdecl**.

Модификатор pascal

Применение модификатора **pascal** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. Этот идентификатор может использоваться для именования в программе на языке Си глобального объекта, который используется также в программе на языке Паскаль. В объектном коде, сгенерированном компилятором языка Си, и в объектном коде, сгенерированном компилятором языка Паскаль, идентификатор будет представлен идентично.

Если модификатор **pascal** применяется к идентификатору функции, то он оказывает влияние также и на передачу аргументов. Засылка аргументов в стек производится в этом случае не в обратном порядке, как принято в компиляторах языка Си в СП MSC и СП TC, а в прямом—первым засылается в стек первый аргумент.

Функции типа **pascal** не могут иметь переменное число аргументов, как, например, функция **printf**. Поэтому нельзя использовать завершающее многоточие в списке параметров функции типа **pascal**.

Пример: см. пример 4 в разделе 3.3.3.4.

Модификатор cdecl

Существует опция компиляции, которая присваивает всем функциям и указателям на функции тип **pascal**. Это значит, что они будут использовать вызывающую

последовательность, принятую в языке Паскаль, а их идентификаторы будут приемлемы для вызова из программы на Паскале. При этом можно указать, что некоторые функции и указатели на функции используют вызывающую последовательность, принятую в языке Си, а их идентификаторы имеют традиционный вид для идентификаторов языка Си. Для этого их объявления должны содержать модификатор **cdecl**.

Примечание. Все функции в стандартных включаемых файлах (например, stdio.h) объявлены с модификатором **cdecl**. Это позволяет использовать библиотеки стандартных функций даже в тех программах, которые компилируются с упомянутой выше опцией компиляции.

Примечание. Главная функция программы (**main**) должна быть всегда объявлена с модификатором **cdecl**, поскольку модуль поддержки выполнения передает ей управление, используя вызывающую последовательность языка Си.

Пример: см. пример 3 в разделе 3.3.3.4.

3.3.3.4 Модификаторы near, far, huge

Эти модификаторы оказывают воздействие на работу с адресами объектов.

Компилятор языка Си позволяет использовать при компиляции одну из нескольких моделей памяти. Виды моделей памяти и методы их применения рассмотрены в разделе 8 "Модели памяти".

Модель, которую вы используете, определяет размещение в оперативной памяти вашей программы и данных, а также внутренний формат указателей. Однако при использовании какой-либо модели памяти можно объявить указатель с форматом, отличным от действующего по умолчанию. Это делается с помощью модификаторов **near**, **far** и **huge**.

Указатель типа **near** — 16-битовый; для определения адреса объекта он использует смещение относительно текущего содержимого сегментного регистра. Для указателя типа **near** доступная память ограничена размером текущего 64-килобайтного сегмента данных.

Указатель типа **far** — 32-битовый; он содержит как адрес сегмента, так и смещение. При использовании указателей типа **far** допустимы обращения к памяти в пределах 1-мегабайтного адресного пространства процессора Intel 8086/8088, однако значение указателя типа **far** циклически изменяется в пределах одного 64-килобайтного сегмента.

Указатель типа **huge** — 32-битовый; он также содержит адрес сегмента и смещение. Значение указателя типа **huge** может быть изменено в пределах всего 1-мегабайтного адресного пространства. В СП ТС указатель типа huge всегда хранится в нормализованном формате. Это имеет следующие следствия:

—операции отношения ==, !=, <, >, <=, >= выполняются корректно и предсказуемо над указателями типа **huge**, но не над указателями типа **far**;

—при использовании указателей типа **huge** требуется дополнительное время, т. к. программы нормализации должны вызываться при выполнении любой арифметической операции над этими указателями. Объем кода программы также возрастает.

В СП MSC модификатор **huge** применяется только к массивам, размер которых превышает 64 К. В СП ТС недопустимы массивы больше 64 К, а модификатор **huge** применяется к функциям и указателям для спецификации того, что адрес функции или указуемого объекта имеет тип **huge**.

Для вызова функции типа **near** используются машинные инструкции ближнего вызова, для типов **far** и **huge** — дальнего.

Примеры.

```
/* пример 1 */
int huge database [65000];
/* пример 2 */
char far *x;
/* пример 3 */
double near cdecl calc(double, double);
double cdecl near calc(double, double);
/* пример 4 */
char far pascal initlist[INITSIZE];
char far nextchar, far *prevchar, far *currentchar;
```

В первом примере объявляется массив с именем *database*, содержащий 65000 элементов типа **int**. Поскольку размер массива превышает 64 Кбайта, его описатель должен быть модифицирован специальным ключевым словом **huge**.

Во втором примере специальное ключевое слово *far* модифицирует расположенную справа от него звездочку, делая x указателем на **far** указатель на значение типа **char**. Это объявление можно для ясности записать и так:

```
char *(far *x);
```

В примере 3 показано два эквивалентных объявления. В них объявляется **calc** как функция с модификаторами near и **cdecl**.

В примере 4 также представлены два объявления. Первое объявляет массив типа **char** с именем **initiist** и модификаторами **far** и **pascal**. Модификатор **pascal** указывает на то, что имя данного массива используется не только в программе на языке Си, но и в программе на языке Паскаль (или другом языке программирования с подобными правилами написания имен внешних переменных). Модификатор **far** указывает на то, что для доступа к элементам массива долины использоваться 32-битовые адреса.

Второе объявление объявляет три указателя на **far** значения типа **char** с именами **nextchar**, **prevchar** и **currentchar**. Эти указатели могут быть, в частности, использованы для хранения адресов элементов массива

initlist. Обратите внимание на то, что специальное ключевое слово **far** должно быть повторено перед каждым описателем.

3.3.3.5Модификатор interrupt

Модификатор **interrupt** предназначен для объявления функций, работающих с векторами прерываний процессора 8086/8088. Для функции типа **interrupt** при компиляции генерируется дополнительный код в точке входа и выхода из функции, для сохранения и восстановления регистров микропроцессора AX, BX, CX, DX, SI, DI, ES и DS. Остальные регистры — BP, SP, SS, CS и IP сохраняются всегда как часть вызывающей последовательности языка Си или часть самой системы обработки прерывания.

См. пример в разделе 3.3.3.1.

Функции прерываний следует объявлять с типом возвращаемого значения **void**.

Функции прерываний поддерживаются для всех моделей памяти. В СП MSC, в малой и средней модели в регистр DS заносится при входе в функцию адрес сегмента данных всей программы, а в компактной, большой и максимальной модели в регистр DS заносится адрес сегмента данных программного модуля. В СП TC только в максимальной модели в регистр DS заносится адрес сегмента данных программного модуля, а в остальных моделях—адрес сегмента данных всей программы.

Модификатор **interrupt** не может использоваться совместно с модификаторами **near**, **far**, **huge**.

3.4 Объявление переменных

В этом разделе дано последовательное описание синтаксиса и семантики объявлений переменных. Разновидности переменных перечислены в следующей таблице:

Таблица 3.4.

Вид переменной	Пояснение
Простая переменная	Скалярная переменная целого или плавающего типа
Переменная перечислимого типа	Простая переменная целого типа, принимающая значения из предопределенного набора именованных значений
Структура	Переменная, содержащая совокупность элементов, которые могут иметь различные типы
Объединение	Переменная, содержащая совокупность элементов, которые могут иметь различные типы, но занимают одну и ту же область памяти
Массив	Переменная, содержащая совокупность элементов одинакового типа
Указатель	Переменная, которая указывает на другую переменную (содержит ее адрес)

Общая синтаксическая форма объявления переменных описана в начале раздела 3. В данном разделе для простоты изложения объявления описываются без спецификаций класса памяти и инициализаторов. Спецификации класса памяти описаны в разделе 3.6, инициализаторы — в разделе 3.7.

В объявлении простой переменной, массива и указателя спецификация типа может быть опущена. Если это объявление записано на внешнем уровне, то спецификация класса памяти тоже может быть опущена. В объявлении внутреннего уровня хотя бы одна из спецификаций — класса памяти или типа — должна присутствовать.

3.4.1 Объявление простой переменной

Синтаксис:

<*спецификация типа***>** **<***идентификатор*> [,**<***идентификатор*>…];

Объявление простой переменной определяет имя переменной и ее тип. Имя переменной задается <*идентификатором*>. <*Спецификация типа*> задает тип переменной. Тип может быть базовым типом, либо типом структура, либо типом объединение. Если спецификация типа опущена, предполагается тип **int**.

Можно объявить несколько переменных в одном объявлении, задавая список <*идентификаторов*>, разделенных запятыми. Каждый <*идентификатор*> в списке именует отдельную переменную. Все переменные, заданные в таком объявлении, имеют одинаковый тип.

Примеры.

```
int x; /* пример 1 */
unsigned long reply, flag; /* пример 2 */
double order; /* пример 3 */
```

В первом примере объявляется простая переменная **x**. Эта переменная может принимать любое значение из области значений типа **int**.

Во втором примере объявлены две переменные: **reply** и **flag**. Обе переменные имеют тип **unsigned long**.

В третьем примере объявлена переменная **order**, которая имеет тип **double**. Этой переменной могут быть присвоены значения с плавающей точкой.

3.4.2 Объявление переменной перечислимого типа

Синтаксис:

enum [**<***тег***>**]{**<***список-перечисления***>**} **<***описатель***>[**,<*описатель*>…];

enum <*тег*> <*идентификатор*> [<*идентификатор*>…];

Объявление переменной перечислимого типа задает имя переменной и определяет список именованных констант, называемый списком перечисления. Каждому элементу списка перечисления ставится в соответствие целое число. Переменная перечислимого типа может принимать только значения из своего списка перечисления. Элементы списка имеют тип **int**. Поэтому переменной перечислимого типа выделяется ячейка памяти, необходимая для размещения значения типа **int**. Перечислимый тип, таким образом, представляет собой подмножество целого типа. Над объектами перечислимого типа определены те же операции, что и над объектами целого типа.

<*Описатель*> специфицирует либо переменную перечислимого типа, либо указатель на значение перечислимого типа, либо массив элементов перечислимого типа, либо функцию, возвращающую значение перечислимого типа, либо более сложный объект, являющийся комбинацией перечисленных типов.

Объявление переменной перечислимого типа начинается с ключевого слова **enum** и имеет две формы представления.

В первой форме задается список перечисления, содержащий именованные константы. Необязательный <*тег*> — это идентификатор, который именует перечислимый тип, специфицированный данным списком перечисления, <*идентификатор*> — это имя переменной перечислимого типа. В одном объявлении может быть описано более одной переменной данного перечислимого типа.

Во второй форме объявления список перечисления отсутствует, однако используется <*тег*>, который ссылается на перечислимый тип, объявленный в другом месте программы. Если заданный тег ссылается на неизвестный перечислимый тип либо область действия определения этого перечислимого типа не распространяется на текущий блок, то компилятор языка Си сообщает об ошибке. Допускаются объявление указателя на перечислимый тип и объявление **typedef** для перечислимого типа, использующие тег ранее не определенного перечислимого типа. Однако этот тип должен быть определен к моменту использования этого тега или типа, объявленного посредством **typedef**.

<*Список-перечисления*> содержит одну или более конструкций вида:

<*идентификатор*> [=<*константное-выражение*>]

Конструкции в списке разделяются запятыми. Каждый <идентификатор> именует элемент списка перечисления. По умолчанию, если не задано <*константное-выражение*>, первому элементу присваивается значение 0, следующему элементу—значение 1 и т. д. Элемент списка перечисления является константой.

Запись =<*константное-выражение*> изменяет умалчиваемую последовательность значений. Элемент, идентификатор которого предшествует записи =<*константное-выражение*>, принимает значение, задаваемое этим константным выражением. Константное выражение должно иметь тип **int** и может быть как положительным, так и отрицательным. Следующий элемент списка получает значение, равное <*константное-выражение*>+1, если только его значение не задается явно другим константным выражением.

В списке перечисления могут содержаться элементы, которым сопоставлены одинаковые значения, однако каждый идентификатор в списке должен быть уникальным. Например, двум различным идентификаторам **null** и **zero** может быть задано значение 0 в одном и том же списке перечисления. Кроме того, идентификатор элемента списка перечисления должен быть отличным от идентификаторов элементов всех остальных списков перечислений с той же областью действия, а также от других идентификаторов с той же областью действия (см. раздел 2.5). Тег перечислимого типа должен быть отличным от тегов других перечислимых типов, структур и объединений с той же самой областью действия.

Примеры.

```
/* пример 1 */
enum day {
   SATURDAY,
   SUNDAY = 0,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
```

```
    THURSDAY,
    FRIDAY
} workday;

/* пример 2 */
enum day today = WEDNESDAY;
```

В первом примере перечислимый тип определяется списком. Перечислимый тип именуется тегом **day**, и объявляется переменная *workday* этого перечислимого типа. С элементом *SATURDAY* по умолчанию ассоциируется значение 0. Элементу *SUNDAY* явно присваивается значение ноль. Остальные идентификаторы по умолчанию принимают значение от 1 до 5.

Во втором примере переменная перечислимого типа **today** инициализируется значением одного из элементов списка перечисления. Так как перечислимый тип с тегом **day** был предварительно объявлен, то при объявлении **today** достаточно сослаться только на тег **day**, не записывая повторно сам список перечисления.

Примечание. В языке Си принято записывать имена элементов перечисления прописными буквами, однако это необязательно.

3.4.3 Объявление структуры

Структура позволяет объединить в одном объекте совокупность значений, которые могут иметь различные типы. Однако в языке Си реализован очень ограниченный набор операций над структурами как единым целым: передача функции в качестве аргумента, возврат в качестве значения функции, получение адреса. Можно присваивать одну структуру другой, если они имеют одинаковый тег.

Синтаксис:

struct [**<***тег***>**] {**<***список-объявлений-элементов***>**} **<***описатель***>** [,<*описатель*>…];

struct **<***тег***>** **<***описатель***>** [,**<***описатель***>**…];

Объявление структуры может задавать имя структурного типа и/или последовательность объявлений переменных, называемых элементами структуры. Эти элементы могут иметь различные типы.

Объявление структуры начинается с ключевого слова **struct** и имеет две формы записи, как показано выше. В первой форме типы и имена элементов структуры специфицируются в списке объявлений элементов. Необязательный в данном случае <*тег*> — это идентификатор, который именует структурный тип, определенный данным списком объявлений элементов.

<*Описатель*> специфицирует либо переменную структурного типа, либо указатель на структуру данного типа, либо массив структур данного типа, либо функцию, возвращающую структуру данного типа, либо более сложный объект, являющийся комбинацией перечисленных типов.

Вторая синтаксическая форма объявления использует тег структуры для ссылки на структурный тип, определенный где-то в другом месте программы. В этой форме объявления список объявлений элементов отсутствует. Объявление должно находиться в области действия данного тега, т. е. определение структурного типа, именованного тегом, должно предшествовать объявлению, использующему этот тег, за исключением двух случаев: когда тег используется для объявления либо указателя на структуру, либо структурного типа в **typedef**. Однако при этом определение структурного типа должно предшествовать использованию данного указателя либо типа, объявленного посредством **typedef**.

Список объявлений элементов представляет собой последовательность из одного или более объявлений переменных или битовых полей (см. ниже). Каждая переменная, объявленная в этом списке, называется элементом структуры. Особенность синтаксиса объявлений переменных в списке состоит в том, что они не могут содержать спецификаций класса памяти и инициализаторов. Элементы структуры могут иметь базовый тип, либо быть массивом, указателем, объединением или, в свою очередь, структурой.

Элемент структуры не может быть структурой того же типа, в которой он содержится. Однако он может быть объявлен как указатель на тип структуры, в которую он входит. Это позволяет создавать связанные списки структур.

Идентификаторы элементов структуры должны различаться между собой. Идентификаторы элементов разных структур могут совпадать. В пределах одной области действия тег структурного типа должен отличаться от тегов других структурных типов, тегов объединений и перечислимых типов.

Элементы структуры запоминаются в памяти последовательно в том порядке, в котором они объявляются: первому элементу соответствует меньший адрес памяти, а последнему — больший. Однако в СП ТС, если в одном объявлении содержатся описатели нескольких элементов, порядок их размещения в памяти будет обратным. Каждый элемент в памяти выровнен на границу, соответствующую его типу. Для микропроцессора Intel 8086/8088 это означает, что любой элемент, отличный от типа **char** или **unsigned char**, выравнивается на четную границу. Поэтому внутри структур могут появляться неименованные, пустые участки памяти между соседними элементами.

В версии 4.0 СП MSC элемент структуры, представляющий собой структуру нечетной длины, дополняется лишним байтом в конце, чтобы его длина стала четной. В версии 5.0 СП MSC это дополнение лишним байтом производится только в том случае, когда тип следующего элемента структуры требует его размещения с четного адреса.

В СП TC по умолчанию выравнивания в структурах не производится, однако существует опция компиляции, специфицирующая выравнивание. При этом обеспечивается следующее:

—структура будет начинаться на границе машинного слова (иметь четный адрес);

—любой элемент, имеющий тип, отличный от **char** или **unsigned char**, будет иметь четное смещение от начала структуры;

—чтобы структура содержала четное число байтов, в конец структуры будет при необходимости добавлен лишний байт.

Битовые поля

Битовые поля структур используются преимущественно в двух целях: для экономии памяти, поскольку позволяют плотно упаковать значения, и для организации удобного доступа к регистрам внешних устройств, в которых различные биты могут иметь самостоятельное функциональное назначение.

Объявление битового поля имеет следующий синтаксис:

<*спецификация типа*> [<*идентификатор*>]:<*константное выражение*>;

Битовое поле состоит из некоторого числа разрядов машинного слова. Число разрядов, т.е. размер битового поля, задается <*константным выражением*>. Константное выражение должно иметь неотрицательное целое значение. Это значение не может превышать числа разрядов, требуемого для представления значения специфицированного типа. Для битового поля в версия 4.0 СП MSC спецификация типа должна задавать беззнаковый целый тип (**unsigned int**). Для версии 5.0 СП MSC спецификация типа может задавать как знаковый, так и беззнаковый целый тип, причем любого размера — **char**, **int**, **long**. Однако знаковый целый тип для битовых полей реализован лишь синтаксически, а в выражениях битовые поля участвуют как беззнаковые значения. Недопустимы массивы битовых полей, указатели на битовые поля и функции, возвращающие битовые поля. Нельзя применять к битовым полям операцию адресации (&).

<*Идентификатор*> именует битовое поле. Его наличие, однако, необязательно. Неименованное битовое поле означает пропуск соответствующего числа битов перед размещением следующего элемента структуры. Неименованное битовое поле, для которого указан нулевой размер, имеет специальное назначение: оно гарантирует, что память для следующей переменной в этой структуре (в том числе и для следующего битового поля) будет начинаться на границе машинного слова (**int**). В версии 5.0 СП MSC выравнивание будет производиться на границу того типа, который задан для неименованного битового поля (**char**, **int** или **long**).

Битовое поле не может выходить за границу ячейки объявленного для него типа. Например, битовое поле, объявленное с типом **unsigned int**, упаковывается либо в пространство, оставшееся в текущей ячейке **unsigned int** от размещения предыдущего битового поля, либо, если предыдущий элемент структуры не был битовым полем или памяти в текущей ячейке недостаточно, в новую ячейку **unsigned int**.

В СП TC битовое поле может иметь либо тип **unsigned int**, либо тип **signed int**. Поля целого типа хранятся в дополнительном коде; крайний левый бит — знаковый. Например, битовое поле типа **signed int** размером 1 бит может только хранить значение -1 и 0, т.к. любое ненулевое значение будет интерпретироваться как -1.

Примеры:

```
/* пример 1 */
struct {
float x, y;
} complex;

/* пример 2 */
struct employee {
char name [20];
int id;
long class;
} temp;

/* пример 3 */
struct employee student, faculty, staff;

/* пример 4 */
struct sample {
char h; float *pf;
struct sample *next; )x;
```

```
/* пример 5 */
struct {
unsigned icon: 8;
unsigned color: 4;
unsigned underline: 1;
unsigned blink: 1;
} screen [25][80];
```

В первом примере объявляется переменная с именем *complex*, имеющая тип структура. Эта структура состоит из двух элементов *x* и *y* типа **float**. Тип структуры не поименован, поскольку тег в объявлении отсутствует.

Во втором примере объявляется переменная с именем *temp*, имеющая тип структура. Структура состоит из трех элементов с именами *name*, *id* и *class*. Элемент с именем *name* — это массив из 20 элементов типа **char**. Элементы с именами *id* и *class* — это простые переменные типа **int** и **long** соответственно. Структурный тип поименован тегом **employee**.

В третьем примере объявлены три переменные типа структура с именами *student*, *faculty* и *staff*. Объявление каждой из этих структур ссылается на структурный тип **employee**, определенный в предыдущем примере.

В четвертом примере объявляется переменная с именем *x* типа структура. Тип структуры поименован тегом **sample**. Первые два элемента структуры — переменная *h* типа char и указатель *pf* на значения типа **float**. Третий элемент с именем *next* объявлен как указатель на структуру того же самого типа **sample**.

В пятом примере объявляется двумерный массив с именем *screen*, элементы которого имеют структурный тип. Массив состоит из 2000 элементов. Каждый элемент — это отдельная структура, состоящая из четырех элементов — битовых полей с именами *icon*, *color*, *underline* и *blink*.

3.4.4 Объявление объединения

Объединение позволяет в разные моменты времени хранить в одном объекте значения различного типа. В процессе объявления объединения с ним ассоциируется набор типов значений, которые могут храниться в данном объединении. В каждый момент времени объединение может хранить значение только одного типа из набора. Контроль над тем, какого типа значение хранится в данный момент в объединении, возлагается на программиста. Синтаксис:

union [<*тег*>] {<*список-объявлений-элементов*>} <*описатель*> [,<*описатель*>…];

union **<***тег***>** **<***описатель*> [,**<***описатель***>**…];

Объявление объединения специфицирует его имя и совокупность объявлений переменных, называемых элементами объединения, которые могут иметь различные типы.

Объявление объединения имеет тот же синтаксис, что и объявление структуры, за исключением того, что оно начинается с ключевого слова *union*, а не с ключевого слова *struct*. Кроме того, СП MSC (в отличие от СП TC) не допускает в объединении битовые поля.

Память, которая выделяется переменной типа объединение, определяется размером наиболее длинного из элементов объединения. Все элементы объединения размещаются в одной и той же области памяти с одного и того же адреса. Значение текущего элемента объединения теряется, когда другому элементу объединения присваивается значение.

Примеры:

```
/* пример 1 */
union sign {
int svar;
unsigned uvar;
} number;

/* пример 2 */
union {
char *a, b;
float f[20];
} jack;
```

В первом примере объявляется переменная типа объединение с именем **number**. Список объявлений элементов объединения содержит две переменных: **svar** типа **int** и **uvar** типа **unsigned**. Это объединение позволяет запоминать целое значение в знаковом или беззнаковом виде. Тип объединения поименован тегом **sign**.

Во втором примере объявляется переменная типа объединение с именем **Jack**. Список объявлений элементов содержит три объявления: указателя **a** на значение типа **char**, переменной **b** типа **char** и массива **ж** из 20 элементов типа **float**. Тип объединения не поименован тегом. Память, выделяемая переменной **jack**, равна памяти, необходимой для хранения массива **f**, поскольку это самый длинный элемент объединения.

3.4.5 Объявление массива

Синтаксис:

[**<***спецификация типа*]**>** **<***описатель***>** [**<***константное выражение***>**];

[<*спецификация типа*]> **<***описатель***>** [];

Квадратные скобки, следующие за описателем, являются элементом языка Си, а не признаком необязательности синтаксической конструкции.

Массив позволяет хранить как единое целое последовательность переменных одинакового типа. Объявление массива определяет тип элементов массива и его имя. Оно может определять также число элементов в массиве. Переменная типа массив участвует в выражениях как константа - указатель на значение заданного спецификацией типа. Если спецификация типа опущена, предполагается тип **int**.

Объявление массива может иметь одну из двух синтаксических форм, указанных выше. Квадратные скобки, следующие за <*описателем*>, являются признаком типа массив. Если <*описатель*> представляет собой идентификатор (имя массива), то объявляется массив элементов специфицированного типа. Если же <*описатель*> представляет собой более сложную конструкцию (см. раздел 3.3.1), то каждый элемент массива имеет тип, заданный совокупностью <*спецификации типа*> и оставшейся части описателя. Это может быть любой тип, кроме типов **void** и функция. Таким образом, элементы массива могут иметь базовый, перечислимый, структурный тип, быть объединением, указателем или, в свою очередь, массивом.

Константное выражение, заключенное в квадратные скобки, определяет число элементов в массиве. Индексация элементов массива начинается с нуля. Таким образом, последний элемент массива имеет индекс на единицу меньше, чем число элементов в массиве.

Во второй синтаксической форме константное выражение в квадратных скобках опущено. Эта форма может быть использована, если в объявлении массива присутствует инициализатор, либо массив объявляется как формальный параметр функции, либо данное объявление является ссылкой на объявление массива где-то в другом месте программы. Однако для многомерного массива может быть опущена только первая размерность.

Многомерный массив, или массив массивов, объявляется путем задания последовательности константных выражений в квадратных скобках, следующей за описателем:

<*спецификация типа*> <*описатель*> [<*константное выражение*>] {<*константное выражение*>]…;

Каждое константное выражение в квадратных скобках определяет число элементов в данном измерении массива, поэтому объявление двумерного массива содержит два константных выражения, трехмерного — три и т. д.

Массиву выделяется память, которая требуется для размещения всех его элементов. Элементы массива с первого до последнего размещаются в последовательных ячейках памяти, по возрастанию адресов. Между элементами массива в памяти разрывы отсутствуют. Элементы многомерного массива запоминаются построчно. Например, массив, представляющий собой матрицу размером две строки на три столбца

char a[2][3]

будет храниться следующим образом: сначала в памяти запоминаются три элемента первой строки, затем три элемента второй строки. При таком методе хранения последний индекс массива меняется быстрее предпоследнего. Для доступа к отдельному элементу массива используется индексное выражение, которое описано в разделе 4.2.5 "Индексные выражения".

Примеры.

```
/* пример 1 */
int scores[10], game:
/* пример 2 */
float matrix[10][15];
/* пример 3 */
struct {
float x, y;
} complex[100];
/* пример 4 */
char *name[20];
```

В первом примере объявляется переменная типа массив с именем **scores** из 10 элементов типа **int**. Переменная с именем **game** объявлена как простая переменная целого типа.

Во втором примере объявляется двумерный массив с именем **matrix**. Строго говоря, **matrix** представляет собой массив, состоящий из 10 элементов, каждый из которых является массивом из 15 элементов типа **float**.

В третьем примере объявляется массив структур типа **complex**. Он состоит из 100 элементов. Каждый элемент массива представляет собой структуру, содержащую два элемента типа **float**.

В четвергом примере объявлен массив указателей. Массив содержит 20 элементов, каждый из которых является указателем на значение типа **char**.

3.4.6 Объявление указателя

Указатель — это переменная, предназначенная для хранения адреса объекта некоторого типа. Указатель на функцию содержит адрес точки входа в функцию.

Синтаксис:

[<*спецификация типа*]> *<*описатель*>;

Объявление указателя специфицирует имя переменной-указателя и тип объекта, на который может указывать эта переменная. Спецификация типа может задавать базовый, перечислимый, пустой, структурный тип или тип объединение. Если спецификация типа опущена, предполагается тип **int**.

Если <*описатель*> представляет собой идентификатор (имя указателя), то объявляется указатель на значение специфицированного типа. Если же <*описатель*> представляет собой более сложную конструкцию (см. раздел 3.3.1), то тип объекта, на который указывает указатель, определяется совокупностью оставшейся части описателя и спецификации типа. Указатель может указывать на значения базового, перечислимого типа, структуры, объединения, массивы, функции, указатели.

Специальное применение имеют указатели на тип **void**. Указатель на **void** может указывать на значения любого типа. Однако для выполнения операций над указателем на **void** либо над указуемым объектом, необходимо явно привести тип указателя к типу, отличному от **void**. Например, если объявлена переменная *i* типа **int** и указатель *p* на тип **void**

```
int i;
void *p;
```

то можно присвоить указателю p адрес переменной i

```
p = &i;
```

но изменить значение указателя нельзя. В СП ТС нельзя также получить значение указуемого объекта по операции косвенной адресации (в СП MSC в этом случае выдается предупреждающее сообщение).

```
p++; /* недопустимо */
(int *)p++; /* допустимо */
j = *p; /* недопустимо в СП ТС */
```

Можно объявить функцию с типом возвращаемого значения указатель на **void**. Ее значение может быть присвоено указателю на тот тип, который требуется.

Переменная, объявленная как указатель, хранит адрес памяти. Размер памяти, требуемый для адреса, и формат этого адреса зависит от компьютера и реализации компилятора языка Си. Указатели на один и тот же тип данных не обязательно имеют одинаковый размер и формат, поскольку эти параметры зависят от выбранной модели памяти. Кроме того, существуют модификаторы **near**, **far**, **huge**, специфицирующие формат указателя. Объявления, использующие эти модификаторы, рассмотрены в разделе 3.3.3.4.

Указатель на структуру, объединение или перечислимый тип может быть объявлен до того, как этот тип определен, однако указатель не должен использоваться до определения этого типа. Указатель при этом объявляется посредством использования тега структуры, объединения или перечислимого типа (см. ниже пример 4). Такие объявления допускаются, поскольку компилятору языка Си не требуется знать размер структуры или объединения, чтобы распределить память под указатель.

В стандартном включаемом файле **stdio.h** определена константа с именем NULL. Она предназначена для инициализации указателей. Гарантируется, что никакой программный объект никогда не будет иметь адрес NULL.

Примеры.

```
char *message;/* пример 1 */
im *array1 [10]; /* пример 2 */
int (*pointer1)[10];/* пример 3 */
struct list *next, *previous; /* пример 4 */
struct list {/* пример 5 */
   char *token;
   int *count;
   struct list *next;
} line;
struct id {/* пример 6 */
   unsigned int id_no;
   struct name *pname;
} record;
```

В первом примере объявляется указатель с именем **message**. Он указывает на значения типа **char**.

Во втором примере объявлен массив указателей с именем **array1**. Массив состоит из 10 элементов. Каждый элемент представляет собой указатель на значения типа **int**.

В третьем примере объявлен указатель с именем **pointer1**. Он указывает на массив из 10 элементов. Каждый элемент этого массива имеет тип **int**.

В четвертом примере объявлены два указателя, которые указывают на объекты структурного типа, именованного тегом **list** (см. следующий пример). Определение типа **list** должно либо предшествовать данному объявлению, либо находиться в пределах области действия данного объявления.

В пятом примере объявляется структура с именем **line**, тип которой поименован тегом **list**. Структурный тип **list** содержит три элемента. Первый элемент — указатель на значение типа **char**, второй — указатель на значение типа **int**, третий — указатель на структуру типа **list**.

В шестом примере объявляется структура с именем **record**, тип которой поименован тегом **id**. Обратите внимание на то, что элемент с именем **pname** объявлен как указатель на другой структурный тип с тегом **name**.

Не считается ошибкой появление этого объявления в программе раньше, чем будет объявлен тег **name** (но тип **name** должен быть объявлен до первого использования указателя **pname** в выражении).

3.5 Объявление функции (прототип)

Метод объявления функции, описанный в данном разделе, используется только в версии 4.0 СП MSC. В версии 5.0 СП MSC, а также в СП TC реализован более современный метод — объявление прототипа функции, а старый метод поддерживается в этих версиях лишь для совместимости программ. В конце данного раздела приведены основные отличия метода объявления прототипа.

Синтаксис:

[<*спецификация класса памяти*>][<*спецификация типа*>] <*описатель*>([<*список типов аргументов*>]);

Объявление функции специфицирует имя функции, тип возвращаемого значения и, возможно, типы ее аргументов и их число. Эти атрибуты функции необходимы для проверки компилятором языка Си корректности обращения к ней до того, как она определена. Определение функций рассмотрено в разделе 6.2.

Если <*описатель*> функции представляет собой идентификатор (имя функции), то объявляется функция, тип возвращаемого значения которой задан спецификацией типа. Если же <*описатель*> представляет собой более сложную конструкцию (см. раздел 3.3.1), то оставшаяся часть описателя в совокупности со <*спецификацией типа*> задает тип возвращаемого значения. Функция не может возвращать массив или функцию, но может возвращать указатель на эти объекты.

Если спецификация типа в объявлении функции опущена, то предполагается тип **int**. На внешнем уровне может быть также опущена спецификация класса памяти, а на внутреннем уровне хотя бы одна из спецификаций — класса памяти или типа—должна присутствовать.

В объявлении функции можно задать спецификацию класса памяти **extern** или **static**. Классы памяти рассматриваются в разделе 3.6.

Список типов аргументов

Список типов аргументов определяет типы аргументов функции и их число.

Список типов — это список из одного или более имен типов. Каждое имя типа отделяется от другого запятой. Список ограничивается круглыми скобками.

Первое имя типа задает тип первого аргумента, второе имя задает тип второго аргумента и т.д. Концом списка является закрывающая круглая скобка, однако перед ней может быть записана запятая и многоточие (,…). Это означает, что число аргументов функции переменно, но не меньше, чем имен типов, заданных до многоточия.

Если список типов аргументов содержит только многоточие (…), то число аргументов функции является переменным и может быть равным нулю.

Примечание. Для совместимости с программами предыдущих версий допускается символ запятой без многоточия в конце списка типов аргументов для обозначения того, что число аргументов переменно. Запятая также может быть использована вместо многоточия как признак того, что функция имеет нуль или более аргументов. Для новых программ рекомендуется использование многоточия.

Имя типа для базового, перечислимого типа, структуры или объединения представляет собой спецификацию этого типа (например, **int**). Имена типов для указателей и массивов формируются путем комбинации спецификации типа с "абстрактным описателем". Абстрактный описатель—это описатель, в котором опущен идентификатор. В разделе 3.8.3 "Имена типов" объясняется, каким образом формировать и интерпретировать абстрактные описатели.

Для того чтобы объявить функцию, не имеющую аргументов, рекомендуется записать ключевое слово *void* на месте списка типов аргументов. Компилятор языка Си выдает предупреждающее сообщение, если в вызове такой функции будут указаны аргументы (однако для этого вызов функции должен находиться в области действия данного объявления).

В списке типов аргументов в качестве имени типа допускается также конструкция *void**, которая специфицирует аргумент типа "указатель на любой тип".

Список типов аргументов может быть пуст, однако скобки после идентификатора функции все же обязательны. В этом случае в объявлении функции не специфицированы ни типы, ни число аргументов функции. Следовательно, компилятор языка Си не может проверить соответствие типов аргументов при вызове функции. Несоответствие типов аргументов может привести к трудно выявляемым ошибкам во время выполнения программы. Более подробная информация о правилах соответствия типов аргументов приведена в разделе 6.4 "Вызов функции".

Примеры:

```
int add (int, int);                    /* пример 1 */
double calc();                         /* пример 2 */
```

```
char *strfind (char *, …);            /* пример 3 */
void draw(void);                      /* пример 4 */
double (*sum (double, double))[3];    /* пример 5 */
int (*select(void))(int);             /* пример 6 */
char *p;                              /* пример 7 */
short *q;
int prt(void *);
fff(int);                             /* пример 8 */
```

В первом примере объявляется функция *с* именем *add,* которая принимает два аргумента типа **int** и возвращает значение типа **int**.

Во втором примере объявляется функция с именем *calc,* которая возвращает значение типа **double**. Список типов аргументов пуст.

В третьем примере объявляется функция с именем *strfind,* которая возвращает указатель на значение типа **char**. Функция требует по крайней мере один аргумент—указатель на значение типа **char** Список типов аргументов заканчивается запятой и многоточием. Это значит, что функция может принять и большее число аргументов.

В четвертом примере объявляется функция с типом возвращаемого значения **void** (ничего не возвращающая). Список типов аргументов также содержит ключевое слово *void,* означающее отсутствие аргументов функции.

В пятом примере *sum* объявляется как функция, возвращающая указатель на массив из трех значений типа **double**. Функция *sum* требует два аргумента, каждый из которых имеет тип **double**.

В шестом примере функция с именем *select* объявлена как не имеющая аргументов и возвращающая указатель на функцию, требующую один аргумент типа **int** и возвращающую значение типа **int**.

В седьмом примере объявлена функция *prt,* которая принимает в качестве аргумента указатель на любой тип и возвращает значение типа **int**. Любой из указателей p и q мог бы быть вполне корректно использован в качестве аргумента функции.

В восьмом примере объявлена функция *fff*, принимающая один аргумент типа **int** и возвращающая (по умолчанию) значение типа **int**. Очевидно, что эта функция объявлена на внешнем уровне, поскольку в ее объявлении отсутствует и спецификация класса памяти, и спецификация типа.

Далее рассмотрим отличия метода объявления прототипов функций. В списке типов аргументов прототип может содержать также и идентификаторы этих аргументов. Они необязательны, их область действия ограничивается только прототипом, в котором они определены. Следовательно, необязательно именовать их так же, как формальные параметры в определении функции. Основное назначение использования идентификаторов аргументов в прототипе — повышение читабельности программы. Например, стандартная функция копирования строк **strcpy** имеет два аргумента: исходную строку и результирующую строку. Чтобы не перепутать их, можно объявить прототип функции

```
char *strcpy (char *result, char *ishod);
```

Идентификатор, указанный в объявлении, используется только в диагностическом сообщении компилятора языка Си, в случае несоответствия типов аргументов в вызове функции типам ее формальных параметров в прототипе.

Файлы стандартного заголовка СП MSC версии 5.0 и СП TC содержат объявления прототипов стандартных библиотечных функций. Вы можете распечатать эти файлы, и практически вся информация, необходимая для обращения к функциям, будет у Вас под рукой.

Еще одно отличие метода объявления прототипов состоит в том, что объявление аргумента в прототипе может содержать спецификацию класса памяти **register**.

3.6 Классы памяти

Спецификация класса памяти переменной определяет, какое время жизни она имеет (глобальное или локальное), и влияет на область действия переменной. Объект с глобальным временем жизни существует и имеет значение на протяжении всего времени выполнения программы. Все функции имеют глобальное время жизни.

Переменной с локальным временем жизни выделяется новая ячейка памяти каждый раз, когда управление передается блоку, в котором она определена. Когда управление возвращается из блока, переменная теряет свое значение.

В языке Си имеется четыре спецификации класса памяти:

```
auto
register
static
extern
```

Область действия функций, объявленных со спецификацией класса памяти **extern**, распространяется на все исходные файлы, которые составляют программу; следовательно, такие функции могут быть вызваны из любой функции в любом исходном файле программы.

Переменные классов памяти **auto** и **register** имеют локальное время жизни. Спецификации **static** и **extern** определяют объекты с глобальным временем жизни.

В совокупности с местоположением объявления объекта спецификация класса памяти определяет область действия переменной или функции. Термин "область действия" определяет часть программы, в которой к функции или переменной возможен доступ. Например, переменная с глобальным временем жизни существует в течение всего времени

выполнения исходной программы, но она может быть доступна не во всех частях программы. Область действия и связанное с ней понятие времени жизни рассмотрены в разделе 2.4.

Объявления, расположенные вне тел всех функций, относятся к внешнему уровню, а объявления внутри тел функций относятся к внутреннему уровню. Особый случай представляют объявления формальных параметров функции. В последующих разделах описывается смысл спецификаций класса памяти для каждого варианта объявления, а также поясняются правила умолчания в случае отсутствия в объявлении спецификации класса памяти.

Функции могут быть объявлены со спецификацией класса памяти **static** или **extern** либо вообще без спецификации класса памяти. Функции всегда имеют глобальное время жизни.

Объявления функций, в которых опущена спецификация класса памяти, аналогичны объявлениям со спецификацией класса памяти **extern**.

Если в объявлении функции специфицирован класс памяти **static**, то и в ее определении должен быть также указан класс памяти **static** (это требование не является обязательным для СП ТС).

Объявление функции на внутреннем уровне по смыслу эквивалентно объявлению внешнего уровня, т. е. область действия функции распространяется не до конца блока, а до конца файла.

Область действия функций для различных спецификаций класса памяти рассмотрена подробно в разделе 2.4 "Время жизни и область действия".

3.6.1 Объявление переменной на внешнем уровне

Объявления переменной на внешнем уровне используют спецификации класса памяти **static** и **extern** или вообще опускают их. Спецификации класса памяти **auto** и **register** не допускаются на внешнем уровне.

Объявления переменных на внешнем уровне—это либо определения переменных, либо объявления, т.е. ссылки на определения, сделанные в другом месте.

Определение внешней переменной—это объявление, которое вызывает выделение памяти для этой переменной и инициализирует ее (явно или неявно). Определение на внешнем уровне может задаваться в следующих различных формах:

1) Переменная может быть определена путем ее объявления со спецификацией класса памяти **static**. Такая переменная может быть явно инициализирована константным выражением. Если инициализатор отсутствует, то переменная автоматически инициализируется нулевым значением во время компиляции. Таким образом, каждое из объявлений:

```
static int k = 16;
```

и

```
static int k;
```

рассматривается как определение.

2) Переменная может быть определена, если спецификация класса памяти в ее объявлении опущена, и переменная явно инициализируется, например,

```
int j = 3;
```

Область действия переменной, определенной на внешнем уровне, распространяется от точки, где она определена, до конца исходного файла. Переменная недоступна выше своего определения в том же самом исходном файле. На другие исходные файлы программы область действия переменной распространяется лишь в том случае, если ее определение не содержит спецификации класса памяти **static** и если в других исходных файлах имеется ее объявление.

Если в объявлении переменной задана спецификация класса памяти **static**, то в других исходных файлах могут быть определены другие переменные с тем же именем и любым классом памяти. Эти переменные никак не будут связаны между собой, поскольку каждое определение **static** доступно только в пределах своего исходного файла.

Спецификация класса памяти **extern** используется для объявления переменной, определенной где-то в другом месте программы. Такие объявления используются в случае, когда нужно распространить на данный исходный файл область действия переменной, определенной в другом исходном файле, либо сделать переменную доступной в том же исходном файле выше ее определения. Область действия переменной распространяется от места объявления до конца исходного файла.

В объявлениях, которые используют спецификацию класса памяти **extern**, инициализация не допускается, так как они ссылаются на переменные, значения которых определены в другом месте.

Каждая переменная внешнего уровня обязательно должна быть определена один и только один раз в каком-либо из исходных файлов, составляющих программу.

Существует одно исключение из правил, описанных выше. Можно опустить в объявлении переменной на внешнем уровне и спецификацию класса памяти, и инициализатор. Например, объявление **int n;** будет вполне корректным внешним объявлением. Это объявление имеет различный смысл в зависимости от контекста:

1) Если в каком-то другом исходном файле программы (возможно, в другом исходном файле) есть определение на внешнем уровне переменной с таким же именем, то данное объявление является ссылкой на это определение. В этом случае объявление аналогично объявлению со спецификацией класса памяти **extern**.

2) Если же такого определения переменной в программе нет, то данное объявление само считается определением переменной. На этапе компоновки программы переменной выделяется память, которая инициализируется нулевым значением. Если в программе имеется более одного объявления переменной с одним и тем же именем, то размер выделяемой памяти будет равен размеру наиболее длинного типа среди всех объявлений этой переменной. Например, если программа содержит два неинициализированных объявления переменной **i** на внешнем уровне **int i;** и **char i;**, то память будет выделена под переменную **i** типа **int.**

Примечание. В описании языка Си, данном его разработчиками в [1], отсутствовала ясная трактовка понятий объявления и определения глобальной переменной. Это привело к тому, что различные компиляторы языка Си используют различные схемы разбора подобных ситуаций. Схема разбора, описанная в данном разделе, рассматривает глобальную переменную как общий блок, разделяемый несколькими исходными файлами. Глобальная переменная фактически представляет собой единую область памяти, которая разделяется несколькими исходными файлами, причем в каждом из них переменная может иметь различный тип.

Рекомендуется всегда инициализировать объявления переменных на внешнем уровне в файлах, которые предназначены для помещения в библиотеку. Это повышают вероятность выявления случаев нежелательного совпадения имен внешних переменных в библиотечном файле и пользовательской программе. Если переменная в программе пользователя также инициализирована, то компоновщик обнаружит два инициализированных объявления одной и той же глобальной переменной и сообщит об ошибке.

Возможно наличие в одном исходном файле на внешнем уровне нескольких объявлений переменной с одним и тем же именем. Следующая таблица позволяет определить реакцию компилятора языка Си в различных ситуациях изменения спецификации класса памяти в объявлении. Слово "пусто" в таблице означает ситуацию отсутствия спецификации класса памяти. Очевидно, что компилятор СП MSC строже ограничивает возможность переопределения класса памяти переменной.

Класс 1	Класс 2	СП TC	СП MSC
extern	static	static	static
static	пусто	static	ошибка
static	extern	static	static
пусто	static	static	ошибка

Пример:
```
/* ИСХОДНЫЙ ФАЙЛ 1 */
/* объявление i, ссылающееся на данное ниже определение i */
extern int i;
main()
{
i = i + 1;
printf("%d\n", i);            /* значение i равно 4 */
next();
}
int i = 3;                    /* определение i */

next()
{
printf("%d\n", i);            /* значение i равно 5 */
other());
}
/* ИСХОДНЫЙ ФАЙЛ 2 */
/* объявление i, ссылающееся на определение i в первом исходном файле */
extern int i;
other()
{
i = i + 1;
printf("%d\n", i);            /* значение i равно 6 */
}
```

Два исходных файла в совокупности содержат три внешних объявления **i**. Только в одном объявлении содержится инициализация:

int i = 3; — глобальная переменная **i** определена с начальным значением 3.

Самое первое объявление **extern** в первом исходном файле делает глобальную переменную **i** доступной прежде ее определения в файле. Без этого объявления функция **main** не могла бы использовать глобальную

переменную **i**. Объявление переменной **i** во втором исходном файле делает глобальную переменную **i** доступной во втором исходном файле.

Все три функции выполняют одно и то же действие: увеличивают **i** на 1 и печатают полученное значение. Значения распечатываются с помощью стандартной библиотечной функции **printf**. Печатаются значения 4, 5 и 6.

Если бы переменная **i** не была инициализирована ни в одном из объявлений, она была оы неявно инициализирована нулевым значением при компоновке. В этом случае программа напечатала бы значения 1, 2 и 3.

3.6.2 Объявление переменной на внутреннем уровне

Любая из четырех спецификаций класса памяти может быть использована для объявления переменной на внутреннем уровне. Если спецификация класса памяти опущена в объявлении переменной на внутреннем уровне, то подразумевается класс памяти **auto**. Как правило, ключевое слово **auto** опускается. Понятия объявления и определения для переменных внутреннего уровня совпадают, если только в объявлении не задана спецификация класса памяти **extern**.

Спецификация класса памяти **auto** объявляет переменную с локальным временем жизни. Область действия переменной распространяется на блок, в котором она объявлена, (и на все вложенные в него блоки). Переменные класса памяти **auto** автоматически не инициализируются, поэтому в случае отсутствия инициализации в объявлении значение переменной класса памяти **auto** считается неопределенным. Память под переменные класса памяти **auto** отводится в стеке.

Спецификация класса памяти **register** требует, чтобы компилятор языка Си выделил переменной память в регистре микропроцессора, если это возможно. Использование регистровой памяти обычно ускоряет доступ к переменной и уменьшает размер выполняемого кода программы. Переменные, объявленные с классом памяти **register**, имеют ту же самую область действия, что и переменные **auto**.

Число регистров, которое может быть использовано для хранения переменных, зависит от компьютера и от реализации компилятора языка Си. Если компилятор языка Си обнаруживает спецификацию класса памяти **register** в объявлении переменной, а свободного регистра не имеется, или переменная данного типа не может быть размещена в регистре, то переменной выделяется память класса **auto**. В СП MSC регистровая память всегда выделяется переменным в том порядке, в котором они объявляются в исходном файле. В СП TC, при наличии нескольких переменных класса памяти **register** в одном объявлении, регистровая память будет выделяться переменным в обратном порядке. Так, по объявлению **register i, j;** первой получит регистровую память переменная **j**.

В регистровой памяти может быть размещен объект размером не больше, чем тип **int**. К переменной, размещенной в регистре, нельзя применять операцию адресации. При вызове функций из блока, в котором определены регистровые переменные, содержимое регистров будет сохранено в памяти, а по возвращении в блок восстановлено.

Для каждого рекурсивного входа в блок порождается новый набор переменных класса памяти **auto** и **register**. При этом каждый раз производится инициализация переменных, в объявлении которых заданы инициализаторы.

Переменная, объявленная на внутреннем уровне со спецификацией класса памяти **static**, имеет глобальное время жизни, но ее область действия распространяется только на блок, в котором она объявлена (и на все вложенные блоки). В отличие от переменных класса памяти **auto**, переменные, объявленные со спецификацией класса памяти **static**, сохраняют свое значение при выходе из блока. Переменные класса памяти **static** могут быть инициализированы константным выражением. Если явной инициализации нет, то переменная класса памяти **static** автоматически инициализируется нулевым значением. Инициализация выполняется один раз во время компиляции и не повторяется при каждом входе в блок. Все рекурсивные вызовы данного блока будут разделять единственный экземпляр переменной класса памяти **static**.

Переменная, объявленная со спецификацией класса памяти **extern**, является ссылкой на переменную с тем же самым именем, определенную на внешнем уровне в любом исходном файле программы. Цель внутреннего объявления **extern** состоит в том, чтобы сделать определение переменной внешнего уровня (как правило, данное в другом исходном файле) доступным именно внутри данного блока. Внутреннее объявление **extern** не влияет на область действия объявляемой глобальной переменной в любой другой части программы.

Пример:

```
inl i = 1; /* определение i */
main()
{
/* объявление i, ссылающееся на данное выше определение */
extern int i;
/* начальное значение a равно нулю; область действия a — функция main */
static int a;
/* b будет (по возможности) помещено в регистр */
register int b = 0;
```

```
/* по умолчанию c будет иметь класс памяти auto */
int c = 0;
/* печатаются значения 1, 0, 0, 0*/
printf("%d,%d,%d,%d\n", i, a, b, c);
}
other()
/* локальное переопределение переменной i */
int i = 16;
/* область действия переменной a — функция other */
static int a = 2;
a += 2;
/* печатаются значения 16, 4 */
printf("%d,%d\n", i, a);
}
```

Переменная i определяется на внешнем уровне с начальным значением 1; В функции **main** объявление i является ссылкой на определение переменной i внешнего уровня. Эта ссылка необязательна, поскольку и без нее внешняя переменная i доступна во всех функциях данного исходного файла. Переменная a класса памяти **static** автоматически инициализируется нулевым значением, так как явная инициализация опущена. Определяется переменная b регистрового класса памяти и переменная c класса памяти **auto**. Вызывается стандартная функция **printf**, которая печатает значения 1, 0, 0, 0.

В функции **other** переменная i переопределяется как локальная переменная с начальным значением 16. Это не влияет на значение внешней переменной i, поскольку эти переменные никак не связаны между собой. Переменная a объявляется со спецификацией класса памяти **static** и начальным значением 2. Она никак не связана с переменной a, объявленной в функции **main**, так как область действия переменных класса памяти **static** на внутреннем уровне ограничена блоком, в котором они объявлены. Значение переменной a увеличивается на 2 и становится равным 4. Если бы функция **other** была вызвана еще раз в той же функции **main**, то значение a при входе было бы равно 4, а при выходе—6. Внутренние переменные класса памяти **static** сохраняют свои значения при входе в блок и выходе из блока, в котором они объявлены. Значение переменной a в функции **main** при этом не изменилось бы.

3.7 Инициализация

Переменной в объявлении может быть присвоено начальное значение посредством инициализатора. Записи инициализатора в объявлении предшествует знак равенства

=<*инициализатор*>

Можно инициализировать переменные любого типа. Функции не инициализируются. Объявления, которые используют спецификацию класса памяти **extern**, не могут содержать инициализатор.

Переменная, объявленная на внешнем уровне без спецификации класса памяти, может быть инициализирована не более одного раза в каком-либо из исходных файлов, составляющих программу. Если же она явно не инициализирована ни в одном из исходных файлов, то компоновщик инициализирует ее нулевым значением.

Переменная класса памяти **static**, объявленная как на внешнем, так и на внутреннем уровне, может быть инициализирована константным выражением не более одного раза в исходном файле. Если ее явная инициализация отсутствует, то компилятор языка Си инициализирует ее нулевым значением.

Инициализация переменных класса памяти **auto** и **register** выполняется каждый раз при входе в блок (за исключением входа в блок по оператору **goto**), в котором они объявлены. Если инициализатор опущен в объявлении переменной класса памяти **auto** или **register**, то ее начальное значение не определено. Инициализация переменных составных типов (массив, структура, объединение), имеющих класс памяти **auto**, запрещена в СП MSC, но допускается в СП TC даже для переменных, объявленных с модификатором **const**. Переменные составного типа, имеющие класс памяти **static**, могут быть инициализированы на внутреннем уровне.

Инициализирующими значениями для переменных внешнего уровня, а также переменных класса памяти **static** внутреннего уровня должно быть константное выражение (см. раздел 4.2.9). Переменные классов памяти **auto** и **register** могут быть инициализированы не только константными выражениями, но и выражениями, содержащими переменные и вызовы функций.

3.7.1 Базовые типы и указатели

Синтаксис:

=<*выражение***>**

Значение выражения присваивается переменной. При необходимости выполняются правила преобразования типов.

Примеры:

```
int x = 10, y = 20;        /* пример 1 */
register int *px = 0;      /* пример 2 */
int c = (3*1024);          /* пример 3 */
int *b = &x;               /* пример 4 */
```

В первом примере переменная **x** инициализируется константным выражением 10, переменная **y** инициализируется константным выражением 20. Во втором примере указатель **px** инициализирован нулевым

значением. В третьем примере используется константное выражение для инициализации переменной **c**. В четвертом примере указатель **b** инициализируется адресом переменной **x**.

3.7.2 Составные типы

Элементы объектов составных типов инициализируются только константными выражениями.

Инициализация объектов составных типов имеет следующий синтаксис:

= {<*список инициализаторов*>}

Список инициализаторов представляет собой последовательность инициализаторов, разделенных запятыми. Список инициализаторов заключается в фигурные скобки. Каждый инициализатор в списке представляет собой либо константное выражение, либо, в свою очередь, список инициализаторов. Таким образом, заключенный в фигурные скобки список может появиться внутри другого списка инициализаторов. Эта конструкция используется для инициализации тех элементов объектов составных типов, которые сами имеют составной тип.

Значения константных выражений из каждого списка инициализаторов присваиваются элементам объекта составного типа в порядке их следования.

Для инициализации объединения список инициализаторов должен содержать единственное константное выражение. Значение этого константного выражения присваивается первому элементу объединения. В СП ТС не обязательно заключать это константное выражение в фигурные скобки.

Наличие списка инициализаторов в объявлении массива позволяет не указывать число элементов по его первой размерности. В этом случае количество элементов в списке инициализаторов и определяет число элементов по первой размерности массива. Тем самым определяется размер памяти, необходимой для хранения массива. Число элементов по остальным размерностям массива, кроме первой, указывать обязательно.

Если в списке инициализаторов меньше элементов, чем в объекте составного типа, то оставшиеся элементы объекта неявно инициализируются нулевыми значениями. Если же число инициализаторов больше, чем требуется, то выдается сообщение об ошибке. Эти правила применяются и к каждому вложенному списку инициализаторов.

Пример 1:

```
int p[4][3] =
{
   {1, 1, 1},
   {2, 2, 2},
   {3, 3, 3,},
   {4, 4, 4,},
};
```

В примере объявляется двумерный массив p, размером 4 строки на 3 столбца, элементы первой строки инициализируются единицами, второй строки — двойками и т. д. Обратите внимание на то, что списки инициализаторов двух последних строк содержат в конце запятую. За последним списком инициализаторов (4,4,4,) также стоит запятая. Эти дополнительные запятые не требуются, но допускаются. Требуются только те запятые, которые разделяют константные выражения и списки инициализаторов. Если список инициализаторов не имеет вложенной структуры, аналогичной структуре объекта составного типа, то элементы списка присваиваются элементам объекта в порядке следования. Поэтому вышеприведенная инициализация эквивалентна следующей:

```
int p[4][3] = {1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4};
```

При инициализации объектов составных типов нужно внимательно следить за правильностью расстановки фигурных скобок в списках инициализаторов. В следующем примере этот вопрос иллюстрируется более детально.

Пример 2.

```
struct {
int n1, n2, n3;
} nlist[2][3] = {
{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}), /* строка 1 */
{{10,11,12}, {13,14,15}, {15,16,17}} /* строка 2 */
}
```

В примере переменная **nlist** объявляется как двумерный массив структур, состоящий из двух строк и трех столбцов. Каждая структура содержит три элемента. В строке 1 значения присваиваются первой строке массива **nlist** следующим образом:

1) Первая левая фигурная скобка строки 1 информирует компилятор языка Си о том, что начинается инициализация первой строки массива **nlist** (т. е. **nlist**[0]).

2) Вторая левая фигурная скобка означает, что начинается инициализация первого элемента первой строки массива (т. е. **nlist**[0][0]).

3) Первая правая фигурная скобка сообщает об окончании инициализации структуры **nlist**[0][0]. Следующая левая фигурная скобка сообщает о начале инициализации второго элемента первой строки **nlist**[0][1].

4) Процесс инициализации элементов подмассива **nlist**[0] продолжается до конца строки 1 и заканчивается по последней правой фигурной скобке.

Аналогично, в строке 2 присваиваются значения второй строке массива **nlist**, т. е. **nlist**[1].

Следует понимать, что фигурные скобки, охватывающие инициализаторы строки 1 и строки 2, необходимы. Следующая конструкция, в которой внешние фигурные скобки опущены, неверна.

```
struct {
int n1, n2, n3;
} nlist[2][3] = {
```

```
{1, 2, 3}, {4, 5, 6), {7, 8, 9),/* строка 1 */
{10,11,12), {13,14,15}, {16,17,18} /* строка 2 */
};
```

В этом примере по первой левой фигурной скобке в строке 1 начинается инициализация подмассива **nlist**[0], который является массивом из трех структур. Значения 1, 2, 3 назначаются трем элементам первой структуры (**nlist**[0][0]). Когда встретится правая фигурная скобка (после значения 3), инициализация подмассива **nlist**[0] закончится и две оставшиеся структуры — **nlist**[0][1] и **nlist**[0][2] — будут по умолчанию инициализированы нулевыми значениями. Аналогично, список {4,5,6} инициализирует первую структуру во второй строке **nlist** (т. е. **nlist**[1][0]), а оставшиеся две структуры — **nlist**[1][1] и **nlist**[1][2] — по умолчанию инициализируются нулевыми значениями. Когда компилятор языка Си обнаружит следующий список инициализаторов {7,8,9), он попытается инициализировать подмассив **nlist**[2]. Однако, поскольку **nlist** содержит только две строки и элемента **nlist**[2] в нем не существует, будет выдано сообщение об ошибке.

Пример 3.

```
union {
char m[2][3];
int i, j, k;
} y = {
{'1'},
{'4'}
};
```

В третьем примере инициализируется переменная y типа объединение. Первым элементом объединения является массив; он и будет инициализироваться. Список инициализаторов {'1'} задает значения для первой строки массива (m[0]). Поскольку в списке всего одно значение, то только первый элемент строки массива — m[0][0] —инициализируется символом '1', а оставшиеся два элемента в строке инициализируются по умолчанию нулевыми значениями (символом '\0'). Аналогично, первый элемент второй строки массива m инициализируется значением '4', а остальные элементы инициализируются по умолчанию нулевыми значениями.

3.7.3 Строковые инициализаторы

Существует специальная форма инициализации массива типа **char** — с помощью символьной строки. Например, объявление

```
char code[] = "abc";
```

инициализирует массив *code* четырьмя символами—'a', 'b', 'c' и символом '\0', который завершает символьную строку.

Если в объявлении размер массива указан, а длина инициализирующей строки превышает указанный размер, то лишние символы отбрасываются. Следующее объявление инициализирует трехэлементный массив *code* типа **char**:

```
char code[3] = "abcd";
```

В примере только три первые символа инициализатора заносятся в массив *code*. Символ *d* и символ '\0' отбрасываются.

Если инициализирующая строка короче, чем специфицированный размер массива, то оставшиеся элементы массива инициализируются нулевым значением (символом '\0').

Символьной строкой можно инициализировать не только массив типа **char**, но и указатель на тип **char**. Например, в объявлении

```
char *ptr = "abcd";
```

указатель ptr будет инициализирован адресом массива типа **char**, содержащего символы 'a', 'b', 'c', 'd', '\0'.

3.8 Объявление типа

Существует два особых вида объявления, в которых объявляется не переменная или функция, а тип данных. Первый вид позволяет определить тег и элементы структуры, объединения или перечислимого типа. После такого объявления имя типа (тег) может быть использовано в объявлениях переменных и функций для ссылки на этот тип.

Второй вид объявления типа использует ключевое слово **typedef**. Это объявление позволяет присвоить осмысленные имена типам, уже существующим в языке или создаваемым пользователем.

Объявление типа имеет такую же блочную область действия, как и объявление переменной. Локальное переобъявление имени типа также возможно. Однако теги занимают отдельное пространство имен, а идентификаторы, объявленные посредством **typedef**, разделяют пространство имен с переменными и функциями.

3.8.1 Объявление тега

Объявление типа структуры, объединения или перечислимого типа имеет такую же синтаксическую форму, как и объявление переменной этих типов, однако в объявлении типа идентификатор переменной (а в общем случае описатель) опущен. Именем типа структуры, объединения или перечислимого типа является тег, который в данном случае обязателен.

Примеры.

```
/* пример 1 */
enum status {
```

```
   loss = -1,
   bye,
   tie = 0,
   win
};
/* пример 2 */
struct student {
   char name [20];
   int id, class;
}
```

В первом примере объявляется перечислимый тип с именем **status**. Имя типа может быть использовано в объявлениях переменных этого перечислимого типа. Идентификатору **loss** явно присваивается значение -1. Идентификаторы **bye** и **tie** ассоциируются со значением 0, а **win** - со значением 1.

Во втором примере объявляется структурный тип с именем **student**. Объявление типа **student** позволяет записывать впоследствии лаконичные объявления переменных этого типа, например объявление **struct student employee**, в котором объявляется структурная переменная **employee** типа **student**.

3.8.2 Объявление typedef

Синтаксис:

typedef **<***спецификация типа***>** **<***описатель***>** {,**<***описатель*>…];

Объявление **typedef** синтаксически аналогично объявлению переменной или функции, за исключением того, что вместо спецификации класса памяти записывается ключевое слово **typedef** и отсутствует инициализатор.

Объявление **typedef** интерпретируется таким же образом, как объявление переменной или функции, однако идентификатор, входящий в состав описателя, специфицирует не переменную или функцию, а тип. Идентификатор становится синонимом для объявленного типа и может употребляться в последующих объявлениях. Другими словами, создаются не новые типы, а имена для специфицированных программистом типов. С помощью **typedef** может быть объявлено имя для любого типа, как базового, так и составного — указателя, функции, массива.

Объявление **typedef** для типа указатель на структуру, объединение или значение перечислимого типа, использующее только тег этой структуры, объединения или перечислимого типа, может быть записано раньше, чем данный тег будет определен в программе, однако определение тега должно находиться в пределах области действия этого объявления **typedef** и до того, как объявленный тип будет использован.

Принято записывать идентификаторы типов, объявленные посредством **typedef**, прописными буквами, однако это не является требованием языка.

Примеры:

```
/* пример 1 */
typedef int WHOLE;
/* пример 2 */
typedef struct club {
char name [30];
int size, year;
} GROUP;
/* пример 3 */
typedef GROUP *PG;
/* пример 4 */
typedef void DRAWF (int, int);
```

В первом примере объявляется тип WHOLE как синоним для типа **int**. Во втором примере объявляется тип GROUP для структурного типа, содержащего три элемента. Поскольку специфицирован также тег **club**, то в последующих объявлениях переменных может быть использован либо тип GROUP, либо тег **club**. Например, объявления GROUP **stgr**; и **struct club stgr**, эквивалентны по смыслу.

В третьем примере используется имя типа GROUP для объявления типа указатель. Тип PG объявляется как указатель на тип GROUP, который определен ранее как структурный тип. Например, объявление *PG ptr*, эквивалентно объявлению *struct club *pfr.*

В последнем примере объявлен тип DRAWF для функции, не возвращающей значения и требующей два аргумента типа **int**. Например, объявление DRAWF box; эквивалентно объявлению void box(int, int);.

3.8.3 Абстрактные имена типов

В разделах 3.8.1 и 3.8.2 рассматривались объявления, в которых типам присваиваются идентификаторы для последующего использования. Однако иногда возникает необходимость специфицировать некоторый тип данных без присвоения ему идентификатора и без объявления какого-либо объекта. Такая конструкция, определяющая тип без имени, называется абстрактным именем типа. Абстрактные имена типов используются в трех контекстах: в списках типов аргументов при объявлении функций, в операции приведения типа и в операции **sizeof**. Списки типов аргументов рассматривались в разделе 3.5 "Объявление функции". Операция приведения типа и операция **sizeof** обсуждаются в разделах 4.7.2 и 4.3.2, соответственно.

Абстрактными именами для базовых, перечислимых, структурных типов и объединений являются просто соответствующие им спецификации типа. Если в абстрактном имени типа

задано определение тега (см. раздел 3.8.1), то область действия этого тега распространяется в СП MSC на остаток блока, а в СП ТС — на остаток тела функции. Абстрактные имена для типов указатель, массив и функция задаются следующей синтаксической конструкцией:

<спецификация типа> <абстрактный описатель>

Абстрактный описатель отличается от обычного Описателя только тем, что он не содержит идентификатора. Как и обычный описатель, он может содержать один или более признаков указателя, массива и функции. Для интерпретации абстрактного описателя следует прежде всего определить в нем место подразумеваемого идентификатора. После этого интерпретация проводится так же, как описано в разделе 3.3.2. Абстрактный описатель, состоящий только из пары пустых круглых скобок, недопустим, поскольку он не позволяет однозначно определить, где подразумевается идентификатор: если внутри скобок, то описан простой тип (заданный только спецификацией типа), а если перед скобками, то тип функция.

Объявления **typedef**, рассмотренные в разделе 3.8.2, позволяют присваивать короткие, осмысленные идентификаторы абстрактным именам типов и могут использоваться в том же контексте, что и они.

Примеры:

```
long *                  /* пример 1 */
int (*)[5]              /* пример 2 */
int (*)(void)           /* пример 3 */
PG                      /* пример 4 */
```

В первом примере задано абстрактное имя типа для указателя на тип **long**.

Во втором примере задано абстрактное имя типа для указателя на массив из пяти элементов типа **int**.

В третьем примере задано абстрактное имя типа для указателя на функцию, не требующую аргументов и возвращающую значение типа **int**.

В четвертом примере с помощью идентификатора PG, объявленного посредством **typedef** в разделе 3.8.2, задано абстрактное имя типа "указатель на структуру с тегом **club**".

4 ВЫРАЖЕНИЯ

4.1 Введение

Выражение — это комбинация операндов и операций, задающая порядок вычисления некоторого значения. Операции определяют действия, выполняемые над операндами. Операнд в простейшем случае является константой или переменной. В общем случае каждый операнд выражения также представляет собой выражение, имеющее некоторое значение.

В отличие от многих языков программирования высокого уровня, в языке Си присваивание само является выражением. Как любое выражение, присваивание имеет значение — это тот результат, который присваивается переменной, задаваемой левым операндом. Помимо простого присваивания, в языке Си существуют составные операции присваивания, которые выполняют дополнительные операции над своими операндами.

Результат вычисления выражения зависит от приоритета операций, а также от возможных побочных эффектов. Приоритет операций определяет группирование операндов в выражении и последовательность выполнения операций. Побочным эффектом называется изменение значения какого-либо операнда, вызванное вычислением другого операнда. Для некоторых операций возникновение побочного эффекта зависит от порядка вычисления операндов.

Значение, представляемое операндом в выражении, имеет тип. Этот тип может быть в ряде случаев преобразован (явно или неявно) к другому типу по некоторым правилам. Преобразования типов описаны в разделе 4.7.

4.2 Операнды

Операндом выражения может быть константа, идентификатор или символьная строка. Эти операнды могут посредством так называемых первичных операций комбинироваться в первичные выражения — вызов функции, индексное выражение, выражение выбора элемента. Эти первичные выражения, в свою очередь, являются операндами содержащего их выражения. Комбинация их с другими операциями приводит к образованию новых, более сложных выражений, также являющихся операндами содержащего их выражения, и т.д. Часть выражения, заключенная в круглые скобки, также рассматривается как операнд выражения. Если все операнды выражения являются константами, оно называется константным выражением.

Каждый операнд имеет тип. В разделе 4.3 "Операции" рассматриваются допустимые типы операндов для каждого вида операций. Следует помнить, что перечислимый тип является подмножеством целого типа, и его значения участвуют в выражениях как значения целого типа. Тип операнда может быть явно преобразован к другому типу

посредством операции приведения типа (см. раздел 4.7.2). Выражение приведения типа само рассматривается как операнд содержащего его выражения.

4.2.1 Идентификаторы

Идентификаторы именуют переменные и функции. С каждым идентификатором ассоциируется тип, который задается при его объявлении. Значение объекта, именуемого идентификатором, зависит от типа следующим образом:

1) Идентификаторы переменных целого и плавающего типа представляют значения соответствующего типа.

2) Идентификатор переменной перечислимого типа представляет значение одной константы из соответствующего этому типу списка перечисления. Тип этого значения—**int**.

3) Идентификатор структуры или объединения представляет совокупность значений, специфицированных этой структурой или объединением.

4) Идентификатор указателя представляет адрес некоторого объекта специфицированного типа. Если указателю не присвоено никакого значения, то использование его в выражении может привести к трудно выявляемой ошибке. В языке Си определено, что никакой программный объект не может иметь адрес NULL (ноль), поэтому указатель со значением NULL не указывает ни на какой объект.

5) Идентификатор массива представляет массив, но если в выражении требуется скалярная величина, то представляет адрес первого элемента этого массива. Тип идентификатора — указатель на тип элементов массива. Из этого следует, что использование в выражениях массивов и указателей имеет много общего. Во многих случаях способ организации доступа к объекту — либо по указателю, либо как к элементу массива — не имеет принципиальной разницы и определяется исключительно выбранным стилем программирования. Это родство между массивами и указателями является существенной особенностью языка Си, выделяющей его среди других языков программирования высокого уровня.

Существуют, однако, некоторые различия между массивами и указателями. Во-первых, указатель занимает одну ячейку памяти, предназначенную для хранения машинного адреса (в частности, адреса какого-либо массива). Массив же занимает столько ячеек памяти, сколько элементов определено в нем при его объявлении. Только в выражении массив представляется своим адресом, который эквивалентен указателю. Во-вторых, адрес массива является постоянной величиной, поэтому, в отличие от идентификатора указателя, идентификатор массива не может составлять левую часть операции присваивания.

6) Идентификатор функции представляет функцию, т.е. адрес точки входа в функцию. Тип идентификатора — функция, возвращающая значение специфицированного для нее типа. Однако если в выражении требуется скалярная величина, то типом идентификатора считается указатель на функцию. Адрес функции не изменяется во время выполнения программы и, следовательно, не является переменной величиной. Поэтому идентификатор функции не может составлять левую часть операции присваивания.

4.2.2 Константы

Операнду-константе соответствует значение и тип представляющей его константы. Типы констант подробно описаны в разделе 1.2. Символьная константа имеет тип **int**. Целая константа имеет один из следующих типов: **int**, **long**, **unsigned int** или **unsigned long**, в зависимости от размера целого на данном компьютере и от того, как специфицировано ее значение. Константы с плавающей точкой имеют тип **double** (в версии 2.0 СП ТС допустимы также константы типа **float**). Символьные строки имеют тип массив символов; они обсуждаются в разделе 4.2.3.

4.2.3 Символьные строки

Символьная строка состоит из последовательности символов, заключенных в двойные кавычки. Эта последовательность представляется в памяти как массив элементов типа **char**. Символьная строка представляет в выражении адрес этого массива, т. е. адрес первого элемента строки.

Поскольку символьная строка представляет адрес массива, она может быть использована в контексте, допускающем значение типа указатель, подчиняясь при этом тем же ограничениям. Однако, поскольку адрес символьной строки является постоянной величиной, символьная строка не может составлять левую часть операции присваивания.

4.2.4 Вызовы функций

Синтаксис:

<выражение> (*<список-выражений>*)

Значением <выражения> должен быть адрес функции. В простейшем случае это идентификатор функции. *<Список выражений>* содержит выражения, разделенные запятыми. Значение каждого из этих выражений соответствует фактическому аргументу функции. Список выражений может быть пустым, если функция не имеет аргументов, однако наличие скобок и в этом случае обязательно.

Выражение вызова функции имеет тип — тип возвращаемого функцией значения. Если объявлен тип возвращаемого значения **void**, то и выражение вызова функции имеет тип **void**. Если возврат из вызванной функции произошел не в результате выполнения оператора **return**, содержащего выражение, то значение функции не определено. В разделе 6.4 дана более полная информация о вызовах функций.

4.2.5 Индексные выражения

Синтаксис:

<выражение1>[*<выражение2>*]

Здесь квадратные скобки являются символами языка Си, а не элементами описания.

Значение индексного выражения находится по адресу, который вычисляется как сумма значений *<выражения1>* и *<выражения2>*. Выражение1 должно иметь тип указателя на некоторый тип, например быть идентификатором массива, а выражение2, заключенное в квадратные скобки, должно иметь целый тип. Однако требование синтаксиса состоит лишь в том, чтобы одно из выражений было указателем, а другое имело целый тип; порядок же следования выражений безразличен.

Индексное выражение обычно используется для доступа к элементам массива, однако индексацию можно применить к любому указателю.

Индексное выражение вычисляется путем сложения целого значения со значением указателя (или с адресом массива) и последующим применением к результату операции косвенной адресации. Операция косвенной адресации описана в разделе 4.3.2. Например, для одномерного массива следующие четыре выражения эквивалентны, если a — массив или указатель, a b — целое.

```
a[b]
*(a + b)
*(b + a)
b[a]
```

В соответствии с правилами преобразования типов для операции сложения (смотри раздел 4.3.4) целочисленное значение при сложении с указателем (адресом) должно умножаться на размер типа, адресуемого указателем. Предположим, например, что идентификатор **line** определен как массив типа **int**. При вычислении выражения **line[i]**, целое значение **i** умножается на размер типа **int**. Полученное значение представляет **i** ячеек типа **int**. Это значение складывается со значением указателя **line**, что дает адрес объекта, смещенного на **i** ячеек типа **int** относительно **line**, т.е. адрес **i**-го элемента **line**.

Заключительным шагом вычисления индексного выражения является применение к полученному адресу операции косвенной адресации. Результатом является значение **i**-го элемента массива **line**.

Следует помнить, что индексное выражение **line**[0] представляет значение первого элемента массива, так как индексация элементов массива начинается с нуля. Следовательно, выражение **line**[5] ссылается на шестой по порядку следования в памяти элемент массива.

Доступ к многомерному массиву

Индексное выражение может иметь более одного индекса. Синтаксис такого выражения следующий:

<выражение1>[*<выражение2>*][*<выражение3>*]…

Индексное выражение интерпретируется слева направо. Сначала вычисляется самое левое индексное выражение — *<выражение1>*[*<выражение2>*]. С адресом, полученным в результате сложения *<выражения1>* и *<выражения2>*, складывается (по правилам сложения указателя и целого) *<выражение3>* и т. д. *<Выражение3>* и последующие *<выражения>* имеют целый тип. Операция косвенной адресации осуществляется после вычисления последнего индексного выражения. Однако, если значение последнего указателя адресует значение типа массив, операция косвенной адресации не применяется (смотри третий и четвертый примеры ниже).

Выражения с несколькими индексами ссылаются на элементы многомерных массивов. Многомерный массив в языке Си понимается как массив, элементами которого являются массивы. Например, элементами трехмерного массива являются двумерные массивы.

Примеры:

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
i = prop[0][0][1];        /* пример 1 */
i = prop[2][1][3];        /* пример 2 */
ip = prop[2][1];          /* пример 3 */
ipp = prop[2];            /* пример 4 */
```

Массив с именем **prop** содержит 3 элемента, каждый из которых является двумерным массивом значений типа **int**. В примере 1 показано, каким образом получить доступ ко второму элементу (типа **int**) массива **prop**. Поскольку массив заполняется построчно, последний индекс меняется наиболее быстро. Выражение **prop**[0][0][2] ссылается на следующий (третий) элемент массива и т. д.

Во втором примере выражение вычисляется следующим образом:

1) Первый индекс 2 умножается на размер двумерного массива (4 на 6), затем на размер типа **int** и прибавляется к значению указателя **prop**. Результат будет указывать на третий двумерный массив (размером 4 на 6 элементов) в трехмерном массиве **prop**.

2) Второй индекс 1 умножается на размер 6-элементного массива типа **int** и прибавляется к адресу, представляемому выражением **prop**[2].

3) Каждый элемент 6-элементного массива имеет тип **int**, поэтому индекс 3 умножается на размер типа **int** и прибавляется к адресу, представляемому выражением **prop**[2][1]. Результирующий указатель адресует четвертый элемент массива из шести элементов.

4) На последнем шаге вычисления выражения **prop**[2][1][3] выполняется косвенная адресация по указателю. Результатом является элемент типа **int**, расположенный по вычисленному адресу.

В примерах 3 и 4 представлены случаи, когда косвенная адресация не применяется. В примере 3 выражение **prop**[2][1] представляет указатель на массив из шести элементов в трехмерном массиве **prop**. Поскольку значение указателя адресует массив, операция косвенной адресации не применяется. Аналогично, результатом вычисления выражения **prop**[2] в примере 4 является значение указателя, адресующего двумерный массив.

4.2.6 Выбор элемента

Синтаксис:

<**выражение**>.<**идентификатор**>

<**выражение**> -> <**идентификатор**>

Выражение выбора элемента позволяет получить доступ к элементу структуры или объединения. Выражение имеет значение и тип выбранного элемента.

В первой синтаксической форме <*выражение*> представляет значение типа **struct** или **union**, а идентификатор именует элемент специфицированной структуры или объединения. Во второй синтаксической форме <*выражение*> представляет указатель на структуру или объединение, а идентификатор именует элемент специфицированной структуры.

Обе синтаксические формы выражения выбора элемента дают одинаковый результат. Запись

<*выражение*> -> <*идентификатор*>

для случая, когда <*выражение*> имеет тип указатель, эквивалентна записи

(*<*выражение*>).<*идентификатор*>

однако более наглядна.

Примеры:

```
struct pair {
int a;
inl b;
struct pair *sp;
} item, list[10];
item.sp = &item;            /* пример 1 */
(item.sp)->a = 24;          /* пример 2 */
list[8].b = 12;             /* пример 3 */
```

В первом примере адрес структуры Нет присваивается элементу **sp** этой же структуры. В результате структура **item** содержит указатель на себя.

Во втором примере используется адресное выражение **item.sp** с операцией выбора элемента ->, присваивающее значение элементу **a**. Учитывая результат примера 1, пример 2 эквивалентен записи

```
item.a = 24;
```

В третьем примере показано, каким образом в массиве структур осуществить доступ к элементу отдельной структуры.

4.2.7 Операции и L-выражения

В зависимости от используемых операций выражения подразделяются на первичные, унарные, бинарные, тернарные, выражения присваивания и выражения приведения типа.

Первичные выражения рассмотрены в разделах 4.2.4, 4.2.5, 4.2.6.

Унарное выражение состоит из операнда с предшествующей ему унарной операцией.

Синтаксис:

<унарная-операция> <операнд>

Унарные операции рассмотрены в разделе 4.3.2.

Бинарное выражение состоит из двух операндов, разделенных бинарной операцией.

Синтаксис:

<операнд1> <бинарная-операция> <операнд2>

Бинарные операции рассмотрены в разделах 4.3.3 — 4.3.9.

Тернарное выражение состоит из трех операндов, разделенных знаками условной операции "?:".

Синтаксис:

<операнд1> ? ***<операнд2>*** : ***<операнд3>***

Условная операция рассмотрена в разделе 4.3.10.

Выражения присваивания используют унарные или бинарные операции присваивания. Унарными операциями присваивания являются инкремент "++" и декремент "--". Бинарные операции присваивания — это простое присваивание "=" и составные операции присваивания. Каждая составная операция присваивания представляет собой комбинацию какой-либо бинарной операции с простой операцией присваивания.

Синтаксис выражений присваивания:

Унарные операции присваивания:

<операнд> ++

<операнд> --

++ *<операнд>*

--*<операнд>*

Бинарные операции присваивания:

<операнд1> = ***<операнд2>***

<операнд1> *<составное-присваивание>* *<операнд2>*

Операция присваивания рассмотрена в разделе 4.4.

Выражения приведения типа используют операцию приведения типа для явного преобразования типа переменной скалярного типа (целого, перечислимого, плавающего, пустого, указателя).

Синтаксис:

(*<абстрактное-имя-типа>*) *<операнд>*

Операция приведения типа подробно рассматривается в разделе 4.7.2. Абстрактные имена типов описаны в разделе 3.8.3.

Операнды некоторых операций в языке Си должны представлять собой так называемые L-выражения (Lvalue expressions). L-выражением является выражение, которое ссылается на ячейку памяти и потому имеет смысл в левой части бинарной операции присваивания. Простейшим примером L-выражения является идентификатор переменной: он ссылается на ячейку памяти, которая хранит значение этой переменной.

Поскольку L-выражение ссылается на ячейку памяти, адрес этой ячейки может быть получен с помощью операции адресации (&). Имеются, однако, исключения: не может быть получен адрес битового поля и адрес переменной класса памяти **register**, хотя значение им может быть присвоено.

К L-выражениям относятся:

— идентификаторы переменных целого, плавающего, перечислимого типов, указателей, структур и объединений;

— индексные выражения, исключая те из них, значение которых имеет тип массив;

— выражение выбора элемента, если выбранный элемент сам является одним из допустимых L-выражений;

— выражение косвенной адресации, если только его значение не имеет тип массив или функция;

— L-выражение в скобках;

— выражение приведения типа переменной, если размер результирующего типа не превышает размера первоначального типа. Следующий пример иллюстрирует этот случай:

```
char *p;
int i;
long n;
(long *)p = &n; /* допустимое приведение типа */
(long)i = n; /* недопустимое приведение типа */
```

Перечисленные L-выражения называются также модифицируемыми L-выражениями. Кроме того, существуют немодифицируемые L-выражения; их адрес может быть получен, но использоваться в левой части бинарной операции присваивания они не могут. К ним относятся идентификаторы массивов, функций, а также переменных, объявленных с модификатором **const**.

4.2.8 Скобочные выражения

Любой операнд может быть заключен в круглые скобки. В выражении

(10+5)/5

скобки означают, что выражение 10+5 является левым операндом операции деления. Результат выражения равен 3. В отсутствие скобок значение выражения равнялось бы 11.

Хотя скобки влияют на то, каким путем группируются операнды в выражении, они не гарантируют определенный порядок вычисления операндов для операций, обладающих свойством коммутативности (мультипликативные, аддитивные, поразрядные операции). Например, выражение (a+b)+c компилятор может вычислить как a+(b+c) или даже как (a+c)+b.

В СП ТС можно гарантировать порядок вычисления выражений в скобках для коммутативных операций с помощью операции унарного плюса.

4.2.9 Константные выражения

Константное выражение — это выражение, результатом вычисления которого является константа. Операндами константного выражения могут быть целые, символьные, плавающие константы, константы перечислимого типа, выражения приведения типа константного выражения, выражения с операцией **sizeof** и другие константные выражения. Имеются некоторые ограничения на использование операций в константных выражениях. В константных выражениях нельзя использовать операции присваивания, операцию последовательного вычисления. Кроме того, использование операции адресации, выражений приведения типа и плавающих констант ограничено.

Константные выражения, используемые в директивах препроцессора, имеют дополнительные ограничения, поэтому они называются ограниченными константными выражениями. Ограниченные константные выражения не могут содержать операцию **sizeof** (в СП ТС — могут), констант перечисления и выражений приведения типа и плавающих констант. Однако ограниченные константные выражения, используемые в директивах препроцессора, могут содержать специальные константные выражения **defined** (<*идентификатор*>), описанные в разделе 7.2.1 "Директива #**define**". . Только выражения инициализации допускают применение плавающих констант, выражений приведения типа к неарифметическим типам и операции адресации. Операция адресации может быть применена к переменной внешнего уровня базового или структурного типа, к объединению, а также к элементу массива. В этих выражениях допускается сложение или вычитание адресного выражения с константным выражением, не содержащим операции адресации.

4.3 Операции

Операции в языке Си имеют либо один операнд (унарные операции), либо два операнда (бинарные операции), либо три (тернарная операция). Операция присваивания может быть как унарной, так и бинарной (см. раздел 4.4).

Существенным свойством любой операции является ее ассоциативность. Ассоциативность определяет порядок выполнения в том случае, когда подряд применено несколько операций одного вида. Ассоциативность "слева направо" означает, что первой будет выполняться операция, знак которой записан левее остальных. Например, выражение

```
b << 2 << 2
```

выполняется как (b << 2) << 2, а не как b << (2 << 2). Ассоциативность "справа налево" означает, что первой будет выполняться операция, знак которой записан правее остальных.

В языке Си реализованы следующие унарные операции:

Знак операции	Наименование
-	унарный минус
+	унарный плюс
~	обратный код
!	логическое отрицание
&	адресация
*	косвенная адресация
sizeof	определение размера

Примечание. Операция унарного плюса реализована полностью только в СП ТС. В СП MSC версии 4 она отсутствует, а в версии 5 реализована только синтаксически.

Унарные операции предшествуют своему операнду и ассоциируются справа налево.

В языке Си реализованы следующие бинарные операции:

Знак	Наименование
* / %	мультипликативные операции
+ -	аддитивные операции
<< >>	операции сдвига
< > <= >= == !=	операции отношения
& | ^	поразрядные операции
&& ||	логические операции

,	операция последовательного вычисления

Бинарные операции ассоциируются слева направо. В языке Си имеется одна тернарная операция — условная, обозначаемая ?:. Она ассоциируется справа налево.

4.3.1 Преобразования по умолчанию

Большинство операций языка Си выполняют преобразование типов для приведения своих операндов к общему типу либо для того, чтобы расширить значения коротких по размеру типов до размера, используемого в машинных операциях. Преобразования, зависящие от конкретной операции и от типа операнда (или операндов), рассмотрены в разделе 4.7. Тем не менее, многие операции выполняют одинаковые преобразования целых и плавающих типов. Эти преобразования называются далее преобразованиями по умолчанию.

Преобразования по умолчанию осуществляются следующим образом:

1) Все операнды типа **float** преобразуются к типу **double**.
2) Только для СП TC: если один операнд имеет тип **long double**, то второй операнд также преобразуется к типу **long double**.
3) Если один операнд имеет тип **double**, то второй операнд преобразуется к типу **double**.
4) Если один операнд имеет тип **unsigned long**, то второй операнд преобразуется к типу **unsigned long**.
5) Если один операнд имеет тип **long**, то второй операнд преобразуется к типу **long**.
6) Если один операнд имеет тип **unsigned int**, то второй операнд преобразуется к типу **unsigned int**.
7) Все операнды типов **char** или **short** преобразуются к типу **int**.
8) Все операнды типов **unsigned char** или **unsigned short** преобразуются к типу **unsigned int**.
9) Иначе оба операнда имеют тип **int**.

4.3.2 Унарные операции

Унарный минус (-)

Операция унарного минуса выполняет арифметическое отрицание своего операнда. Операнд должен быть целым или плавающим значением. Выполняются преобразования операнда по умолчанию. Тип результата совпадает с преобразованным типом операнда.

Унарный плюс (+)

Эта операция реализована полностью в СП TC. В СП MSC версии 5 она реализована только синтаксически. Операция применяется для того, чтобы запретить компилятору языка Си реорганизовывать скобочные выражения.

Операнд унарного плюса должен иметь целый или плавающий тип. Над операндом выполняются преобразования по умолчанию. Операция унарного плюса не изменяет значения своего операнда.

Обычно компиляторы языка Си осуществляют перегруппировку выражений, переупорядочивая операции, обладающие свойством коммутативности (умножение, сложение, поразрядные операции), пытаясь сгенерировать как можно более эффективный код. При этом скобки, ограничивающие операции, не принимаются в расчет. Однако СП TC не будет реорганизовывать выражения в скобках, если перед скобками записана операция унарного плюса. Это позволяет, в частности, контролировать точность вычислений с плавающей точкой. Например, если a, b, c и f имеют тип **float**, выражение

```
f = a *+ (b * c)
```

будет гарантированно вычисляться следующим образом: результат сложения b и c будет прибавлен к a.

В СП MSC для гарантии порядка вычислений следует пользоваться вспомогательной переменной, например

```
t = b * c;
f = a * t
```

Обратный код (~)

Операция обратного кода вырабатывает двоичное дополнение своего операнда, т. е. инвертирует его битовое представление. Операнд должен иметь целый тип. Над операндом производятся преобразования по умолчанию. Результат имеет тип преобразованного операнда.

Если операнд имеет знаковый бит, то бит знака также участвует в операции обратного кода (инвертируется).

Логическое отрицание (!)

Операция логического отрицания вырабатывает значение 0, если операнд есть ИСТИНА, и значение 1, если операнд есть ЛОЖЬ. Результат имеет тип **int**. Операнд должен иметь целый или плавающий тип либо быть указателем.

Примеры:

```
/* пример 1 */
short x = 987;
x = ~x;
/* пример 2 */
unsigned short y = 0xAAAA;
y = ~y;
/* пример 3 */
if(!(x<y))…
```

В первом примере новое значение x равно -987.

Во втором примере переменной y присваивается новое значение, которое является обратным кодом беззнакового значения 0xAAAA, т. е. 0x5555.

В третьем примере, если x больше или равен y, то результат условного выражения в операторе if равен 1 (ИСТИНА). Если x меньше y, то результат равен 0 (ЛОЖЬ).

Адресация "&"

Операция адресации вырабатывает адрес своего операнда. Операндом может быть L-выражение, в т. ч. немодифицируемое (см. раздел 4.2.7). Результат операции адресации является указателем на операнд. Тип результата — указатель на тип операнда.

Операция адресации не может применяться к битовым полям, а также к идентификаторам, объявленным с классом памяти **register**.

См. примеры после описания операции косвенной адресации.

Косвенная адресация "*"

Операция косвенной адресации осуществляет доступ к значению по указателю. Ее операнд должен иметь тип указатель. В качестве операнда может также выступать идентификатор массива; в этом случае он преобразуется к указателю на тип элементов массива, и к этому указателю применяется операция косвенной адресации.

Результатом операции является значение, на которое указывает операнд. Типом результата является тип, ассоциированный с этим указателем. Если указателю перед операцией не было присвоено никакого значения, то результат непредсказуем.

Примеры:

```
int *pa, x;
int a[20];
double d;
pa = &a[5];             /* пример 1 */
x = *pa;                /* пример 2 */
if ( x == *&x )         /* пример 3 */
printf("ВЕРНО\n");
d = *(double *)(&x);    /* пример 4 */
```

В первом примере операция адресации вырабатывает адрес шестого (по порядку следования) элемента массива **a**. Результат записывается в адресную переменную (указатель) **pa**.

Во втором примере используется операция косвенной адресации для доступа к значению типа **int**, адрес которого хранится в указателе **pa**. Результат присваивается целой переменной **x**.

В третьем примере будет печататься слово ВЕРНО. Пример демонстрирует симметричность операций адресации и косвенной адресации: ***&x** эквивалентно **x**.

Четвертый пример показывает полезное приложение этого свойства. Адрес **x** преобразуется операцией приведения типа к типу указатель на **double**. К полученному указателю применяется операция косвенной адресации. Результатом выражения является значение типа **double**.

Операция sizeof

Операция **sizeof** определяет размер памяти, который соответствует объекту или типу. Операция **sizeof** имеет следующий вид:

sizeof <*выражение*>

sizeof (<*абстрактное имя типа*>)

Операндом является либо <*выражение*>, либо абстрактное имя типа в скобках. Результатом операции **sizeof** является размер памяти в байтах, соответствующий заданному объекту или типу. Тип результата — **unsigned int**. Если размер объекта не может быть представлен значением типа **unsigned int** (например, в СП MSC допустимы массивы типа **huge** размером более 64 Кбайтов), то следует использовать приведение типа:

(long) sizeof <*выражение*>

В СП MSC версии 4 допустимым выражением является L-выражение, а в версии 5 и в СП TC — произвольное выражение. Следует учитывать, что само <*выражение*> не вычисляется, т. к. операция **sizeof** выполняется на этапе компиляции программы. Для нее существен только тип результата <*выражения*>, а не его значение. Недопустим тип **void**. Применение операции **sizeof** к идентификатору функции в СП TC считается ошибкой, а в СП MSC эквивалентно определению размера указателя на функцию.

Если операция **sizeof** применяется к идентификатору массива, то результатом является размер всего массива в байтах, а не размер одного элемента.

Если операция **sizeof** применяется к типу структуры или объединения либо к идентификатору, имеющему тип структура или объединение, то результатом является фактический размер в байтах структуры или объединения, который может включать и участки пространства, используемые для выравнивания элементов структуры или объединения на границы слов памяти. Таким образом, этот результат может превышать размер, вычисленный путем сложения размеров отдельных элементов структуры. Например, если объявлена следующая структура

```
struct {
char m[3][3];
} s;
```

то значение **sizeof(s.m)** будет равно 9, а значение **sizeof(s)** будет равно 10.

Используя операцию **sizeof** для ссылок на размеры типов данных (которые могут различаться для разных компьютеров), можно повысить переносимость программы. В следующем примере операция **sizeof** используется для спецификации размера типа **int** в качестве аргумента стандартной функции распределения памяти **calloc**. Значение, возвращаемое функцией (адрес выделенного блока памяти), присваивается переменной **buffer**.

```
buffer = calloc(100, sizeof(int));
```

4.3.3 Мультипликативные операции

К мультипликативным операциям относятся операции умножения *, деления / и получения остатка от деления %. Операндами операции % должны быть целые значения. Операции умножения * и деления / выполняются над целыми и плавающими операндами. Типы первого и второго операндов могут отличаться, при этом выполняются преобразования операндов по умолчанию. Типом результата является тип операндов после преобразования.

В процессе выполнения мультипликативных операций ситуация переполнения или потери значимости не контролируется. Если результат мультипликативной операции не может быть представлен типом операндов после преобразования, то информация теряется.

Умножение (*)

Операция умножения выполняет умножение одного из своих операндов на другой.

Деление (/)

Операция деления выполняет деление первого своего операнда на второй. Если оба операнда являются целыми значениями не делятся нацело, то результат округляется в сторону нуля. Деление на нуль дает ошибку во время выполнения.

Остаток от деления (%)

Результатом операции является остаток от деления первого операнда на второй. Знак результата совпадает со знаком делимого.

```
Примеры:
int i = 10, j = 3, n;
double x = 2.0, y,
y = x*i;       /* пример 1 */
n = i/j;       /* пример 2 */
n = i%j;       /* пример 3 */
```

В первом примере **x** умножается на i. Результат равен 20.0 и имеет тип **double**.
Во втором примере 10 делится на 3. Результат округляется до 3 и имеет тип **int**.
В третьем примере n присваивается остаток от деления 10 на 3, т.е. 1.

4.3.4 Аддитивные операции

К аддитивным операциям относятся сложение (+) и вычитание (-). Их операндами могут быть целые и плавающие значения. В некоторых случаях аддитивные операции могут также выполняться над адресными значениями. Над операндами выполняются преобразования по умолчанию. Типом результата является тип операндов после преобразования. В процессе выполнения аддитивных операций ситуация переполнения или потери значимости не контролируется. Если результат аддитивной операции не может быть представлен типом операндов после преобразования, то информация теряется.

Сложение (+)

Операция сложения складывает два своих операнда. Операнды могут иметь целый или плавающий тип. Типы первого и второго операндов могут различаться. Один из операндов может быть указателем; тогда другой должен быть целым значением. Когда целое значение (назовем его i) складывается с указателем, то i масштабируется путем умножения его на

размер типа, с которым ассоциирован данный указатель. После преобразования целое значение представляет i ячеек памяти, где каждая ячейка соответствует по размеру типу, с которым ассоциирован данный указатель. Когда преобразованное целое значение складывается с указателем, то результатом является указатель, адресующий область памяти, расположенную на i ячеек дальше от первоначального адреса. Новый указатель указывает на тот же самый тип данных, что и исходный указатель.

Вычитание (-)

Операция вычитания вычитает второй операнд из первого. Операнды могут иметь целый или плавающий тип. Типы первого и второго операндов могут различаться. Допускается вычитание целого из указателя и вычитание двух указателей.

Когда целое значение вычитается из указателя, предварительно производится то же масштабирование, что и при сложении целого значения с указателем. Результатом вычитания является указатель, адресующий область памяти, расположенную на i ячеек перед первоначальным адресом. Новый указатель указывает на тот же самый тип данных, что и исходный указатель.

Один указатель может быть вычтен из другого, если они указывают на один и тот же тип данных. Разность между двумя указателями преобразуется к знаковому целому значению, путем деления разности на длину типа, который адресуется указателями. Результат представляет число ячеек памяти данного типа между двумя адресами.

Тип, который имеет разность указателей, зависит от компьютера, поэтому он определен посредством **typedef** в стандартном включаемом файле **stddef.h**. Имя этого типа — **ptrdiff.t**. Если разность указателей не может быть представлена этим типом, следует явно приводить ее к типу **long**.

4.3.4.1Адресная арифметика

Аддитивные операции, выполняемые над указателем и целым, имеют осмысленный результат в том случае, если указатель адресует массив памяти, а целое значение представляет смещение в пределах этого массива. Преобразование целого значения к адресному смещению предполагает, что в пределах смещения вплотную расположены элементы одинакового размера. Это предположение справедливо именно для элементов массива, поскольку массив определяется как последовательность значений одинакового типа, расположенных в смежных ячейках памяти. Способ хранения других типов данных не гарантирует сплошного заполнения памяти, т.е. даже между ячейками памяти, содержащими элементы одного и того же типа данных, возможны участки неиспользованной памяти. Поэтому корректность сложения и вычитания адресов, ссылающихся на какие-либо другие объекты, не гарантируется.

На компьютерах с сегментной архитектурой памяти (в частности, с микропроцессором типа 8086/8088) аддитивные операции над адресным и целым значениями могут не всегда выполняться правильно. Это вызвано тем, что указатели, используемые в программе, могут иметь различные размеры в зависимости от используемой модели памяти. Например, при компиляции программы в некоторой стандартной модели памяти адресные модификаторы (**near**, **huge**, **far**) могут специфицировать для какого-либо указателя другой размер, чем определяемый по умолчанию выбранной моделью памяти. Более подробная информация о работе с указателями в различных моделях памяти приведена в разделе 8 "Модели памяти".

Примеры:

```
int i = 4, j;
float x[10];
float *px;
px = &x[4] + 1;         /* пример 1 */
j = &x[i] — &x[i-2];    /* пример 2*/
```

В первом примере целочисленный операнд **i** складывается с адресом пятого (по порядку следования) элемента массива **x**. Значение **i** умножается на длину типа **float** и складывается с адресом **x[4]**. Значение результирующего указателя представляет собой адрес девятого элемента массива.

Во втором примере адрес третьего элемента массива **x** (заданный как **&x[i-2]**) вычитается из адреса пятого элемента (заданного как **&x[i]**). Полученная разность делится на размер типа **float**. В результате получается целое значение **2**.

4.3.5 Операции сдвига

Операции сдвига сдвигают свой первый операнд влево (<<) или вправо (>>) на число разрядов машинного слова, специфицированное вторым операндом. Оба операнда должны быть целыми значениями. Выполняются преобразования по умолчанию, причем в СП MSC над обоими операндами совместно, а в СП TC независимо над каждым операндом. Например, если переменная **b** имеет тип **int**, а переменная и тип **unsigned long**, то перед

выполнением операции **b<<u** в СП MSC переменная **b** будет преобразована к типу **unsigned long**.

Тип результата в СП TC — это тип левого операнда после преобразования, а в СП MSC — единый тип преобразованных операндов. В некоторых ситуациях результат в СП TC и в СП MSC может оказаться различным.

При сдвиге влево правые освобождающиеся биты заполняются нулями. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от того, какой тип результата получен после преобразования первого операнда. Если тип **unsigned**, то свободные левые биты заполняются нулями. В противном случае они заполняются копией знакового бита.

Если второй операнд отрицателен, то результат операции сдвига не определен.

При выполнении операций сдвига ситуация потери значимости не контролируется. Если результат сдвига не может быть представлен типом первого операнда после преобразования, то информация теряется.

Пример:

```
unsigned int x, y, z;
x = 0x00AA;
y = 0x5500;
z= (x<<8) + (y>>8);
```

В примере **x** сдвигается влево на 8 позиций, а **y** сдвигается вправо на 8 позиций. Результаты сдвигов складываются, давая значение OxAA5a, которое присваивается **z**.

4.3.6 Операции отношения

Операции отношения сравнивают первый операнд со вторым и вырабатывают значение 1 (ИСТИНА) или 0 (ЛОЖЬ). Результат имеет тип **int**. Имеются следующие операции отношения:

Операция	Проверяемое отношение
<	Первый операнд меньше, чем второй операнд
>	Первый операнд больше, чем второй операнд
<=	Первый операнд меньше или равен второму операнду
>=	Первый операнд больше или равен второму операнду
==	Первый операнд равен второму операнду
!=	Первый операнд не равен второму операнду

Операнды могут иметь целый, плавающий тип, либо быть указателями. Типы первого и второго операндов могут различаться. Над операндами выполняются преобразования по умолчанию.

Операндами любой операции отношения могут быть два указателя на один и тот же тип. Для операции проверки на равенство или неравенство результат сравнения означает, указывают ли оба указателя на одну и ту же ячейку памяти или нет. Результат сравнения указателей для других операций (<, >, <=, >=) отражает относительное положение двух адресов памяти.

Сравнение между собой адресов двух несвязанных объектов, вообще говоря, не имеет смысла. Однако сравнение адресов различных элементов одного и того же массива может быть полезным, поскольку элементы массива хранятся в памяти последовательно. Адрес предшествующего элемента массива всегда меньше, чем адрес последующего элемента.

Сравнение между собой указателей типа **far** не всегда имеет смысл, поскольку один и тот же адрес может быть представлен различными комбинациями значений сегмента и смещения и, следовательно, различными указателями типа **far**. Указатели типа **huge** в СП TC хранятся в нормализованном формате, поэтому их сравнение всегда корректно.

Указатель можно проверять на равенство или неравенство константе NULL (ноль). Указатель, имеющий значение NULL, не указывает ни на какую область памяти. Он называется нулевым указателем.

Из-за специфики машинной арифметики не рекомендуется проверять плавающие значения на равенство, поскольку 1.0/3.0*3.0 не будет равно 1.0.

Примеры:

```
int x, y;
x < y          /* выражение 1 */
y > x          /* выражение 2 */
x <= У         /* выражение 3 */
x >= У         /* выражение 4 */
x == У         /* выражение 5 */
x != y         /* выражение 6 */
```

Если x и y равны, то выражения 3, 4, 5 имеют значение 1, а выражения 1, 2, 6 имеют значение 0.

4.3.7 Поразрядные операции

Поразрядные операции выполняют над разрядами своих операндов логические функции И (&), включающее ИЛИ (|) и исключающее ИЛИ (^). Операнды поразрядных операций должны иметь целый тип, но бит знака, если он есть, также участвует в операции. Над операндами выполняются преобразования по умолчанию. Тип результата определяется типом операндов посте преобразования.

Таблица значений для поразрядных операций:

x	0	0	1	1
y	0	1	0	1
x|y	0	1	1	1
x&y	0	0	0	1
x^y	0	1	1	0

Примеры:

```
short i = 0xAB00;
short j = 0xABCD;
short n;
n = i & j;        /* пример 1 */
n = i | j;        /* пример 2 */
n = i ^ j;        /* пример 3 */
```

В первом примере n присваивается шестнадцатеричное значение AB00.

Во втором примере результатом операции включающего ИЛИ будет шестнадцатеричное значение ABCD, а в третьем примере результатом операции исключающего ИЛИ будет шестнадцатеричное значение CD.

4.3.8 Логические операции

Логические операции выполняют над своими операндами логические функции И (&&) и ИЛИ (||). Операнды логических операций могут иметь целый, плавающий тип, либо быть указателями. Типы первого и второго операндов могут различаться. Сначала всегда вычисляется первый операнд; если его значения достаточно для определения результата операции, то второй операнд не вычисляется.

Логические операции не выполняют преобразования по умолчанию. Вместо этого они вычисляют операнды и сравнивают их с нулем. Результатом логической операции является либо 0 (ЛОЖЬ), либо 1 (ИСТИНА). Тип результата — **int**.

Логическое И (&&)

Логическая операция И вырабатывает значение 1, если оба операнда имеют ненулевое значение. Если один из операндов равен нулю, то результат также равен нулю. Если значение первого операнда равно нулю, то значение второго операнда не вычисляется.

Логическое ИЛИ (||)

Логическая операция ИЛИ выполняет над своими операндами операцию включающее ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0; если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд не равен нулю, то значение второго операнда не вычисляется.

Примеры:

```
int x, y;
if(x<y && y<z) printf("x меньше z\n");              /* пример 1 */
if(x==y !! x==z) printf("x равен y или z\n");       /* пример 2 */
```

В первом примере функция **printf** вызывается для печати сообщения в том случае, если x меньше y и y меньше 2. Если x больше y, то второй операнд (y<z) не вычисляется и печати не происходит.

Во втором примере сообщение печатается в том случае, если x равен y или z. Если x равен y, то значение второго операнда (x==z) не вычисляется.

4.3.9 Операция последовательного вычисления

Операция последовательного вычисления последовательно вычисляет два своих операнда, сначала первый, затем второй. Оба операнда являются выражениями. Синтаксис операции:

<выражение1>, *<выражение2>*

Знак операции - запятая, разделяющая операнды. Результат операции имеет значение и тип второго операнда. Ограничения на типы операндов (т. е. типы результатов выражений) не накладываются, преобразования типов не выполняются.

Операция последовательного вычисления обычно используется для вычисления нескольких выражений в ситуациях, где по синтаксису допускается только одно выражение.

Примеры:

```
/* пример 1 */
for(i=j=1; i+j<20; i+=i, j--)…
/* пример 2 */
func_one( x, y + 2, z);
func_two((x--, y + 2), z);
```

В первом примере каждый операнд третьего выражения оператора цикла **for** вычисляется независимо. Сначала вычисляется **i+=i**, затем **j--**.

Во втором примере символ "запятая" используется как разделитель в двух различных контекстах. В первом вызове функции **func_onc** передаются три аргумента, разделенных запятыми: **x**, **y+2**, **2**. Здесь символ "запятая" используется просто как разделитель

В вызове функции **func_two** внутренние скобки вынуждают компилятор интерпретировать первую запятую как операцию последовательного вычисления. Этот вызов передает функции **func_two** два аргумента. Первый аргумент — это результат последовательного вычисления **(x--,y+2)**, имеющий значение и тип выражения **y+2**. Вторым аргументом является **z**.

4.3.10 Условная операция

В языке Си имеется одна тернарная операция — уловная. Она имеет следующий синтаксис:

<*операнд1***>** ? <*операнд2*>**: <***операнд3***>**

Выражение <*операнд1*> вычисляется и сравнивается с нулем. Выражение может иметь целый, плавающий тип, либо быть указателем. Если <*операнд1*> имеет ненулевое значение, то вычисляется <*операнд2*> и результатом условной операции является его значение. Если же <*операнд1*> равен нулю, то вычисляется <*операнд3*> и результатом является его значение. В любом случае вычисляется только один из операндов, <*операнд2*> или <*операнд3*>, но не оба.

Тип результата зависит от типов второго и третьего операндов (они могут различаться) следующим образом:

1) Если второй и третий операнды имеют целый или плавающий тип, то выполняются преобразования по умолчанию. Типом результата является тип операндов после преобразования.

2) Второй и третий операнды могут быть структурами, объединениями или указателями одного и того же типа. Типом результата будет тот же самый тип структуры, объединения или указателя.

3) Если либо второй, либо третий операнд имеет тип **void** (например, является вызовом функции, тип значения которой **void**), то другой операнд также должен иметь тип **void**, и результат имеет тип **void**.

4) Если либо второй, либо третий операнд является указателем на какой-либо тип, а другой является указателем на **void**, то результат имеет тип указатель на **void**.

5) Если либо второй, либо третий операнд является указателем, то другой может быть константным выражением со значением 0. Типом результата является указатель.

Пример:

```
j = (i < 0 )?(-i) : (i);
```

В примере **j** присваивается абсолютное значение **i**. Если **i** меньше нуля, то **j** присваивается **-i**. Если **i** больше или равно нулю, то **j** присваивается **i**.

4.4 Операции присваивания

В языке Си имеются следующие операции присваивания:

Операция	Действие
++	Унарный инкремент
--	Унарный декремент
=	Простое присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
<<=	Сдвиг влево с присваиванием
>>=	Сдвиг вправо с присваиванием
&=	Поразрядное И с присваиванием
|=	Поразрядное включающее ИЛИ с присваиванием
^=	Поразрядное исключающее ИЛИ с присваиванием

При присваивании тип правого операнда преобразуется к типу левого операнда. Специфика этого преобразования зависит от обоих типов и подробно описана в разделе 4.7.1. Левый (или единственный) операнд операции присваивания должен быть модифицируемым L-выражением (см. раздел 4.2.7).

Важное отличие присваивания в языке Си от операторов присваивания в других языках программирования состоит в том, что в языке Си операция присваивания вырабатывает значение, которое может быть использовано далее в вычислении выражения.

4.4.1 Операции инкремента и декремента

Операции ++ и -- инкрементируют (увеличивают на единицу) и декрементируют (уменьшают на единицу) свой операнд. Операнд должен иметь целый, плавающий тип или быть указателем. В качестве операнда допустимо только модифицируемое L-выражение.

Операнды целого или плавающего типа увеличиваются или уменьшаются на целую единицу. Над операндом не производятся преобразования по умолчанию. Тип результата соответствует типу операнда. Операнд типа указатель инкрементируется или декрементируется на размер объекта, который он адресует, по правилам, описанным в разделе 4.3.4. Инкрементированный указатель адресует следующий элемент данного типа, а декрементированный указатель—предыдущий.

Операции инкремента и декремента могут записываться как перед своим операндом (префиксная форма записи), так и после него (постфиксная форма записи). Для операции в префиксной форме операнд сначала инкрементируется или декрементируется, а затем его новое значение участвует в дальнейшем вычислении выражения, содержащего данную операцию. Для операции в постфиксной форме операнд инкрементируется лишь после того, как его старое значение участвует в вычислении выражения. Таким образом, результатом операций инкремента и декремента является либо новое, либо старое значение операнда.

Примеры:

```
/* пример 1 */
if(pos++ > 0) *p++ = *q++;
/* пример 2 */
if(line[--i] != '\n') return;
```

В первом примере переменная *pos* проверяется на положительное значение, а затем инкрементируется. Если значение *pos* до инкремента было положительно, то выполняется следующий оператор. В нем значение, указуемое **q**, заносится по адресу, содержащемуся в **p**. После этого **p** и **q** инкрементируются.

Во втором примере переменная **i** декрементируется перед ее использованием в качестве индекса массива **line**.

4.4.2 Простое присваивание

Операция простого присваивания обозначается знаком =. Значение правого операнда присваивается левому операнду. Левый операнд должен быть модифицируемым L-выражением. При присваивании выполняются правила преобразования типов, описанные в разделе 4.7.1.

Операция вырабатывает результат, который может быть далее использован в выражении. Результатом операции является присвоенное значение. Тип результата — тип левого операнда.

Пример 1:

```
double x;
int y;
x = y; Значение y преобразуется к типу double и присваивается x.
```

Пример 2:

```
int a, b, c; b = 2; a = b + (c = 5);
```

Переменной **c** присваивается значение **5**, переменной **a** — значение **b + 5**, равное **7**.

4.4.3 Составное присваивание

Операция составного присваивания состоит из простой операции присваивания, скомбинированной с какой-либо другой бинарной операцией. При составном присваивании вначале выполняется действие, специфицированное бинарной операцией, а затем результат присваивается левому операнду. Выражение составного присваивания со сложением, например имеет вид:

<*выражение1***>** += **<***выражение2***>**

Оно может быть записано и таким образом:

<*выражение1***> = <***выражение1***>** + **<***выражение2***>**

Значение операции вырабатывается по тем же правилам, что и для операции простого присваивания. Однако выражение составного присваивания не эквивалентно обычной записи, поскольку в выражении составного присваивания <*выражение1*> вычисляется только один раз, в то время как в обычной записи оно вычисляется дважды: в операции сложения и в операции присваивания. Например, оператор

```
*str1.str2.ptr += 5;
```

легче для понимания и выполняется быстрее, чем оператор

```
*str1.str2.ptr = *str1.str2.ptr + 5;
```

Использование составных операций присваивания может повысить эффективность программ. Каждая операция составного присваивания выполняет преобразования, которые определяются входящей в ее состав бинарной операцией, и соответственно ограничивает типы своих операндов. Результатом операции составного присваивания является значение, присвоенное левому операнду. Тип результата — тип левого операнда.

Пример:

```
n &= 0xFFFE;
```

В этом примере операция поразрядное И выполняется над n и шестнадцатеричным значением FFFE, и результат присваивается n.

4.5 Приоритет и порядок выполнения

Приоритет и ассоциативность операций языка Си влияют на порядок группирования операндов и вычисления операций в выражении. Приоритет операций существен только при наличии нескольких операций, имеющих различный приоритет. Выражения с более приоритетными операциями вычисляются первыми.

В таблице 4.1 приведены операции в порядке убывания приоритета. Операции, расположенные в одной строке таблицы, или объединенные в одну группу, имеют одинаковый приоритет и одинаковую ассоциативность.

Таблица 4.1.

Приоритет и ассоциативность операций в языке Си

Знак операции	Наименование	Ассоциативность
() [] . ->	Первичные	Слева направо
+ - ~ ! * & ++ -- sizeof приведение типа	Унарные	Справа налево
* / %	Мультипликативные	Слева направо
+ -	Аддитивные	Слева направо
>> <<	Сдвиг	Слева направо
< > <= >=	Отношение	Слева направо
== !=	Отношение	Слева направо
&	Поразрядное И	Слева направо
^	Поразрядное исключающее ИЛИ	Слева направо
|	Поразрядное включающее ИЛИ	Слева направо
&&	Логическое И	Слева направо
||	Логическое ИЛИ	Слева направо
?:	Условная	Справа налево
= *= /= %= += -= <<= >>= &= |= ^=	Простое и составное присваивание	Справа налево
,	Последовательное вычисление	Слева направо

Из таблицы 4.1. следует, что операнды, представляющие вызов функции, индексное выражение, выражение выбора элемента и выражение в скобках, имеют наибольший приоритет и ассоциативность слева направо. Приведение типа имеет тот же приоритет и порядок выполнения, что и унарные операции.

Выражение может содержать несколько операций одного приоритета. Когда несколько операций одного и того же уровня приоритета появляются в выражении, то они применяются в соответствии с их ассоциативностью — либо справа налево, либо слева направо.

Следует отметить, что в языке Си принят неудачный порядок приоритета для некоторых операций, в частности для операции сдвига и поразрядных операций. Они имеют более низкий приоритет, чем арифметические операции (сложение и др.). Поэтому выражение

```
a = b & 0xFF + 5
```

вычисляется как

```
a = b & (0xFF + 5),
```

а выражение

```
a +c >> 1
```

вычисляется как

```
(a + c) >> 1
```

Мультипликативные, аддитивные и поразрядные операции обладают свойством коммутативности. Это значит, что результат вычисления выражения, включающего несколько коммутативных операций одного и того же приоритета, не зависит от порядка выполнения этих операций. Поэтому компилятор оставляет за собой право вычислять такие выражения в любом порядке, даже в случае, когда в выражении имеются скобки, специфицирующие порядок вычисления.

В СП ТС реализована операция унарного плюса, позволяющая гарантировать порядок вычисления выражений в скобках.

Операция последовательного вычисления, логические операции И и ИЛИ, условная операция и операция вызова функции гарантируют определенный порядок вычисления своих операндов. Операция последовательного вычисления обеспечивает вычисление своих

операндов по очереди, слева направо (запятая, разделяющая аргументы в вызове функции, не является операцией последовательного вычисления и не обеспечивает таких гарантий). Гарантируется лишь то, что к моменту вызова функции все аргументы уже вычислены.

Условная операция вычисляет сначала свой первый операнд, а затем, в зависимости от его значения, либо второй, либо третий.

Логические операции также обеспечивают вычисление своих операндов слева направо. Однако логические операции вычисляют минимальное число операндов, необходимое для определения результата выражения. Таким образом, второй операнд выражения может вообще не вычисляться.

Пример:

```
int x, y, z, f();
z = x > y || f(x, y);
```

Сначала вычисляется выражение x>y. Если оно истинно, то переменной z присваивается значение 1, а функция f не вызывается. Если же значение x не больше y, то вычисляется выражение f(x,y). Если функция f возвращает ненулевое значение, то переменной z присваивается 1, иначе 0. Отметим также, что при вызове функции f гарантируется, что значение ее первого аргумента больше второго.

Рассмотренный пример показывает основные возможности использования порядка выполнения логических операций. Это, во-первых, повышение эффективности за счет помещения наиболее вероятных условий в качестве первых операндов логических операций. Во-вторых, это возможность вставки в выражение проверок, при ложности которых последующие действия не будут производиться. Так, в следующем условном операторе **if** чтение очередного символа из файла будет выполняться только в том случае, если конец файла еще не достигнут:

```
if(!feof(pf)) && (c = getc(pf)) …
```

Здесь **feof** — функция проверки на конец файла, **getc** — функция чтения символа из файла (см. раздел 12).

В-третьих, можно гарантировать, что в выражении **f(x)&&g(y)** функция **f** будет вызвана раньше, чем функция **g**. Для выражения **f(x)+g(y)** этого утверждать нельзя.

В последующих примерах показано группирование операндов для различных выражений.

Выражение	Группирование операндов
a & b || c	(a & b) || c
a = b || c	a = (b || c)
q && r || s--	(q && r) || (s--)
p == 0 ? p += 1 : p += 2	(p == 0 ? p += 1 : p) += 2

В первом примере поразрядная операция И (&) имеет больший приоритет, чем -логическая операция ИЛИ (||), поэтому выражение **a&b** является первым операндом логической операции ИЛИ.

Во втором примере логическая операция ИЛИ (||) имеет больший приоритет, чем операция простого присваивания, поэтому выражение **b||c** образует правый операнд операции присваивания. (Обратите внимание на то, что значение, присваиваемое **a**, есть нуль или единица.)

В третьем примере показано синтаксически корректное выражение, которое может выработать неожиданный результат. Логическая операция И (&&) имеет более высокий приоритет, чем логическая операция ИЛИ (||), поэтому запись **q&&r** образует операнд. Поскольку логические операции сначала вычисляют свой левый операнд, то выражение **q&&r** вычисляется раньше, чем **s--**. Однако если **q&&r** дает ненулевое значение, то **s--** не будет вычисляться и **s** не декрементируется. Более надежно было бы поместить **s--** на место первого операнда выражения либо декрементировать **s** отдельной операцией.

В четвертом примере показано неверное выражение, которое приведет к ошибке при компиляции. Операция равенства (==) имеет наибольший приоритет, поэтому **p==0** группируется в операнд. Тернарная операция **?:** имеет следующий приоритет. Ее первым операндом является выражение **p==0**, вторым операндом — выражение **p+=1**. Однако последним операндом тернарной операции будет считаться **p**, а не **p+=2**. так как в данном случае идентификатор **p** по приоритету операций связан более тесно с тернарной операцией, чем с составной операцией сложения с присваиванием. В результате возникает синтаксическая ошибка, поскольку левый операнд составной операции присваивания не является L-выражением.

Чтобы предупредить ошибки подобного рода и сделать программу более наглядной, рекомендуется использовать скобки. Предыдущий пример может быть корректно оформлен следующим образом:

```
(p == 0) ? (p += 1) : (p += 2)
```

4.6 Побочные эффекты

Побочный эффект выражается в неявном изменении значения переменной в процессе вычисления выражения. Все операции присваивания могут вызывать побочный эффект. Вызов функции, в которой изменяется значение какой-либо внешней переменной, либо путем явного присваивания, либо через указатель, также имеет побочный эффект.

Порядок вычисления выражения зависит от реализации компилятора, за исключением случаев, в которых явно гарантируется определенный порядок вычислений (см. раздел 4.5). При вычислении выражения в языке Си существуют так называемые контрольные точки. По достижении контрольной точки все предшествующие вычисления, в том числе все побочные эффекты, гарантированно произведены. Контрольными точками являются операция последовательного вычисления, условная операция, логические операции И и ИЛИ, вызов функции. Другие контрольные точки:

—конец полного выражения (т.е. выражения, которое не является частью другого выражения);

—конец инициализирующего выражения для переменной класса памяти **auto**;

—конец выражений, управляющих выполнением операторов **if**, **switch**, **for**, **do**, **while** и выражения в операторе **return**. Приведем примеры побочных эффектов:

```
add(i + 1, i = j + 2);
```

Аргументы вызова функции **add** могут быть вычислены в любом порядке. Выражение **i+1** может быть вычислено перед выражением **i=j+2**, или после него, с различным результатом в каждом случае.

Унарные операции инкремента и декремента также содержат в себе присваивание и могут быть причиной побочных эффектов, как это показано в следующем примере:

```
int i, a [10];
i = 0;
a[i++] = i;
```

Неизвестно, какое значение будет присвоено элементу a[0] — нуль или единица, поскольку для операции присваивания порядок вычисления аргументов не оговаривается.

4.7 Преобразования типов

Преобразование типов производится либо неявно, например при преобразовании по умолчанию или в процессе присваивания, либо явно, путем выполнения операции приведения типа. Преобразование типов выполняется также, когда преобразуется значение, передаваемое как аргумент функции. Далее рассматриваются правила преобразования для каждого из этих случаев.

4.7.1 Преобразования типов при присваивании

В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Преобразования при присваивании допускаются даже в тех случаях, когда они влекут за собой потерю информации.

Тип **long** d**o**uble ведет себя в преобразованиях аналогично типу **double**.

Преобразования знаковых целых типов Знаковое целое значение преобразуется к короткому знаковому целому значению (**short signed int**) посредством усечения старших битов. Знаковое целое значение преобразуется к длинному знаковому целому значению (**long signed int**) путем расширения знака влево. Преобразование знаковых целых значений к плавающим значениям происходит путем преобразования к типу **long**, а затем преобразования к плавающему типу. При этом возможна некоторая потеря точности. При преобразовании знакового целого значения к беззнаковому целому значению (**unsigned int**) производится лишь преобразование к размеру беззнакового целого типа, и результат интерпретируется как беззнаковое целое значение.

Правила преобразования знаковых целых типов приведены в таблице 4.2. Предполагается, что тип **char** по умолчанию является знаковым. Если во время компиляции используется опция, которая изменяет умолчание для типа **char** со знакового на беззнаковый, то для него выполняется преобразование как для типа **unsigned char** (см. таблицу 4.3).

Таблица 4.2.

Преобразование знаковых целых типов

От типа	К типу	Метод
char	short	дополнение знаком
char	long	дополнение знаком
char	unsigned char	сохранение битового представления;
char	unsigned short	старший бит теряет функцию знакового бита дополнение знаком до short; преобразование short в unsigned short
char	unsigned long	дополнение знаком до long; преобразование long в unsigned long
char	float	дополнение знаком до long; преобразование long к float
char	double	дополнение знаком до long; преобразование long к double
short	char	сохранение младшего байта
short	long	дополнение знаком
short	unsigned char	сохранение младшего байта
short	unsigned short	сохранение битового представления; старший бит теряет функцию знакового бита
short	unsigned long	дополнение знаком до long; преобразование long в unsigned long

short	float	дополнение знаком до long; преобразование long к float
short	double	дополнение знаком до long; преобразование long к double
long	char	сохранение младшего байта
long	short	сохранение младшего слова
long	unsigned char	сохранение младшего байта
long	unsigned short	сохранение младшего слова
long	unsigned long	сохранение битового представления; старший бит теряет функцию знакового бита
long	float	представляется как float; возможна некоторая потеря точности
long	double	представляется как double; возможна некоторая потеря точности

Примечание. В СП MSC и СП TC тип **int** эквивалентен типу **short** и преобразование для типа **int** производится как для типа **short**. В некоторых реализациях языка Си тип **int** эквивалентен типу **long** и преобразование для типа **int** производится как для типа **long**.

Преобразование беззнаковых целых типов

Беззнаковое целое значение преобразуется к короткому беззнаковому целому значению или короткому знаковому целому значению путем усечения старших битов. Беззнаковое целое значение преобразуется к длинному беззнаковому целому значению или длинному знаковому целому значению путем дополнения нулями слева. Беззнаковое целое значение преобразуется к значению с плавающей точкой путем преобразования к типу **long**, а затем преобразования значения типа **long** к значению с плавающей точкой.

Если беззнаковое целое значение преобразуется к знаковому целому значению того же размера, то битовое представление не меняется. Однако, если старший (знаковый) бит был установлен в единицу, представляемое значение изменится.

Правила преобразования беззнаковых целых типов приведены в таблице 4.3.

Таблица 4.3.

Преобразование беззнаковых целых типов

От типа	К типу	Метод
unsigned char	char	сохранение битового представления; старший бит становится знаковым
unsigned char	short	дополнение нулевыми битами
unsigned char	long	дополнение нулевыми битами
unsigned char	unsigned short	дополнение нулевыми битами
unsigned char	unsigned long	дополнение нулевыми битами
unsigned char	float	дополнение нулевыми битами до long; преобразование long к float
unsigned char	double	дополнение нулевыми битами до long; преобразование long к double
unsigned short	char	сохранение младшего байта
unsigned short	short	сохранение битового представления; старший бит становится знаковым
unsigned short	long	дополнение нулевыми битами
unsigned short	unsigned char	сохранение младшего байта
unsigned short	unsigned long	дополнение нулевыми битами
unsigned short	float	дополнение нулевыми битами до long; преобразование long к float
unsigned short	double	дополнение нулевыми битами до long; преобразование long к double
unsigned long	char	сохранение младшего байта
unsigned long	short	сохранение младшего слова
unsigned long	long	сохранение битового представления; старший бит становится знаковым
unsigned long	unsigned char	сохранение младшего байта
unsigned long	unsigned short	сохранение младшего слова
unsigned long	float	преобразование к long; преобразование long к float
unsigned long	double	преобразование к long; преобразование long к double (в версии 5 СП MSC это преобразование производится напрямую, без промежуточного типа long)

Примечание. В СП MSC и СП TC тип **unsigned int** эквивалентен типу **unsigned short** и преобразование для типа **unsigned int** производится как для типа **unsigned short**. В некоторых реализациях языка Си тип **unsigned int** эквивалентен типу **unsigned long** и преобразование для типа **int** производится как для типа **unsigned long**.

Преобразование плавающих типов Значения типа **float** преобразуются к типу **double** без потери точности. Значения типа **double** при преобразовании к типу **float** представляются с некоторой потерей точности. Однако если порядок значения типа **double** слишком велик для представления экспонентой значения типа **float**, то происходит потеря значимости, о чем сообщается во время выполнения.

Значения с плавающей точкой преобразуются к целым типам в два приема: сначала производится преобразование к типу **long**, а затем преобразование этого значения типа **long** к требуемому типу. Дробная часть плавающего значения отбрасывается при преобразовании к **long**; если полученное значение слишком велико для типа **long**, то результат преобразования не определен. Правила преобразования плавающих типов приведены в таблице 4.4.

Таблица 4.4.

От типа	К типу	Метод
float	char	преобразование к long; преобразование long к char
float	short	преобразование к long; преобразование long к short
float	long	усечение дробной части; результат не определен, если он слишком велик для представления типом long
float	unsigned short	преобразование к long; преобразование long к unsigned short
float	unsigned long	преобразование к long; преобразование long к unsigned long
float	double	дополнение мантиссы нулевыми битами справа
double	char	преобразование к float; преобразование float к char
double	short	преобразование к float; преобразование float к short
double	long	усечение дробной части; результат не определен, если он слишком велик для представления типом long
double	unsigned short	преобразование к long; преобразование long к unsigned short
double	unsigned long	преобразование к long; преобразование long к unsigned long
double	float	усечение младших битов мантиссы; возможна потеря точности; если значение слишком велико для представления типом float, то результат преобразования не определен

Преобразование указателей

Указатель на значение одного типа может быть преобразован к указателю на значение другого типа. Результат может, однако, оказаться неопределенным из-за отличий в требованиях к выравниванию объектов разных типов и в размере памяти, занимаемом различными типами.

Указатель при объявлении всегда ассоциируется с некоторым типом. В частности, это может быть тип **void**. Указатель на **void** можно преобразовывать к указателю на любой тип, и обратно. Указателям на некоторый тип можно присваивать адреса объектов другого типа, однако компилятор выдаст предупреждающее сообщение, если только это не указатель на тип **void**.

Указатели на любые типы данных могут быть преобразованы к указателям на функции, и обратно. Однако в СП MSC для того, чтобы присвоить указатель на данные указателю на функцию (или наоборот), необходимо выполнить явное приведение его типа.

Специальные ключевые слова **near**, **far**, **huge** позволяют модифицировать формат и размер указателей в программе. Компилятор учитывает принятый в выбранной модели памяти размер указателей и может в некоторых случаях неявно производить соответствующие преобразования адресных значений. Так, передача указателя в качестве аргумента функции может вызвать неявное преобразование его размера к большему из следующих двух значений:

—принятому по умолчанию размеру указателя для действующей модели памяти (например, в средней модели указатель на данные имеет тип **near**);

—размеру типа аргумента.

Если задано предварительное объявление функции, в котором указан явно тип аргумента-указателя, в т.ч. с модификаторами **near**, **far**, **huge**, то будет преобразование именно к этому типу.

Указатель может быть преобразован к значению целого типа. Метод преобразования зависит от размера указателя и размера целого типа следующим образом:

—если указатель имеет тот же самый или меньший размер, чем целый тип, то указатель преобразуется по тем же правилам, что и беззнаковое целое;

—если размер указателя больше, чем размер целого типа, то указатель сначала преобразуется к указателю того же размера, что и целый тип, а затем преобразуется к целому типу.

Значение целого типа может быть преобразовано к указателю по следующим правилам. Если целый тип имеет тот же самый размер, что и указатель, то производится преобразование к указателю без изменения в представлении. Если же размер целого типа отличен от размера указателя, то целый тип сначала преобразуется к целому типу, размер которого совпадает с размером указателя, используя правила преобразования, приведенные в таблицах 4.2 и 4.3. Затем полученному значению присваивается тип указатель.

Преобразования других типов

Из определения перечислимого типа следует, что его значения имеют тип **int**. Поэтому преобразования к перечислимому типу и из него осуществляются так же, как для типа **int**.

Недопустимы преобразования объектов типа структура или объединение.

Тип **void** не имеет значения по определению. Поэтому он не может быть преобразован к другому типу, и никакое значение не может быть преобразовано к типу **void** путем присваивания. Тем не менее, значение может быть явно преобразовано операцией приведения типа к типу **void** (см. раздел 4.7.2).

4.7.2 Явные преобразования типов

Явное преобразование типа может быть выполнено посредством операции приведения типа. Она имеет следующую синтаксическую форму

(<*абстрактное-имя-типа*>) <*операнд*>

<*абстрактное-имя-типа*> — специфицирует некоторый тип; <*операнд*> — выражение, значение которого должно быть преобразовано к специфицированному типу (абстрактные имена типов рассмотрены в разделе 3.8.3).

Преобразование операнда осуществляется так, как если бы он присваивался переменной типа <*имя-типа*>. Правила преобразования для операции присваивания, приведенные в разделе 4.7.1, полностью действуют для операции приведения типа. Однако, преобразование к типу **char** или **short** выполняется как преобразование к **int**. а преобразование к типу **float** — как преобразование к **double**.

Имя типа **void** может быть использовано в операции приведения типа, но результирующее выражение не может быть присвоено никакому объекту, и ему также нельзя ничего присвоить. Значение типа **void** не может быть приведено ни к какому типу; например, результат функции, возвращающей **void**, не может быть присвоен.

Результат операции приведения типа L-выражения сам является L-выражением и может представлять левый (или единственный) операнд операции присваивания, если приведенный тип не превышает по размеру исходный тип.

Если объявлен указатель на функцию, то в приведении его типа можно задавать другие типы аргументов. Например:

```
int (*p)(long);              /* объявление указателя на функцию */
(*(int(*)(int))p)(0);        /*вызов функции по указателю */
```

В операции приведения типа можно также задавать объявление структурного типа (тега), например:

```
(struct {int a; int b;} *)p->a = 5;
```

Область действия этого тега распространяется в СП MSC на остаток блока, а в СП TC — на остаток тела функции.

4.7.3 Преобразования типов при вызовах функций

Метод преобразования аргументов функция при ее вызове зависит от того, имеется ли предварительное объявление данной функции, содержащее список типов ее аргументов.

Если предварительное объявление имеется, и оно содержит список типов аргументов, то компилятор осуществляет контроль типов. Процесс контроля типов подробно описан в разделе 6.4.1 "Фактические аргументы".

Если предварительное объявление отсутствует, или в нем опущен список типов аргументов, то над аргументами вызываемой функции выполняются только преобразования по умолчанию. Преобразования выполняются отдельно для каждого аргумента вызова. Смысл этих преобразований сводится к тому, что значения типа **float** преобразуются к типу **double**, значения типов **char** и **short** преобразуются к типу **int**, значения типов **unsigned char** и **unsigned short** преобразуются к типу **unsigned int**.

Если в вызываемой функции в объявлениях формальных параметров - указателей используются модификаторы **near**, **far**, **huge**, то могут также происходить неявные

преобразования формата указателей, передаваемых в функцию. Процесс преобразования можно контролировать путем задания соответствующих модификаторов в предварительном объявлении функции в списке типов аргументов - указателей. В этом случае компилятор выполнит преобразования в соответствии со списком типов аргументов.

5 ОПЕРАТОРЫ

5.1 Введение

Операторы языка Си управляют процессом выполнения программы. Набор операторов языка Си содержит все управляющие конструкции структурного программирования. Ниже представлен полный список операторов:

пустой оператор
составной оператор или блок
оператор-выражение
условный оператор **if**
оператор пошагового цикла **for**
оператор цикла с предусловием **while**
оператор цикла с постусловием **do**
оператор продолжения **continue**
оператор-переключатель **switch**
оператор разрыва **break**
оператор перехода **goto**
оператор возврата **return**

В составе некоторых операторов используются выражения, выполняющие роль условий. В зависимости от значения такого условного выражения выбирается та или иная последовательность действий. В языке Си отсутствуют булевские выражения как самостоятельный класс выражений; в качестве условных выражений применяются обычные выражения языка Си. Значение выражения считается истинным, если оно не равно нулю, и ложным, если равно нулю. Из этого следует, что условные выражения не обязательно должны содержать операции отношения

```
if(a < 0) …
```

а могут выглядеть, например, так:

```
if(a) … или if(a + b)
```

В теле некоторых операторов языка Си могут содержаться другие операторы. Оператор, находящийся в теле другого оператора, в свою очередь может содержать операторы.

Составной оператор ограничивается фигурными скобками. Все другие операторы заканчиваются точкой с запятой (;). Точка с запятой в языке Си является признаком конца оператора, а не разделителем операторов, как в ряде других языков программирования.

Перед любым оператором языка Си может быть записана метка, состоящая из имени и двоеточия. Операторные метки распознаются только оператором **goto** (см. раздел 5.12 "Оператор перехода **goto**").

Программа на языке Си выполняется последовательно, оператор за оператором, за исключением случаев, когда какой-либо оператор явно передает управление в другую часть программы, например при вызове функции или возврате из функции.

5.2 Пустой оператор

Синтаксис:

```
;
```

Действие:

Пустой оператор — это оператор, состоящий только из точки с запятой. Он может появиться в любом месте программы, где по правилам синтаксиса требуется оператор. Выполнение пустого оператора не меняет состояния программы.

Пример:

```
for(i = 0; i < 10; line[i++] = 0);
```

Для таких операторов, как **do**, **for**, **if**, **while**, требуется, чтобы в их теле был хотя бы один оператор. Пустой оператор удовлетворяет требованиям синтаксиса в случаях, когда никаких действий не требуется. В приведенном примере третье выражение в заголовке оператора цикла **for** инициализирует первые 10 элементов массива **line** нулем. Тело оператора **for** состоит из пустого оператора, поскольку нет необходимости в других операторах.

Пустой оператор, подобно любому другому оператору языка Си, может быть помечен меткой. Например, чтобы пометить закрывающую фигурную скобку составного оператора, которая не является оператором, нужно вставить перед ней помеченный пустой оператор.

5.3 Составной оператор

Синтаксис:

{
[<*объявление*>]
.
.
.
[**<***оператор***>**]
}

Действие:

Действие составного оператора заключается в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда какой-либо оператор явно передает управление в другое место программы.

В начале составного оператора могут содержаться объявления (см. разделы 3.6, 3.6.2). Они служат для определения переменных, локальных для данного блока, либо для распространения на данный блок области действия глобальных объектов.

Пример:

```
if(i > 0) {
   line[i] = x;
   x++;
}
```

Типично использование составного оператора в качестве тела другого оператора, например оператора **if**. В приведенном примере, если **i** больше нуля, будут последовательно выполнены операторы, содержащиеся в составном операторе.

Подобно другим операторам языка Си, любой оператор внутри составного оператора может быть помечен. Передача управления по метке внутрь составного оператора возможна, однако если составной оператор содержит объявления переменных с инициализацией, то при входе в блок по метке эта инициализация не будет выполнена и значения переменных будут непредсказуемы.

Можно поставить метку и на сам составной оператор, если только это не оператор, составляющий тело функции.

5.4 Оператор-выражение

Синтаксис:

<*выражение*>;

Действие:

<*Выражение*> вычисляется в соответствии с правилами, изложенными в разделе 4 "Выражения". Отличие оператора-выражения состоит в том, что значение содержащегося в нем выражения никак не используется. Кроме того, он может быть записан лишь там, где по синтаксису допустим оператор.

Примеры:

```
x = y+3;        /*пример 1*/
x++;            /*пример 2*/
f(x);           /*пример 3*/
```

В первом примере **x** присваивается значение **y+3**. Во втором примере **x** инкрементируется.

В третьем примере показано выражение вызова функции. Если функция возвращает значение, то обычно оператор-выражение содержит операцию присваивания, чтобы запомнить значение, возвращаемое вызванной функцией. В данном примере возвращаемое значение не используется.

5.5 Условный оператор if

Синтаксис:

if(<*выражение*>)
<*оператор1*>
[else
<*оператор2***>**]

Действие:

Тело условного оператора **if** выполняется в зависимости от значения <*выражения*>.

Сначала вычисляется <*выражение*>. Если значение выражения истинно (не равно нулю), то выполняется <*оператор1*>. Если же значение выражения ложно, то выполняется <*оператор2*>, непосредственно следующий за ключевым словом **else**.

Если значение <*выражения*> ложно, но конструкция **else** опущена, то управление передается на оператор, следующий в программе за оператором **if**.

Пример:

```
if(i > 0)
y = x/i;
else {
x = 1;
y = f(x);
}
```

В примере, если **i** больше нуля, выполняется оператор **y=x/i;**. Если **i** меньше или равно нулю, то значение **i** присваивается переменной **x**, а значение, возвращаемое функцией **f(x)**, присваивается переменной **y**.

Вложенность

Оператор **if** может быть вложен в <*оператор1*> или <*оператор2*> другого оператора **if**. При вложении операторов **if** рекомендуется для ясности группирования операторов использовать фигурные скобки, ограничивающие <*оператор1*> и <*оператор2*>.

Если же фигурные скобки отсутствуют, то компилятор ассоциирует каждое ключевое слово **else** с ближайшим оператором **if**, у которого отсутствует конструкция **else**.

На ключевое слово **if** можно поставить метку, а на ключевое слово **else** — нельзя (однако можно поставить метку на <*оператор2*>, следующий за **else**).

Примеры.

```
/* пример 1 — без скобок */
if(i > 0)
if(j > i)
x = j;
else x = i;
/* пример 2 — со скобками */
if(i > 0) {
if(j > i)
x = j;
}
else
x = i;
```

В первом примере ключевое слово **else** ассоциируется с внутренним условным оператором **if**. Если **i** меньше или равно нулю, то переменной **x** ничего не присваивается.

Во втором примере фигурные скобки ограничивают внутренний условный оператор **if** и тем самым делают конструкцию **else** частью внешнего условного оператора **if**. Если **i** меньше или равно нулю, то переменной **x** присваивается значение **i**.

5.6 Оператор пошагового цикла for

Синтаксис:

for([<*начальное-выражение*>]; [<*условное-выражение*>]; [<*выражение-приращения*>])

<*оператор***>**

Действие:

Тело оператора цикла **for** выполняется до тех пор, пока <*условное-выражение*> не станет ложным. Если оно изначально ложно, то тело цикла не будет выполнено ни разу. <*Начальное-выражение*> и <*выражение-приращения*> обычно используются для инициализации и модификации параметров цикла или других значений.

Первым шагом при выполнении оператора цикла **for** является вычисление начального выражения, если оно имеется. Затем вычисляется условное выражение и производится его оценка следующим образом:

1) Если условное выражение истинно (не равно нулю), то выполняется тело оператора. Затем вычисляется выражение приращения (если оно есть), и процесс повторяется.

2) Если условное выражение опущено, то его значение принимается за истину и процесс выполнения продолжается, как описано выше. В этом случае оператор цикла **for** представляет бесконечный цикл, который может завершиться только при выполнении в его теле операторов **break**, **goto**, **return**.

3) Если условное выражение ложно, то выполнение оператора **for** заканчивается и управление передается следующему за ним оператору в программе. Оператор **for** может завершиться и при выполнении операторов **break**, **goto**, **return** в теле оператора.

Пример:

```
for(i = space = tab = 0; i < MAX; i++) {
if(line[i] == '\x20')
space++;
```

```
if(line[i] =='\1'){
lab++;
line[i] = '\x20';
}
}
```

В приведенном примере подсчитываются символы пробела ('\x20') и горизонтальной табуляции ('\1') в массиве символов с именем **line** и производится замена каждого символа горизонтальной табуляции на пробел.

Сначала **i**, **space** и **tab** инициализируются нулевыми значениями. Затем **i** сравнивается с константой MAX. Если **i** меньше MAX, то выполняется тело оператора. В зависимости от значения **line[i]** выполняется тело одного из операторов **if** (или ни одного из них). Затем переменная **i** инкрементируется и снова сравнивается с именованной константой MAX. Тело оператора выполняется до тех пор, пока значение **i** не станет больше или равно MAX.

5.7 Оператор цикла с предусловием while

Синтаксис:

while (<*выражение*>) **<***оператор***>**

Действие:

Тело оператора цикла **while** выполняется до тех пор, пока значение <*выражения*> не станет ложным (т.е. равным нулю). Вначале вычисляется <*выражение*>. Если <*выражение*> изначально ложно, то тело оператора **while** вообще не выполняется и управление сразу передается на следующий за телом цикла оператор программы. Если <*выражение*> истинно, то выполняется тело цикла. Перед каждым следующим выполнением тела цикла <*выражение*> вычисляется заново. Этот процесс повторяется до тех пор, пока <*выражение*> не станет ложкам. Оператор цикла **while** может также завершиться при выполнении операторов **break**, **goto**, **return** внутри своего тела.

Пример:

```
while (i >= 0) {
slring1[i] = string2[i];
i--;
}
```

В вышеприведенном примере элементы массива **string2** копируются в массив **string1**. Если **i** больше или равно нулю, то производится копирование (путем присваивания) очередного элемента, после чего **i** декрементируется. Когда **i** становится меньше нуля, выполнение оператора **while** завершается.

5.8 Оператор цикла с постусловием do

Синтаксис:

do **<***оператор***>** while (**<***выражение***>**);

Действие:

Тело оператора цикла **do** выполняется один или несколько раз до тех пор, пока значение <*выражения*> не станет ложным (равным нулю). Вначале выполняется тело цикла — <*оператор*>, затем вычисляется условие — <*выражение*>. Если выражение ложно, то оператор цикла **do** завершается и управление передается следующему за оператором **while** оператору программы. Если значение выражения истинно (не равно нулю), то тело цикла выполняется снова, и снова вычисляется выражение. Выполнение тела оператора цикла **do** повторяется до тех пор, пока выражение не станет ложным. Оператор **do** может также завершиться при выполнении в своем теле операторов **break**, **goto**, **return**.

Пример:

```
do {
y = f(x);
x--;
} while(x > 0);
```

Вначале выполняется составной оператор. Затем вычисляется выражение **x>0**. Если оно истинно, то составной оператор выполняется снова, и снопа вычисляется выражение **x>0**. Тело оператора цикла **do** выполняется до тех пор, пока значение **x** не станет меньше или равно нулю.

5.9 Оператор продолжения continue

Синтаксис:

continue;

Действие:

Оператор продолжения **continue** передает управление на следующую итерацию в операторах цикла **do**, **for**, **while**. Он может появиться только в теле этих операторов. Остающиеся в теле цикла операторы при этом не выполняются. В операторах цикла **do** и **while** следующая итерация начинается с вычисления условного выражения. Для оператора **for** следующая итерация начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения.

Пример:

```
while(i-- > 0) {
x = f(i);
```

```
if(x == 1)
continue;
else
y = x * x;
}
```

Тело оператора цикла **while** выполняется, если **i** больше нуля. Сначала значение **f(i)** присваивается **x**; затем, если **x** не равен **1**, то **y** присваивается значение квадрата **x**, и управление передается в заголовок цикла, т. е. на вычисление выражения **i-->0**. Если же x равен **1**, выполняется оператор продолжения **continue**, и выполнение возобновляется с заголовка оператора цикла **while**, без вычисления квадрата **x**.

5.10 Оператор-переключатель switch

Синтаксис:

switch(**<***выражение***>**)
{
[<*объявление***>**]
[case <*константное-выражение*>:] [**<***оператор***>**]
[case <*константное-выражение*>:] [<*оператор*>]
[default:] [**<***оператор*>]
}

Действие:

Оператор-переключатель **switch** предназначен для выбора одного из нескольких альтернативных путей выполнения программы. Выполнение оператора-переключателя начинается с вычисления значения выражения переключения (выражения, следующего за ключевым словом **switch** в круглых скобках). После этого управление передается одному из <*операторов*> тела переключателя. В теле переключателя содержатся конструкции case <*константное-выражение*>:, которые синтаксически представляют собой метки операторов. Константные выражения в данном контексте называются константами варианта. Оператор, получающий управление, — это тот оператор, значение константы варианта которого совпадает со значением выражения переключения. Значение каждой константы варианта должно быть уникальным внутри тела оператора-переключателя.

Выполнение тела оператора-переключателя **switch** начинается с выбранного таким образом оператора и продолжается до конца тела или до тех пор, пока какой-либо оператор не передаст управление за пределы тела.

Оператор, следующий за ключевым словом **default**, выполняется, если ни одна из констант варианта не равна значению выражения переключения. Если же слово **default** опущено, то ни один оператор в теле переключателя не выполняется, и управление передается на оператор, следующий за переключателем в программе.

Выражение переключения должно иметь целочисленный тип. В версии 4 СП MSC этот тип не должен превышать по размеру **int**; в версии 5 СП MSC и в СП TC это может быть любой целочисленный тип (в том числе **enum**). Однако в версии 5 СП MSC выражение переключения всегда преобразуется к типу **int**. Если при этом преобразовании возможна потеря значащих битов, то компилятор выдаст предупреждающее сообщение. Тип каждой константы варианта также приводится к типу выражения переключения.

Синтаксически конструкции **case** и **default** являются метками (однако, на них нельзя передать управление по оператору **goto**). Метки **case** и **default** существенны только при начальной проверке, когда выбирается оператор для выполнения в теле переключателя. Все операторы тела, следующие за выбранным, выполняются последовательно, как бы "не замечая" меток **case** и **default**, если только какой-либо из операторов не передаст управление за пределы тела оператора **switch**. Для выхода из тела переключателя обычно используется оператор разрыва **break**. Одна из распространенных ошибок состоит в том, что программисты забывают разделять альтернативные операторы в теле переключателя операторами **break**.

В заголовок составного оператора, формирующего тело оператора **switch**, можно помещать объявления (см. раздел 5.3), но инициализаторы, включенные в объявления, не будут выполнены, поскольку при выполнении оператора **switch** управление непосредственно передается на выполняемый оператор внутри тела, обходя строки, которые содержат инициализацию.

Примеры:

```
/* пример 1 */
switch (c) {
case 'A': capa++;
case 'a': lettera++;
default: total++;
}
/* пример 2 */
switch (i) {
case -1: n++;
```

```
break;
case 0: z++;
break;
case 1: p++;
break;
}
/* пример 3 */
switch (i) {
case 1:     if(a > 0) {
case 2: b = 3;
} else
case 3: k = 0;
}
```

В первом примере все три оператора в теле оператора **switch** выполняются, если значение **c** равно 'A'. Передача управления осуществляется на первый оператор (**capa++;**), далее операторы выполняются в порядке их следования в теле.

Если **c** равно 'a', те инкрементируются переменные **lettera** и **total**. Наконец, если **c** не равно 'A' или 'a', то инкрементируется только переменная **total**.

Во втором примере в теле оператора **switch** после каждого оператора следует оператор разрыва **break**, который осуществляет принудительный выход из тела оператора-переключателя **switch**. Если **i** равно **-1**, переменная **n** инкрементируется. Оператор **break**, следующий за оператором **n++**, вызывает передачу управления за пределы тела переключателя, минуя остающиеся операторы. Аналогично, если **i** равно нулю, инкрементируется только переменная **z**; если **i** равно **1**, инкрементируется только переменная **p**. Передний оператор **break** не является обязательным, поскольку без него управление все равно перешло бы на конец составного оператора, но он включен для единообразия.

В третьем примере при **i**, равном **1**, будет выполнена следующая последовательность действий:

```
if(a > 0)
b = 3;
else
k = 0;
```

При i, равном 2, переменной b будет присвоено значение 3. При i, равном 3, переменная k будет обнулена.

Оператор в теле переключателя может быть помечен множественными метками **case**, как показано в следующем примере:

```
switch (c) {
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'i': hexcvt(c);
}
```

В этом примере, если выражение переключения примет любое из значений 'a', 'b', 'c', 'd', 'e', 'i', будет вызвана функция **hexcvt**.

5.11 Оператор разрыва break

Синтаксис:

```
break;
```

Действие:

Оператор разрыва **break** прерывает выполнение операторов **do**, **for**, **while** или **switch**. Он может содержаться только в теле этих операторов. Управление передается оператору программы, следующему за прерванным. Появление оператора **break** вне операторов **do**, **for**, **while**, **switch** компилятор рассматривает как ошибку.

Если оператор разрыва **break** записан внутри вложенных операторов **do**, **for**, **while**, **switch**, то он завершает только непосредственно охватывающий его оператор **do**, **for**, **while**, **switch**. Если же требуется завершение более чем одного уровня вложенности, следует использовать операторы возврата **return** и перехода **goto**.

Пример:

```
for(i = 0; i < LENGTH; i++) {
for(j = 0; j < WIDTH; j++)
if(lines[i][j] == '\0') break;
lengths[i] = j;
}
```

В вышеприведенном примере построчно обрабатывается массив строк переменной длины **lines**. Именованная константа LENGTH задает количество строк в массиве LINES. Именованная константа WIDTH задает максимально допустимую длину строки. Задача состоит в заполнении массива **lengths** длинами всех строк массива **lines**. Оператор разрыва **break** прерывает выполнение внутреннего цикла **for** при обнаружении признака конца символьной строки (\0'). После этого i-му элементу одномерного массива **length [i]** присваивается длина i-й строки в байтах. Управление передается внешнему оператору цикла **for**. Переменная **i** инкрементируется и процесс повторяется до тех пор, пока значение **i** не станет больше или равно значению константы LENGTH.

5.12 Оператор перехода goto

Синтаксис:

goto <*метка*>;

.

.

.

<*метка*>: <*оператор*>

Действие:

Оператор перехода **goto** передает управление непосредственно на <*оператор*>, помеченный <*меткой*>. Метка представляет собой обычный идентификатор, синтаксис которого описан в разделе 1.3. Область действия метки ограничивается функцией, в которой она определена; из этого следует, во-первых, что каждая метка должна быть отлична от других меток в той же самой функции; во-вторых, что нельзя передать управление по оператору **goto** в другую функцию.

Помеченный оператор выполняется сразу после выполнения оператора **goto**. Если оператор с данной меткой отсутствует или существует более одного оператора, помеченного той же меткой, то компилятор сообщает об ошибке. Метка оператора имеет смысл только для оператора **goto**. При последовательном выполнении операторов помеченный оператор выполняется так же, как если бы он не имел метки.

Можно войти в блок, тело цикла, условный оператор, оператор-переключатель по метке.

Нельзя с помощью оператора **goto** передать управление на конструкции **case** и **default** в теле переключателя.

Пример:

```
if(errorcode > 0) goto exit;
…
exit: return (errorcode);
```

В примере оператор перехода **goto** передает управление на оператор, помеченный меткой **exit**, если **errorcode** больше нуля.

5.13 Оператор возврата return

Синтаксис:

return [<*выражение*>];

Действие:

Оператор возврата **return** заканчивает выполнение функции, в которой он содержится, и возвращает управление в вызывающую функцию. Управление передается в точку вызывающей функции, непосредственно следующую за оператором вызова. Значение <*выражения*>, если оно задано, вычисляется, приводится к типу, объявленному для функции, содержащей оператор возврата **return**, и возвращается в вызывающую функцию. Если <*выражение*> опущено, то возвращаемое функцией значение не определено.

Пример:

```
main()
{
void draw(int, int);
long sq(int);
y = sq(x);
draw(x, y);
}
long sq(int x)
{
return (x*x);
}
void draw(int x, int y)
{
return,
}
```

Функция **main** вызывает две функции, **sq** и **draw**. Функция **sq** возвращает значение квадрата **x**. Это значение присваивается переменной **y**. Функция **draw** объявлена с типом **void**, как не возвращающая значения. Попытка присвоить значение. возвращаемое функцией **draw**, привело бы к сообщению компилятора об ошибке.

<*Выражение*> в операторе возврата **return** принято заключать в скобки, как показано в примере. Это, однако, не является требованием языка.

Если оператор **return** отсутствует в теле функции, то управление автоматически передается в вызывающую функцию после выполнения последнего оператора в вызванной функции, т. е. по достижении последней закрывающей фигурной скобки. Возвращаемое значение вызванной функции в этом случае не определено. Если возвращаемое значение не требуется, то функцию следует явно объявлять с типом **void**.

Распространенной ошибкой является наличие в функции, которая должна возвращать значение, операторов возврата, как с выражением, так и без него.

6 ФУНКЦИИ

6.1 Введение

Функция — это совокупность объявлений и операторов, предназначенная для выполнения некоторой отдельной задачи. Количество функций в программе не ограничивается. Любая программа на языке Си содержит, по крайней мере, одну функцию, так называемую главную функцию, с именем **main**. В данном разделе описывается, как определять, объявлять и вызывать функции в языке Си.

Определение функции специфицирует имя функции, атрибуты ее формальных параметров, и тело функции, содержащее Объявления и операторы. В определении функции также может задаваться класс памяти функции и тип возвращаемого значения.

Объявление функции задает имя и тип возвращаемого значения функции, явное определение которой приведено в другом месте программы. В объявлении функции могут быть также специфицированы класс памяти, число аргументов функции и их типы. Прототип функции (версия 5.0 СП MSC и СП ГГС) позволяет, помимо того, задавать в объявлении идентификаторы и класс памяти аргументов. Это позволяет компилятору сравнивать при вызове типы фактических аргументов и формальных параметров функции.

Указание типа возвращаемого значения в определении функции необязательно, если это тип **int**. При другом типе возвращаемого значения необходимо указать этот тип в объявлении функции. К моменту вызова функции тип ее возвращаемого значения должен быть известен, поэтому перед вызовом может потребоваться предварительное объявление функции с указанием типа ее возвращаемого значения.

Вызов функции передает управление от вызывающей функции к вызываемой. Значения фактических аргументов, если они есть, передаются в вызываемую функцию. При выполнении оператора возврата **return** в вызываемой функции управление и возвращаемое значение (если оно есть) передаются в вызывающую функцию.

6.2 Определение функции

Определение функции специфицирует имя, формальные параметры и тело функции. Оно может также специфицировать тип возвращаемого значения и класс памяти функции. Синтаксис определения функции следующий:

[<*спецификация КП*>][<*спецификация типа*>]
<*описатель*> ([<*список объявлений параметров*>]) <*тело функции*>
<*тело функции*>

Спецификация класса памяти <*спецификация КП*> задает класс памяти функции. <*Спецификация типа*> в совокупности с описателем определяет тип возвращаемого значения и имя функции. <*Список объявлений параметров*> аналогичен списку типов аргументов в прототипе функции (см. раздел 3.5 "Объявление функции"). Он содержит объявления формальных параметров через запятую. Однако если в прототипе область действия идентификаторов ограничена этим же прототипом, то в списке объявлений параметров идентификаторы именуют формальные параметры данной функции. Их область действия — тело функции. <*Тело функции*> — это составной оператор, содержащий объявления локальных переменных и операторы.

В следующих разделах детально описываются перечисленные элементы определения функции.

6.2.1 Класс памяти

В определении функции допускается указание спецификации класса памяти **static** или **extern**. Классы памяти функций рассматривались в разделе 3.6.

6.2.2 Модификаторы типа функции

Компилятор языка Си поддерживает ряд модификаторов типа функций: **pascal**, **cdecl**, **interrupt**, **near**, **far** и **huge** (модификатор **interrupt** не реализован в версии 4 СП MSC). Модификаторы рассмотрены в разделе 3.3.3 "Описатели с модификаторами".

6.2.3 Типы возвращаемых значений

Синтаксис задания типа возвращаемого значения функции описан в разделе 3.5 "Объявление функции", функция может возвращать значение любого типа, кроме массива или функции; она может, в частности, возвращать указатель на любой тип, включая массив и функцию.

Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу возвращаемого значения во всех объявлениях этой функции, если они имеются в программе. Для вызова функции с типом возвращаемого значения **int** не требуется ее предварительно объявлять или определять. Функции с другими типами возвращаемого значения должны быть определены или объявлены до того, как они будут вызваны.

Возвращаемое значение функции вырабатывается при выполнении оператора возврата **return**, содержащего выражение. Выражение вычисляется, преобразуется к типу возвращаемого значения и возвращается в точку вызова функции. Если оператор **return** отсутствует или не содержит выражения, то возвращаемое значение функции не определено. Если в этом случае вызывающая функция ожидает возвращаемое значение, то поведение программы непредсказуемо.

Примеры:

```
/* пример 1 */
/* тип возвращаемого значения int */
static add(int x, int y)
{
return (x + y);
}
/* пример 2 */
/* тип возвращаемого значения STUDENT */
typedef struct {
char name [20],
int id;
long class;
} STUDENT;
STUDENT sortstu(STUDENT a, STUDENT b)
{
return (a.id < b.id ? a : b);
}
/* пример 3 */
/* тип возвращаемого значения — указатель на char */
char *smallstr(char *s1, char *s2)
{
int i;
i = 0;
while(s1[i] != '\0' && s2[i] != '\0')
i++;
if(s1[i] == '\0')
return (s1);
else
return (s2);
}
```

В первом примере по умолчанию тип возвращаемого значения функции **add** определен как **int**. Функция имеет класс памяти **static**. Это значит, что она может быть вызвана только функциями того же исходного файла, в котором она определена.

Во втором примере посредством объявления **typedef** создан структурный тип STUDENT. Далее определена функция **sortstu** с типом возвращаемого значения STUDENT, функция возвращает тот из своих двух аргументов структурного типа, элемент **id** которого меньше.

В третьем примере определена функция, возвращающая указатель на значения типа **char**. Функция принимает в качестве аргументов две символьные строки (точнее, два указателя на массивы типа **char**) и возвращает указатель на более короткую из строк.

6.2.4 Формальные параметры

Формальные параметры — это переменные, которые принимают значения, переданные функции при вызове, в соответствии с порядком следования их имен в списке параметров.

Форма объявления формальных параметров аналогична использованию метода прототипов в объявлении функции. Список объявлений параметров содержит объявления формальных параметров через запятую. После списка сразу начинается тело функции (составной оператор). Список может быть и пустым, но и в этом случае он должен быть ограничен круглыми скобками. Если функция не имеет аргументов, рекомендуется указать это явно, записав в списке объявлений параметров ключевое слово **void**.

После последнего идентификатора в списке параметров может быть записана запятая с многоточием *(,…)*. Это означает, что число параметров функции переменно, однако не меньше, чем следует идентификаторов до многоточия.

Для доступа к значениям параметров, имена которых не заданы в списке параметров функции, рекомендуется использовать макроопределения **va_arg**, **va_end**, **va_start**, описанные в разделе 12.

Допускается также список параметров, состоящий только из многоточия (…) и не содержащий идентификаторов. Это означает, что число параметров функции переменно и может быть равно нулю.

Примечание. Для совместимости с программами предыдущих версий компилятор допускает запись символа запятой без многоточия в конце списка параметров для обозначения их переменного числа. Запятая может быть использована вместо многоточия и в том случае, когда надо записать список параметров функции, принимающей нуль или более параметров. Использование запятой поддерживается только для совместимости. Для новых программ рекомендуется использовать многоточие.

Объявления параметров имеют тот же самый синтаксис, что и обычные объявления переменных (смотри раздел 3.4). Формальные параметры могут иметь базовый тип, либо быть структурой, объединением, указателем или массивом. Указание первой (или единственной) размерности для массива не обязательно. Массив воспринимается как указатель на тип элементов массива. Для формального параметра, таким образом, эквивалентны объявления

```
char s[];
char s[10];
char *s;
```

Параметры могут иметь класс памяти **auto** или **register**. Если спецификация класса памяти опущена, то подразумевается класс памяти **auto**. Если формальный параметр представлен в списке параметров, но не объявлен, то предполагается, что он имеет тип **int**. Порядок объявления формальных параметров необязательно должен совпадать с порядком их следования в списке параметров, однако для повышения читабельности программы рекомендуется следовать этому порядку.

Идентификаторы формальных параметров не могут совпадать с идентификаторами переменных, объявляемых внутри тела функции, но возможно локальное переобъявление формальных параметров внутри вложенных блоков функции.

В объявлениях формальных параметров не может быть объявлен никакой другой идентификатор, кроме перечисленных в списке параметров. Если функция имеет переменное число параметров, то программист отвечает и за определение их числа при вызове, и за получение их из стека внутри тела функции.

Тип каждого формального параметра должен соответствовать типу фактического аргумента и типу соответствующего аргумента в списке типов аргументов функции, если имеется предварительное объявление функции со списком типов аргументов. Компилятор выполняет преобразования по умолчанию отдельно над типом каждого формального параметра и над типом каждого фактического аргумента.

После преобразования все формальные параметры имеют тип размером не меньше, чем **int**, и ни один из формальных параметров не имеет тип **float**. Это означает, например, что объявление формального параметра с типом **char** эквивалентно его объявлению с типом **int**, а объявление с типом **float** эквивалентно объявлению с типом **double**.

Если используются модификаторы **near**, **far**, **huge**, то компилятор также может неявно провести преобразование аргументов-указателей. Метод преобразования в этом случае зависит от размера указателей в выбранной модели памяти и от наличия или отсутствия списка типов аргументов функции.

Тип каждого формального параметра (после преобразования) определяет, как интерпретируются размещенные в стеке аргументы. Несоответствие типов фактических аргументов типам формальных параметров может привести к неверной интерпретации. Например, если в качестве аргумента передается 16-битовый указатель, а соответствующий формальный параметр объявлен как 32-битовый, тоне 16, а 32 бита стека проинтерпретируются как аргумент. Эта ошибка повлияет не только на аргумент-указатель, но и на другие аргументы, которые следуют за ним. От ошибок такого рода может предохранить использование объявления функции со списком типов аргументов.

Пример:

```
struct student {
char name [20];
int id;
long class;
struct student *nextstu;
} student;
main(void)
{
int match(struct student *, char *);
.
.
.
if(match (student.nextstu, student.name) > 0) {
.
.
.
}
}
match (struct student *r, char *n)
{
int i = 0;
```

```
while(r->name[i] == n[i])
if(r->name[i++] == '\0')
return(r->id);
return (0);
}
```

В примере содержатся: объявление структурного типа **student**, определение главной функции, содержащей предварительное объявление функции **match** и ее вызов, и определение функции **match**. Обратите внимание на то, что одно и то же имя **student** используется без противоречия для тега структуры и имени структурной переменной.

Функция **match** объявлена с двумя аргументами. Первый аргумент — указатель на структуру типа **student**, второй — указатель на значение типа **char**.

В определении функции **match** заданы два формальных параметра, **r** и **n**. Параметр **r** объявлен как указатель на структуру типа **student**. Параметр **n** объявлен как указатель на значение типа **char**. По умолчанию, для функции **match** подразумевается тип возвращаемого значения **int**.

Функция **match** вызывается с двумя аргументами. Оба аргумента являются элементами переменной структурного типа **student** с именем **student**.

Поскольку имеется предварительное объявление функции **match**, компилятор проверит соответствие типов фактических аргументов в операторе ее вызова списку типов аргументов, а затем соответствие типов фактических аргументов типам формальных параметров. В данном случае несоответствия типов нет и в преобразованиях нет необходимости.

Обратите внимание на то, что имя массива, заданное в качестве второго аргумента в вызове функции, преобразуется по умолчанию к указателю на **char**. В функцию передается не сам массив, а адрес начала массива. Соответствующий формальный параметр также объявлен как указатель на **char**, а мог бы быть объявлен и как **char n[]**, поскольку в выражении используется как идентификатор массива. Идентификатор массива рассматривается в выражении как адресное выражение, поэтому объявление формального параметра **char *n;** эквивалентно объявлению **char n[];**.

Внутри функции объявляется локальная переменная **i**, используемая в качестве индекса массива. Функция возвращает структурный элемент **id**, если структурный элемент **name** совпал с содержимым массива **n**; в противном случае функция возвращает нулевое значение.

6.2.5 Тело функции

Тело функции представляет собой составной оператор, или блок. Он содержит операторы, которые определяют действие функции, и объявления переменных, используемых в этих операторах. Составной оператор описан в разделе 5.3.

Все переменные, объявленные в теле функции, имеют по умолчанию класс памяти **auto**, но можно явно присвоить им другой класс памяти. При вызове функции выделяется память для ее локальных переменных и, если указано, производится их инициализация. Управление передается первому оператору составного оператора. Выполнение продолжается до тех пор, пока не встретится оператор **return** или конец тела функции (составного оператора). Управление возвращается в точку вызова функции.

Если функция возвращает значение, то должен быть выполнен оператор **return**, содержащий выражение. Если оператор **return** не выполнен, или если в операторе **return** отсутствует выражение, то возвращаемое значение не определено.

6.3 Объявление функции

Объявление функции определяет ее имя, тип возвращаемого значения, класс памяти и может также задавать тип некоторых или всех аргументов функции. Детальное описание синтаксиса объявлений функции дано в разделе 3.5. В разделе 3.6 рассмотрена зависимость области действия функции от ее класса памяти.

Однако, помимо явного объявления, функция может быть объявлена неявно, по контексту ее вызова. Неявное объявление имеет место всякий раз, когда функция вызывается без предварительного объявления или определения. В этом случае компилятор языка Си считает, что вызываемая функция имеет тип возвращаемого значения **int** и класс памяти **extern**. Определение функции, если оно имеется далее в том же самом исходном файле, может переопределить тип возвращаемого значения и класс памяти.

Тип возвращаемого значения функции, указанный в предварительном объявлении, должен соответствовать типу возвращаемого значения в определении функции.

Если функция, тип возвращаемого значения которой не **int**, вызывается до ее определения или объявления, то компилятор сообщает об ошибке.

Основное назначение предварительного объявления состоит в задании типов и числа аргументов, ожидаемых в вызове функции. Список типов аргументов позволяет компилятору осуществить контроль типов аргументов при вызове функции. Если предварительное объявление отсутствует, то программист сам должен следить за соответствием типов между фактическими аргументами и формальными параметрами. Более детально контроль типов рассмотрен в разделе 6.4.1 "Фактические аргументы".

Пример:

```
main(void)
{
int a = 0, b = 1;
float x = 2.0, y = 3.0;
double realadd (double, double);
```

```
a = intadd(a, b);
x = realadd(x, y);
}
intadd(int a, int b)
{
return (a + b);
}
double realadd(double x, double y)
{
return (x + y);
}
```

В примере функция **intadd** объявлена неявно с типом возвращаемого значения **int**, так как она вызнана до своего определения. Компилятор не проверит типы аргументов при вызове функции **intadd**, поскольку список типов аргументов для нее не задан.

Функция **realadd** возвращает значение типа **double**. В функции **main** имеется предварительное объявление функции **realadd**. Тип возвращаемого значения (**double**), заданный в определении, соответствует типу возвращаемого значения, заданному в предварительном объявлении. В предварительном объявлении также определены типы двух параметров функции **realadd**. Типы фактических аргументов соответствуют типам, заданным в предварительном объявлении, и также соответствуют типам формальных параметров в определении функции **realadd**.

6.4 Вызов функции

Вызов функции передает управление и фактические аргументы (если они есть) заданной функции. Синтаксически вызов функции имеет следующий вид:

<*выражение***>**([<*список выражений*>])

<*Выражение*> вычисляется, и его результат интерпретируется как адрес функции. Выражение должно иметь тип функция.

<*Список выражений*>, в котором выражения следуют через запятую, представляет собой перечень фактических аргументов, передаваемых функции. Список выражений может быть пустым.

При выполнении вызова функции происходит присвоение значений фактических аргументов формальным параметрам. Перед этим каждый фактический аргумент вычисляется, над ним выполняются необходимые преобразования, и он копируется в стек. Первый фактический аргумент соответствует первому формальному параметру, второй — второму и т. д. Все аргументы передаются по значению, только массивы — по ссылке.

Вызванная функция работает с копией фактических аргументов, поэтому никакое изменение значений формальных параметров не отразится на значениях аргументов, с которых была сделана копия.

Передача управления осуществляется на первый оператор тела функции. Выполнение оператора **return** в теле функции возвращает в точку вызова управление и, возможно, значение. В отсутствие оператора **return** управление возвращается по достижении завершающей фигурной скобки тела функции. В этом случае возвращаемое значение не определено.

Примечание. Порядок вычисления выражений, представляющих аргументы вызова функции, не определен в языке Си, поэтому наличие побочных эффектов в этих выражениях может привести к непредсказуемым результатам. Гарантируется только то, что все побочные эффекты будут вычислены до передачи управления в вызываемую функцию.

<*Выражение*> должно ссылаться на функцию. Это означает, что функция может быть вызвана не только по идентификатору, но и через любое выражение, имеющее тип указателя на функцию.

Вызов функции синтаксически напоминает ее объявление. При объявлении функции сначала записывается ее имя, а затем список типов аргументов в скобках. При вызове также записывается имя функции, а за ним следует список выражений в скобках.

Аналогичным образом функция вызывается через указатель. Предположим, что указатель на функцию объявлен следующим образом:

```
int (*fpointer)(int, int);
```

Идентификатор **fpointer** именует указатель на функцию с двумя аргументами типа **int** и возвращаемым значением типа **int**. Вызов функции в этом случае будет выглядеть так:

```
extern int f (int, int);
fpointer = &f; /*знак & необязателен */
(*fpointer)(3,4); /* можно и просто fpointer(3,4); */
```

Примеры:

```
/* пример 1 */
double *realcomp(double, double);
double a, b, *rp;
rp = realcomp(a, b);
/* пример 2 */
main()
{
long lift(int), slep(int), drop(int);
void work(int, long (*)(int));
```

```
int select, count;
.
.
.
switch(select) {
case 1: work(count, lift); break;
case 2: work(count, step); break;
case 3: work(count, drop); break;
default: break;
}
void work(int n, long (*func)(int))
{
int i;
long j;
for(i = j = 0; i < n; i++)
j += (*func)(i); /* можно просто j += func(i); */
}
```

В первом примере объявляется, а затем вызывается функция **realcomp**, Функции передаются два аргумента типа **double**. Возвращаемое значение—указатель на переменную типа **double** — присваивается **rp**.

Во втором примере функции **work** передаются два аргумента: целая переменная **count** и адрес функции (**lift**, **step**, или **drop**). Обратите внимание на то, что адрес функции может задаваться просто указанием идентификатора функции, поскольку идентификатор функции интерпретируется как адресное выражение. Чтобы использовать идентификатор функции подобным образом, функция должна быть объявлена или определена перед использованием идентификатора, иначе идентификатор не будет распознан. Поэтому в начале функции **main** приведены объявления функций **lift**, **step**, **drop**.

В начале функции **main** задано также предварительное объявление функции **work**. В этом объявлении тип второго формального параметра задан как указатель на функцию, принимающую один аргумент типа **int** и возвращающую значение типа **long**. Скобки, заключающие символ *****, обязательны. Без них объявление специфицировало бы функцию, возвращающую указатель на значение типа **long**. Функция **work** вызывает выбранную функцию оператором

```
(*func) (i);
```

Аргумент **i** передается функции, вызываемой по указателю **func**.

6.4.1 Фактические аргументы

Фактический аргумент может быть любым значением базового типа, структурой, объединением или указателем. Все фактические аргументы передаются по значению. Массивы и функции не могут быть переданы как параметры, могут передаваться указатели на эти объекты. Поэтому массивы и функции передаются по ссылке. Значения фактических аргументов копируются в соответствующие формальные параметры. Функция использует только эти копии, не изменяя сами переменные, с которых копия была сделана.

Возможность доступа из функции не к копиям значений, а к самим переменным обеспечивают указатели. Указатель на переменную содержит ее адрес, и функция может использовать этот адрес для изменения значения переменной.

Фактические аргументы (выражения в вызове функции) вычисляются и преобразуются следующим образом:

1) Если имеется объявление со списком типов аргументов (прототип), то при вызове функции выполняются преобразования по умолчанию над типом каждого фактического аргумента, заданным в списке типов аргументов. Затем фактический аргумент приводится к полученному преобразованному типу. Независимо от аргумента, тип соответствующего формального параметра в списке параметров функции также подвергается преобразованиям по умолчанию. Затем полученный тип фактического аргумента сравнивается с типом соответствующего формального параметра. В случае несоответствия никакого преобразования не производится, но компилятор выдает такое же предупреждающее сообщение, как для выражения присваивания, когда типы левого и правого операнда не совпадают.

2) Если объявление со списком типов аргументов (прототип) отсутствует, то преобразования по умолчанию производятся отдельно для каждого аргумента, не имеющего соответствующего имени типа. Если список типов аргументов завершен многоточием и задано больше фактических аргументов, чем имен типов в списке, то лишние фактические аргументы подвергаются только преобразованиям по умолчанию. Соответствующий формальный параметр в списке параметров функции также подвергается преобразованиям по умолчанию.

Если список типов аргументов не завершен многоточием, а передается больше фактических аргументов, чем объявлено имен в списке, то компилятор выдаст предупреждающее сообщение в СП MSC и сообщение об ошибке в СП TC.

Если список типов аргументов содержит специальное имя типа **void**, то компилятор языка Си ожидает отсутствие фактических аргументов в вызове функции и отсутствие формальных параметров в определении функции. Если какое-либо из этих условий окажется нарушено, то компилятор языка Си выдает предупреждающее сообщение в СП MSC и сообщение об ошибке в СП TC.

Если в списке типов аргументов используются модификаторы **near**, **far**, **huge**, то компилятор языка Си может также выполнить неявно преобразования аргументов-указателей к соответствующему формату (см. раздел 4.7.3 "Преобразования типов при вызовах функций").

Тип каждого формального параметра подвергается преобразованиям по умолчанию. Преобразованный тип каждого формального параметра определяет, каким образом интерпретируются аргументы в стеке. Если тип формального параметра не соответствует типу фактического аргумента, то данные в стеке могут быть проинтерпретированы неверно.

Примечание. Несоответствие типов формальных и фактических параметров может привести к серьезным ошибкам, особенно когда это несоответствие влечет за собой отличия в размерах объектов. Нужно иметь в виду, что эти ошибки не выявляются, если не задан список типов аргументов в предварительном объявлении функции, причем определение функции должно находиться в области действия объявления со списком типов аргументов. Если вы создаете библиотеку функций и соответствующий включаемый файл-заголовок, содержащий списки типов аргументов для всех библиотечных функций, предназначенный для включения в программы, которые будут обращаться к этой библиотеке, рекомендуется включить этот файл-заголовок и во все библиотечные функции, чтобы отловить на этапе их компиляции противоречия между списками типов аргументов и определениями функций.

Пример:

```
main()
{
void swap(int *, int *);
int x, y;
swap(&x, &y);
}
void swap(int *a, int *b)
{
int t;
t = *a;
*a = *b;
*b = t;
}
```

В функции **main** функция **swap** объявлена как не возвращающая значения, с двумя аргументами типа указатель на **int**. Формальные параметры **a** и **b** также объявлены как указатели на **int**. При вызове функции

```
swap(&x, &y)
```

адрес **x** запоминается в **a**, адрес **y** запоминается в **b**. Выражения ***a** и ***b** в функции **swap** ссылаются на переменные **x** и **y** в **main**. Присваивания внутри функции **swap** изменяет содержимое **x** и **y**. Компилятор языка Си проведет проверку типов аргументов при вызове **swap**, поскольку в предварительном объявлении **swap** задан список типов аргументов. В примере типы фактических аргументов соответствуют и списку типов аргументов, и списку формальных параметров.

6.4.2 Вызов функции с переменным числом аргументов

Для вызова функции с переменным числом аргументов не требуется никаких специальных действий: в вызове функции просто задается то число аргументов, которое нужно. В предварительном объявлении (если оно есть) переменное число аргументов специфицируется записью запятой с последующим многоточием (,…) в конце списка типов аргументов (смотри раздел 3.5). Аналогично, список параметров в определении функции может также заканчиваться запятой с последующим многоточием (,…), что подразумевает переменное число аргументов (см. раздел 6.2.4).

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, указанных в определении функции, определяет количество аргументов, которые берутся из стека и присваиваются формальным параметрам. В случае переменного числа аргументов программист сам контролирует реальное количество аргументов, находящихся в стеке, и отвечает за выбор из стека лишних аргументов (сверх объявленных).

См. описания макроопределений **va_arg**, **va_end**, **va_start**, которые могут быть полезны при работе с переменным числом аргументов.

6.4.3 Рекурсивные вызовы

Любая функция в Си-программе может быть вызвана рекурсивно; в частности, она может вызвать сама себя. Компилятор не ограничивает число рекурсивных вызовов одной функции. При каждом вызове новые ячейки памяти выделяются для формальных параметров и локальных переменных класса памяти **auto** и **register**, так что их значения в предшествующих, незавершенных вызовах недоступны и не портятся.

Для переменных, объявленных на внутреннем уровне с классом памяти **static** или **extern**, новые ячейки памяти не выделяются при каждом рекурсивном вызове. Выделенная им память сохраняется в течение всего времени выполнения программы.

Хотя компилятор языка Си не ограничивает число рекурсивных вызовов функции, операционная среда может налагать практические ограничения. Так как каждый

рекурсивный вызов требует дополнительной стековой памяти, то слишком большое количество рекурсивных вызовов может привести к переполнению стека.

7 ДИРЕКТИВЫ ПРЕПРОЦЕССОРА И УКАЗАНИЯ КОМПИЛЯТОРУ

7.1 Введение

Препроцессор языка Си представляет собой макропроцессор, используемый для обработки исходного файла на нулевой фазе компиляции. Компилятор языка Си сам вызывает препроцессор, однако препроцессор может быть вызван и автономно. Директивы препроцессора представляют собой инструкции, записанные в исходном тексте программы на языке Си и предназначенные для выполнения препроцессором языка Си.

Директивы препроцессора обычно используются для того, чтобы облегчить модификацию исходных программ и сделать их более независимыми от особенностей различных реализаций компилятора языка Си, разных компьютеров и операционных сред. Директивы препроцессора позволяют заменить лексемы в тексте программы некоторыми значениями, вставить в исходный файл содержимое другого исходного файла, запретить компиляцию некоторой части исходного файла и т.д. Препроцессор Си распознает следующие директивы:

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

Символ # должен быть первым в строке, содержащей директиву в СП MSC версии 4. В СП MSC версии 5 ив СП TC ему могут предшествовать пробельные символы. Как в СП MSC, так и в СП TC пробельные символы допускаются между символом # и первой буквой директивы.

Некоторые директивы могут содержать аргументы. Директивы могут быть записаны в любом месте исходного файла, но их действие распространяется только от точки программы, в которой они записаны, до конца исходного файла.

Указания компилятору, или прагмы, представляют собой инструкции, записываемые в исходном тексте программы и предназначенные для управления действиями компилятора языка Си в определенных ситуациях. Набор указаний компилятору и их смысл различаются для разных компиляторов языка Си, поэтому в разделе 7.8 описывается только общий синтаксис указаний компилятору.

В рассматриваемых системах программирования есть возможность получить промежуточный текст программы после работы препроцессора, до начала собственно компиляции. В этом файле уже выполнены макроподстановки, а все строки, содержащие директивы **#define** и **#undef**, заменены на пустые строки. На место строк **#include** подставлено содержимое соответствующих включаемых файлов. Выполнена обработка директив условной компиляции **#if**, **#elif**, **#else**, **#ifdef**, **#ifndef**, **#endif**, а строки, содержащие их, заменены пустыми строками. Пустыми строками заменены и исключенные в процессе условной компиляции фрагменты исходного текста. Кроме того, в этом файле есть строки следующего вида:

#<*константа*>["имя файла"]

которые соответствуют точкам изменения номера текущей строки и/или номера файла по директивам **#line** или **#include**.

7.2 Именованные константы и макроопределения

Директива **#define** обычно используется для замены часто используемых в программе констант, ключевых слов, операторов и выражений осмысленными идентификаторами. Идентификаторы, которые заменяют числовые или текстовые константы либо произвольную последовательность символов, называются именованными константами. Идентификаторы, которые представляют некоторую последовательность действий, заданную операторами или выражениями языка Си, называются макроопределениями. Макроопределения могут иметь аргументы. Обращение к макроопределению в программе называется макровызовом.

В языке Си принято записывать идентификаторы именованных констант и макроопределений символами верхнего регистра, чтобы отличать их от имен переменных и функций. Это, однако, не является требованием языка Си.

Директива **#undef** отменяет текущее определение именованной константы. Только когда определение отменено, именованной константе может быть сопоставлено другое значение. Однако многократное повторение определения с одним и тем же значением не считается ошибкой.

Макроопределение напоминает по синтаксису определение функции. Однако замена вызова функции макровызовом может повысить скорость выполнения программы, поскольку для макроопределения не требуется генерировать вызывающую последовательность, которая

занимает относительно большое время (засылка аргументов в стек, передача управления и т.п.). С другой стороны, многократное употребление макроопределения в программе может потребовать значительно большей памяти, чем вызовы функции (код для функции генерируется один раз, а для макроопределения — столько раз, сколько имеется макровызовов в программе).

Имеется возможность задавать определения именованных констант не только в исходном тексте, но и в командной строке компиляции.

Имеется ряд предопределенных идентификаторов, которые нельзя использовать в директивах **#define** и **#undef** в качестве идентификаторов. Они рассмотрены в разделе 7.9 "Псевдопеременные".

7.2.1 Директива #define

Синтаксис:

#define <*идентификатор*> <*текст*>

#define <*идентификатор*> <*список параметров*> <*текст*>

Директива **#define** заменяет все вхождения <*идентификатора*> в исходном файле на <*текст*>, следующий в директиве за <*идентификатором*>. Этот процесс называется макроподстановкой, <*идентификатор*> заменяется лишь в том случае, если он представляет собой отдельную лексему. Например, если <*идентификатор*> является частью строки или более длинного идентификатора, он не заменяется. Если за <*идентификатором*> следует <*список параметров*>, то директива определяет макроопределение с аргументами.

<*Текст*> представляет собой набор лексем, таких как ключевые слова, константы, идентификаторы или выражения. Один или более пробельных символов должны отделять <*текст*> от <*идентификатора*> (или от заключенных в скобки параметров). Если текст не умещается на строке, то он может быть продолжен на следующей строке; для этого следует набрать в конце строки символ обратный слэш и сразу за ним нажать клавишу ENTER.

<*Текст*> может быть опущен. В этом случае все экземпляры <*идентификатора*> будут удалены из исходного текста программы. Тем не менее, сам <*идентификатор*> рассматривается как определенный и при проверке директивой **#if** дает значение 1 (смотри раздел 7.4.1).

<*Список параметров*>, если он задан, содержит один или более идентификаторов, разделенных запятыми. Идентификаторы в списке должны отличаться друг от друга. Их область действия ограничена макроопределением, в котором они заданы. Список должен быть заключен в круглые скобки. Имена формальных параметров в <*тексте*> отмечают позиции, в которые должны быть подставлены фактические аргументы макровызова. Каждое имя формального параметра может появиться в <*тексте*> произвольное число раз.

В макровызове следом за <*идентификатором*> записывается в круглых скобках список фактических аргументов, соответствующих формальным параметрам из <*списка параметров*>. <*Текст*> модифицируется путем замены каждого формального параметра на соответствующий фактический аргумент. Списки фактических аргументов и формальных параметров должны содержать одно и то же число элементов.

Примечание. Не следует путать подстановку аргументов в макроопределение с передачей аргументов функции. Подстановка в препроцессоре носит чисто текстовый характер. Никаких вычислений или преобразований типа при этом не производится.

Выше уже говорилось, что макроопределение может содержать более одного вхождения данного формального параметра. Если формальный параметр представлен выражением с побочным эффектом, то это выражение будет вычисляться более одного раза, а вместе с ним каждый раз будет возникать и побочный эффект. Результат выполнения в этом случае может быть ошибочным.

Внутрь <*текста*> в директиве **#define** могут быть вложены имена других макроопределений или констант. Их расширение производится лишь при расширении <*идентификатора*> этого <*текста*>, а не при его определении директивой **#define**. Это надо учитывать, в частности, при взаимодействии вложенных именованных констант и макроопределений с директивой **#undef**: к моменту расширения содержащего их текста они могут уже оказаться отменены директивой **#undef**.

После того как выполнена макроподстановка, полученная строка вновь просматривается для поиска других имен констант и макроопределений. При повторном просмотре не принимается к рассмотрению имя ранее произведенной макроподстановки. Поэтому директива

```
#define x x
```

не приведет к зацикливанию препроцессора.

Примеры.

```
/* пример 1 */
#define WIDTH 80
#define LENGTH (WIDTH + 10)
/* пример 2 */
#define FILEMESSAGE "Попытка создать файл\
не удалась из-за нехватки дискового пространства"
/* пример 3 */
#define REG1 register
#define REG2 register
#define REG3
/* пример 4 */
#define MAX(x, y)((x)>(y)) ? (x) : (y)
/* пример 5 */
#define MULT(a, b) ((a)*(b))
```

В первом примере идентификатор WIDTH определяется как целая константа со значением 80, а идентификатор LENGTH — как текст (WIDTH + 10). Каждое вхождение идентификатора LENGTH в исходный файл будет заменено на текст (WIDTH + 10), который после расширения идентификатора WIDTH превратится в выражение (80 + 10). Скобки, окружающие текст (WIDTH + 10), позволяют избежать ошибок в операторах, подобных следующему:

```
var = LENGTH * 20;
```

После обработки препроцессором оператор примет вид:

```
var = (80 + 10)* 20;
```

Значение, которое присваивается var, равно 1800. В отсутствие скобок в макроопределении оператор имел бы следующий вид:

```
var = 80 + 10*20;
```

Значение var равнялось бы 280, поскольку операция умножения имеет более высокий приоритет, чем операция сложения.

Во втором примере определяется идентификатор FILEMESSAGE. Его определение продолжается на вторую строку путем использования символа обратный слэш непосредственно перед нажатием клавиши ENTER.

В третьем примере определены три идентификатора, REG1, REG2, REG3. Идентификаторы REG1 и REG2 определены как ключевые слова **register**. Определение REG3 опущено и, таким образом, любое вхождение REG3 будет удалено из исходного файла. В разделе 7.4.1 приведен пример, показывающий, как эти директивы могут быть использованы для задания класса памяти **register** наиболее важным переменным программы.

В четвертом примере определяется макроопределение MAX. Каждое вхождение идентификатора MAX в исходном файле заменяется на выражение ((x)>(y))?(x):(y), в котором вместо формальных параметров x и y подставлены фактические. Например, макровызов

```
MAX(1,2)
```

заменится на выражение

```
((1)>(2))?(1):(2)
```

а макровызов

```
MAX(i, s[i])
```

заменится на выражение

```
((i)>(s(i)))?(i):(s(i))
```

Обратите внимание на то, что в этом макроопределении аргументы с побочными эффектами могут привести к неверным результатам. Например, макровызов

```
MAX(i, s[i++])
```

заменится на выражение

```
((i)>(s[i++]))?(i):(s[i++])
```

Операнды операции > могут быть вычислены в любом порядке, а значение переменной i зависит от порядка вычисления. Поэтому результат выражения непредсказуем. Кроме того, возможна ситуация, когда переменная i будет инкрементирована дважды, что, вероятно, не требуется.

В пятом примере определяется макроопределение MULT. Макровызов MULT(3,5) в тексте программы заменяется на (3)*(5). Круглые скобки, в которые заключаются фактические аргументы, необходимы в тех случаях, когда аргументы макроопределения являются сложными выражениями. Например, макровызов

```
MULT(3+4,5+6)
```

заменится на (3+4)*(5+6), что равняется 76. В отсутствие скобок результат подстановки 3+4*5+6 был бы равен 29.

7.2.2 Склейка лексем и преобразование аргументов макроопределений

СП TC и версия 5.0 СП MSC реализуют две специальные препроцессорные операции: ## и #.

В директиве **#define** две лексемы могут быть "склеены" вместе. Для этого их нужно разделить знаками ## (слева и справа от ## допустимы пробельные символы). Препроцессор объединяет такие лексемы в одну; например, макроопределение

```
#define VAR (i, j) i##j
```

при макровызове VAR(x,6) образует идентификатор x6. Некоторые компиляторы позволяют в аналогичных целях употребить запись x/**/6, но этот метод менее переносим.

Символ #, помещаемый перед аргументом макроопределения, указывает на необходимость преобразования его в символьную строку. При макровызове конструкция #<*формальный параметр*> заменяется на "<*фактический аргумент*>".

Пример: макроопределение TRACE позволяет печатать с помощью стандартной функции **printf** значения переменных типа **int** в формате <*имя*> = <*значение*>.

```
#define TRACE(flag) printf (#flag " = %d\n", flag)
```

Следующий фрагмент текста программы:

```
highval = 1024;
```

```
    TRACE (highval);
```
примет после обработки препроцессором вид:
```
    highval = 1024;
    printf("highval" " = %d\n", highval);
```
Следующие друг за другом символьные строки рассматриваются компилятором языка Си в СП MSC версии 5 и в СП TC как одна строка, поэтому полученная запись эквивалентна следующей:
```
    highval = 1024;
    printf("highval = %d\n", highval);
```
При макровызове сначала выполняется макроподстановка всех аргументов макровызова, а затем их подстановка в тело макроопределения. Поэтому следующая программа напечатает строку "отклонение от стандарта":
```
    main()
    {
    #define AB "стандарт"
    #define A "отклонение"
    #define B "от стандарта"
    #define CONCAT(P,Q) P##Q
    printf(CONCAT(A,B) "\n");
    }
```

7.2.3 Директива #undef

Синтаксис:

#undef <*идентификатор*>

Директива **#undef** отменяет действие текущего определения **#define** для <*идентификатора*>. Чтобы отменить макроопределение посредством директивы **#undef**, достаточно задать его <*идентификатор*>. Задание списка параметров не требуется.

Не является ошибкой применение директивы **#undef** к идентификатору, который ранее не был определен (или действие его определения уже отменено). Это может использоваться для гарантии того, что идентификатор не определен.

Директива **#undef** обычно используется в паре с директивой **#define**, чтобы создать область исходной программы, в которой некоторый идентификатор определен.

Пример:
```
#define WIDTH 80
#define ADD(X, Y) (X)+(Y)
#undef WIDTH
#undef ADD
```
В этом примере директива **#undef** отменяет определение именованной константы WIDTH и макроопределения ADD. Обратите внимание на то, что для отмены макроопределения задается только его идентификатор.

7.3 Включение файлов

Синтаксис:

#include "имя пути"

#include <*имя пути*>

Директива **#include** включает содержимое исходного файла, <*имя пути*> которого задано, в текущий компилируемый исходный файл. Например, общие для нескольких исходных файлов определения именованных констант и макроопределения могут быть собраны в одном включаемом файле и включены директивой **#include** во все исходные файлы. Включаемые файлы используются также для хранения объявлений внешних переменных и абстрактных типов данных, разделяемых несколькими исходными файлами.

Препроцессор обрабатывает включаемый файл таким же образом, как если бы этот файл целиком входил в состав исходного файла в точке, где записана директива **#include**. Включаемый текст также может содержать директивы препроцессора. Препроцессор выполняет обработку включаемого файла, а затем возвращается к обработке первоначального исходного файла.

Имя пути представляет собой имя файла, которому может предшествовать имя устройства и спецификация директории. Синтаксис имени пути определяется соглашениями операционной системы.

Препроцессор использует понятие стандартных директорий для поиска включаемых файлов. Стандартные директории задаются командой PATH операционной системы.

Препроцессор ведет поиск до тех пор, пока не обнаружит файл с заданным именем.

Если имя пути задано однозначно (полностью) и заключено в двойные кавычки, то препроцессор ищет файл только в директории, специфицированной заданным именем пути, а стандартные директории игнорирует.

Если заданная в кавычках спецификация не образует полное имя пути, то препроцессор начинает поиск включаемого файла в текущей рабочей директории (т. е. в той директории, которая содержит исходный файл, в котором записана директива **#include**).

Директива **#include** может быть вложенной. Это значит, что она может встретиться в файле, включенном другой директивой **#include**. Когда препроцессор обнаруживает вложенную директиву **#include**, он начинает поиск файла в текущей директории, соответствующей исходному файлу, который содержит эту вложенную директиву **#include**. После этого препроцессор переходит к поиску в текущей директории, соответствующей охватывающему исходному файлу, т.е. тому, по отношению к которому данная директива **#include** является вложенной. Допустимый уровень вложенности директив **#include** зависит от реализации компилятора. Процесс поиска в охватывающих директориях продолжается до тех пор, пока не будет просмотрена текущая директория самого первого исходного файла, т. е. файла, имя которого было задано при вызове компилятора языка Си.

Затем препроцессор продолжает поиск в директориях, указанных в командной строке компиляции, и, наконец, ищет в стандартных директориях.

Если же имя пути заключено в угловые скобки, то препроцессор вообще не будет осуществлять поиск в текущей рабочей директории, а сразу начнет поиск в директориях, специфицированных в командной строке компиляции, а затем в стандартных директориях.

Примеры:

```
#include <stdio.h>      /* пример 1 */
#include "defs.h"       /* пример 2 */
```

В первом примере в исходный файл включается файл с именем **stdio.h**. Угловые скобки сообщают препроцессору, что поиск файла нужно осуществлять в директории, указанной в командной строке компиляции, а затем в стандартных директориях.

Во втором примере в исходный файл включается файл с именем **defs.h**. Двойные кавычки означают, что при поиске файла сначала должна быть просмотрена директория, содержащая текущий исходный файл.

В СП ТС имеется возможность задавать имя пути в директиве **#include** с помощью именованной константы. Если за словом **include** следует идентификатор, препроцессор проверяет, не именует ли он константу или макроопределение. Если же за словом **include** следует строка, заключенная в кавычки или в угловые скобки, СП ТС не будет искать в ней имя константы.

Примеры:

```
#define myinclude "c:\tc\include\mystuff.h"
#include myinclude
#include "myinclude.h"
```

Первая директива **#include** заставит препроцессор просматривать директорию C:\TC\INCLUDE\MYSTUFF.H, а вторая заставит искать файл MYINCLUDE.H в текущей директории.

Объединение символьных строк и склейку лексем в именованной константе, которая используется в директиве **#include**, использовать нельзя. Результат расширения константы должен сразу читаться как корректная директива **#include**.

7.4 Условная компиляция

В этом разделе описываются директивы, которые управляют условной компиляцией. Эти директивы позволяют исключить из процесса компиляции какие-либо части исходного файла посредством проверки условий (константных выражений).

7.4.1 Директивы #if, #elif, #else, #endif

Синтаксис:

#if <*ограниченное-константное-выражение*> [<*текст*>]
[#elif **<***ограниченное-константное-выражение***>** <*текст*>]
[#elif **<***ограниченное-константное-выражение***>** <*текст*>]
[#else **<***текст***>**]
#endif

Директива **#if** совместно с директивами **#elif**, **#else** и **#endif** управляет компиляцией частей исходного файла. Каждой директиве **#if** в том же исходном файле должна соответствовать завершающая ее директива **#endif**. Между директивами **#if** и **#endif** допускается произвольное количество директив **#elif** (в том числе ни одной) и не более одной директивы **#else**. Если директива **#else** присутствует, то между ней и директивой **#endif** на данном уровне вложенности не должно быть других директив **#elif**.

Препроцессор выбирает один из участков <*текста*> для обработки. <*Текст*> может занимать более одной строки. Обычно это участок программного текста, однако это не обязательно: препроцессор можно использовать для обработки произвольного текста. Если

<*текст*> содержит директивы препроцессора (в том числе и директивы условной компиляции), то эти директивы выполняются. Обработанный препроцессором текст передается на компиляцию.

Участок текста, не выбранный препроцессором, игнорируется на стадии препроцессорной обработки и не компилируется.

Препроцессор выбирает участок текста для обработки на основе вычисления <*ограниченного-константного-выражения*>, следующего за каждой директивой **#if** или **#elif**. Выбирается <*текст*>, следующий за <*ограниченным-константным-выражением*> со значением истина (не нуль), вплоть до ближайшей директивы **#elif**, **#else**, или **#endif**, ассоциированной с данной директивой **#if**.

Если ни одно ограниченное константное выражение не истинно, то препроцессор выбирает <*текст*>, следующий за директивой **#else**. Если же директива **#else** отсутствует, то никакой текст не выбирается.

Ограниченное константное выражение описано в разделе 4.2.9 "Константные выражения". Такое выражение не может содержать операцию **sizeof** (в СП ТС — может), операцию приведения типа, константы перечисления и плавающие константы, но может содержать препроцессорную операцию **defined**(<*идентификатор*>). Эта операция дает истинное (не равное нулю) значение, если заданный <*идентификатор*> в данный момент определен; в противном случае выражение ложно (равно нулю). Следует помнить, что идентификатор, определенный без значения, тем не менее рассматривается как определенный. Операция **defined** может использоваться в сложном выражении в директиве **#if** неоднократно:

```
#if defined(mysym) || defined(yoursym)
```

СП ТС (в отличие от СП MSC) позволяет использовать операцию **sizeof** в ограниченном константном выражении для препроцессора. В следующем примере в зависимости от размера указателя определяется одна из констант — либо SDATA, либо LDATA:

```
#if (sizeof(void *) == 2)
#define SDATA
#else
#define LDATA
#endif
```

Директивы **#if** могут быть вложенными. При этом каждая из директив **#else**, **#elif**, **#endif** ассоциируется с ближайшей предшествующей директивой **#if**.

Примеры:

```
/* пример 1 */
#if defined(CREDIT)
credit();
#elif defined (DEBIT)
debit();
#else
printerror();
#endif
/* пример 2 */
#if DLEVEL > 5
#define SIGNAL 1
#if STACKUSE == 1
#derine STACK 200
#else
#define STACK 100
#endif
#else
#define SIGNAL 0
#if STACKUSE == 1
#define STACK 100
#else
#define STACK 50
#endif
#endif
/* пример 3 */
#if DLEVEL == 0
#define STACK 0
#elif DLEVEL == 1
#define STACK 100
#elif DLEVEL > 5
display(debugptr);
#else
#define STACK 200
#endif
/* пример 4 */
#define REG 1 register
#define REG2 register
```

```
#if defined (M_86)
#define REG3
#define REG4
#else
#ifdefined(M_68000)
#define REG4 register
#endif
#endif
```

В первом примере директивы **#if**, **#elif**, **#else**, **#endif** управляют компиляцией одного из трех вызовов функции. Вызов функции **credit** компилируется, если определена именованная константа CREDIT. Если определена именованная константа DEBIT, то компилируется вызов функции **debit**. Если ни одна из .именованных констант не определена, то компилируется вызов функции **printerror**. Следует учитывать, что CREDIT и **credit** являются различными идентификаторами в языке Си.

В следующих двух примерах предполагается, что константа DLEVEL предварительно определена директивой **#define**.

Во втором примере показаны два вложенных набора директив **#if**, **#else**, **#endif**. Первый набор директив обрабатывается, если значение DLEVEL больше 5. В противном случае обрабатывается второй набор.

В третьем примере директивы уловной компиляции используют для выбора текста значение константы DLEVEL. Константа STACK определяется со значением 0, 100 или 200, в зависимости от значения DLEVEL. Если DLEVEL больше 5, то компилируется вызов функции **display**, а константа STACK не определяется.

В четвертом примере директивы препроцессора используются для контроля за применением спецификации регистрового класса памяти в программе, предназначенной для работы в различных операционных средах.

Компилятор обычно выделяет регистровую память переменным в том порядке, в котором записаны объявления переменных в программе. Если программа содержит больше объявлений переменных класса памяти **register**, чем имеется регистров в данной операционной среде, то регистровую память получат только те переменные, объявления которых записаны раньше. Следовательно, если более интенсивно будут использоваться те переменные, которые объявлены позже, выигрыш в эффективности от использования регистров окажется незначительным.

В примере показано, каким образом предоставить приоритет регистровой памяти наиболее важным переменным. Именованные константы REG1 и REG2 определяются как ключевые слова **register**. Они предназначены для объявления двух наиболее важных локальных переменных функции. Например, в следующем фрагменте программы такими переменными являются b и c.

```
func(REG3 int a)
{
REG1 int b;
REG2 int c;
REG4 int d;
}
```

Если определена константа M_86, препроцессор удаляет идентификаторы REG3 и REG4 из файла путем замены их на пустой текст. Регистровую память в этом случае получат только переменные b и c. Если определен идентификатор M_68000, то все четыре переменные объявляются с классом памяти **register**.

Если не определена ни одна из констант — ни M_86, ни M_68000, — то регистровую память получат переменные a, b и c.

7.4.2 Директивы #ifdef и #ifndef

Синтаксис:

#ifdef <*идентификатор*>

#ifndef <*идентификатор*>

Аналогично директиве **#if**, за директивами **#ifdef** и **#ifndef** может следовать набор директив **#elif** и директива **#else**. Набор должен быть завершен директивой **#endif**.

Использование директив **#ifdef** и **#ifndef** эквивалентно применению директивы **#if**, использующей выражение с операцией **defined**(<*идентификатор*>). Эти директивы поддерживаются исключительно для совместимости с предыдущими версиями компиляторов языка Си. Для новых программ рекомендуется использовать директиву **#if** с операцией **defined**(<*идентификатор*>).

Когда препроцессор обрабатывает директиву **#ifdef**, он проверяет, определен ли в данный момент <*идентификатор*> директивой **#define**. Если да, условие считается истинным, если нет — ложным.

Директива **#ifndef** противоположна по действию директиве **#ifdef**. Если <*идентификатор*> не был определен директивой **#define**, или его определение уже отменено директивой **#undef**, то условие считается истинным. В противном случае условие ложно.

7.5 Управление нумерацией строк

Синтаксис:

#line <*константа*> ["имя-файла"]

Директива **#line** сообщает компилятору языка Си об изменении имени исходного файла и порядка нумерации строк. Это изменение отражается только на диагностических сообщениях компилятора: исходный файл будет теперь именоваться как <*имя-файла*>, а текущая компилируемая строка получит номер <*константа*>. После обработки очередной строки счетчик номеров строк увеличивается на единицу. В случае изменения номера

строки и имени исходного файла директивой **#line** компилятор "забывает" их прежние значения и продолжает работу уже с новыми значениями.

Директива **#line** обычно используется автоматическими генераторами программ для того, чтобы диагностические сообщения относились не к исходному файлу, а к сгенерированной программе.

<*Константа*> в директиве **#line** может быть произвольной целой константой. <*Имя-файла*> может быть произвольной комбинацией символов, заключенной в двойные кавычки. Если имя файла опущено, то имя исходного файла остается прежним.

Текущий номер строки и имя исходного файла доступны в программе через псевдопеременные с именами __LINE__ и __FILE__. Эти псевдопеременные могут быть использованы для выдачи во время выполнения сообщений о точном местоположении ошибки.

Значением псевдопеременной __FILE__ является строка, представляющая имя файла, заключенное в двойные кавычки. Поэтому для печати имени исходного файла не требуется заключать сам идентификатор __FILE__ в двойные кавычки.

Примеры.

```
/* пример 1 */
#line 151 "copy.c"
/* пример 2 */
#define ASSERT(cond) if (!cond)\
{printf ("ошибка в строке %d файла %s\n", \
__LINE__, __FILE__);} else;
```

В первом примере устанавливается имя исходного файла copy.c и текущий номер строки 151.

Во втором примере в макроопределении ASSERT используются псевдопеременные __LINE__ и __FILE__ для печати сообщения об ошибке, содержащего координаты исходного файла, если некоторое условие, заданное макроаргументом **cond**, ложно.

7.6 Директива обработки ошибок

В СП ТС реализована директива #error. Ее формат:

#error <*текст*>

Обычно эту директиву записывают среди директив условной компиляции для обнаружения некоторой недопустимой ситуации. По директиве **#error** препроцессор прерывает компиляцию и выдает следующее сообщение:

Fatal: <*имя-файла*> <*номер-строки*> Error directive: <*текст*>

Fatal — признак фатальной ошибки; <*имя-файла*> — имя исходного файла; <*номер-строки*> — текущий номер строки; Error directive — сообщение об ошибке в директиве; <*текст*> — собственно текст диагностического сообщения.

Например, если именованная константа MYVAL может иметь значение либо 0, либо 1, можно поместить в исходный файл операторы условной компиляции для проверки на некорректное значение MYVAL:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL должно иметь значение либо 0, либо 1
#endif
```

Препроцессор просматривает текст сообщения в директиве **#error**, и исключает из него комментарии (если они имеются), но именованные константы и макроопределения в тексте не выявляет и макроподстановку не производит.

7.7 Пустая директива

Для повышения читабельности программ СП ТС распознает пустую директиву, состоящую из строки, содержащей просто знак #. Эта директива всегда игнорируется.

7.8 Указания компилятору языка Си

Синтаксис:

#pragma **<***последовательность-символов***>**

Указания компилятору, или прагмы, предназначены для исполнения компилятором в процессе его работы. <*Последовательность-символов*> задает определенную инструкцию компилятору и, возможно, аргументы.

Набор прагм для каждого компилятора языка Си различен. Для получения подробной информации о прагмах смотрите системную документацию по используемому вами компилятору.

7.9 Псевдопеременные

Псевдопеременные представляют собой зарезервированные именованные константы, которые можно использовать в любом исходном файле. Каждый из них начинается и оканчивается двумя символами подчеркивания (__).

__LINE__

Номер текущей обрабатываемой строки исходного файла—десятичная константа. Первая строка исходного файла имеет номер 1.

__FILE__

Имя компилируемого исходного файла — символьная строка. Значение данной псевдопеременной изменяется каждый раз, когда компилятор обрабатывает директиву **#include** или директиву **#line**, а также по завершении включаемого файла.

Следующие две псевдопеременные поддерживаются только СП TC.

__DATE__

Дата начала компиляции текущего исходного файла — символьная строка. Каждое вхождение __DATE__ в заданный файл дает одно и то же значение, независимо от того, как долго уже продолжается обработка. Дата имеет формат **mmm dd УУУУ**, где **mmm** — месяц (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec), **dd** — число текущего месяца (1…31; в 1-й позиции **dd** ставится пробел, если число меньше 10), **уууу** — год (например, 1990).

__TIME__

Время начала компиляции текущего исходного файла — символьная строка. Каждое вхождение __TIME__ в заданный файл дает одно и то же значение, независимо от того, как долго уже продолжается обработка. Время имеет формат **hh:mm:ss**, где **hh** — час (00…23), **mm** — минуты (00…59), **ss** — секунды (00…59).

8 МОДЕЛИ ПАМЯТИ

Реализация моделей памяти в СП MSC и в СП TC имеет ряд отличий. В разделах 8.1 и 8.2 описаны модели памяти СП MSC, а в разделе 8.3 приведены отличия моделей памяти СП TC.

8.1 Виды моделей

Применение моделей памяти позволяет контролировать распределение памяти в программе и делать его более эффективным или адекватным решаемой задаче. По умолчанию в процессе компиляции и редактирования связей генерируется код для работы в малой (**small**) модели. Для большинства программ этой модели достаточно. Существуют, однако, два условия, когда малая модель не годится; если программа удовлетворяет хотя бы одному из них, следует использовать другую модель памяти:

—размер кода программы превышает 64 Кбайта;

—размер статических данных программы превышает 64 Кбайта.

Имеется два варианта выбора модели памяти для программы: назначить при компиляции новую модель вместо действующей по умолчанию малой либо использовать в объявлении объектов программы модификаторы **near**, **far**, **huge**. Можно также комбинировать эти способы.

Архитектура микропроцессора типа 8086/8088 предусматривает разбиение оперативной памяти на физические сегменты. Размер одного сегмента не превышает 64 Кбайта. Минимальное количество сегментов, которое выделяется программе, равно двум: один для кода, другой для статических данных. Эти сегменты называются стандартными. Малая модель памяти использует только эти два сегмента. Другие модели позволяют выделять программе более одного сегмента кода и/или данных.

Статические данные — это все данные, объявленные в программе с классом памяти **extern** или **static**. Формальные параметры функций и локальные переменные функций и блоков не являются статическими данными. Они хранятся не в сегменте данных, а в сегменте стека. Он обычно совмещен со стандартным сегментом данных физически.

Помимо статических данных, имеется возможность работать с динамической памятью с помощью стандартных библиотечных функций типа **malloc**. Динамическая память может выделяться как в отдельном сегменте (дальняя динамическая память), так и в стандартном сегменте данных, между концом статических данных и стеком (ближняя динамическая память).

Адрес оперативной памяти состоит из двух частей:

1) 16-битового числа, представляющего базовый адрес сегмента;

2) 16-битового числа, представляющего смещение внутри этого сегмента.

Для доступа к коду или данным, находящимся в стандартном сегменте, достаточно использовать только вторую часть адреса, т.е. смещение. В этом случае можно применить указатель, объявленный с модификатором near (ближний). Поскольку для доступа к объекту используется только одно 16-битовое число, применение указателей типа near компактно по занимаемой памяти и быстро по времени.

Если код или данные располагаются за пределами стандартных сегментов, для доступа к ним должны использоваться обе части адреса — и адрес сегмента, и смещение. Указатели для такого доступа объявляются с модификатором **far** (дальний). Доступ к объектам по указателям типа **far** занимает больше памяти и времени, однако позволяет адресовать всю оперативную память, а не только 64 Кбайта.

Имеется третий вид указателей — **huge** (максимальный). Адрес типа **huge** подобен адресу типа **far**, поскольку оба включают и адрес сегмента, и смещение. Однако адресная арифметика для **far** и **huge** адресов различается. Поскольку объекты, адресуемые **far** указателями, не выходят за границу адресуемого сегмента, действия адресной арифметики выполняются только над второй половиной адреса — над смещением. Это ускоряет доступ, однако ограничивает размер одного программного объекта 64 Кбайтами. Для указателей типа **huge** арифметические действия выполняются над всеми 32 битами адреса.

Тип адреса **huge** определен только для данных (массивов); никакой сегмент кода, т.е. никакой из исходных файлов, составляющих программу, не может сгенерировать больше 64 Кбайтов кода. Поэтому ключевое слово **huge** применимо только к элементам данных — массивам и указателям на них.

Малая модель

В малой (**small**) модели памяти программа занимает два стандартных сегмента: сегмент кода и сегмент данных, в котором размещен также стек. Как код, так и данные программы не могут превышать 64 Кбайтов; следовательно, суммарный размер программы не может превышать 128 Кбайтов. Малая модель подходит для большинства программ и потому назначается компилятором по умолчанию.

В малой модели для доступа к объектам кода или данных используются указатели типа **near**. Можно, однако, изменить это умолчание, применяя модификаторы **far** или **huge** для объявления элементов данных и модификатор **far** для функций.

Средняя модель

В средней (**medium**) модели памяти для данных и стека программы выделяется один сегмент, а для кода — столько сегментов, сколько потребуется. Каждому исходному модулю программы выделяется собственный сегмент кода.

Средняя модель применяется обычно для программ с большим количеством операторов (более 64 Кбайтов кода), но сравнительно небольшим размером данных (менее 64 Кбайтов). Для доступа к функциям по умолчанию используются указатели типа **far**, для доступа к данным — указатели типа **near**. Можно, однако, изменить это умолчание, применяя модификаторы **far** или **huge** для объявления элементов данных и модификатор **near** для функций.

Средняя модель представляет разумный компромисс между скоростью выполнения и компактностью программы, поскольку большинство программ чаще обращается к данным, чем к функциям.

Компактная модель

В компактной (**compact**) модели программному коду выделяется только один сегмент, а данным — столько сегментов, сколько потребуется. Компактная модель применяется для программ, небольших по количеству операторов, но работающих с большим объемом данных.

В компактной модели доступ к коду (функциям) производится по указателям типа **near**, а к данным — по указателям типа **far**. Это умолчание можно обойти, используя модификаторы **near** и **huge** для объявления данных и модификатор **far** для функций.

Большая модель

В большой (**large**) модели и под код, и под данные выделяется несколько сегментов. Большая модель используется для больших программ с большим объемом данных.

В большой модели доступ к элементам кода и данных производится по указателям типа **far**. Это умолчание можно обойти, используя модификаторы **near** и **huge** для объявления данных и модификатор **near** для функций.

Максимальная модель

Максимальная (**huge**) модель аналогична большой модели, за исключением того, что в ней снимается ограничение на размер массивов (указатели типа **far**, применяемые в большой модели, ограничивают размер отдельного элемента данных 64 Кбайтами). Некоторые ограничения, однако, налагаются на размер элементов **huge** массивов, если эти массивы превышают по размеру 64 Кбайта. В целях повышения эффективности адресации не допускается пересечения элементами массива границ сегмента. Из этого вытекает следующее:

1) Никакой элемент массива не может превышать по размеру 64 Кбайта.

2) Если размер массива больше 128 Кбайтов, размер его элементов (в байтах) должен быть степенью двойки (т. е. 2, 4, 8, 16 и т.д.). Если же размер массива меньше или равен 128 Кбайтам, то размер его элементов может быть от 1 байта до 64 Кбайтов (включительно).

Работая в максимальной модели, программист должен быть осторожен в применении операции **sizeof** и при вычитании указателей. В языке Си определено, что значение операции **sizeof** имеет тип **unsigned int**, однако число байтов в **huge** массиве может быть представлено только типом **unsigned long**. Для получения правильного значения в этом случае следует применять приведение типа операции **sizeof**:

```
(unsigned long)sizeof(huge_item)
```

Аналогично, результат вычитания указателей определен в языке Си как значение типа **int**. При вычитании указателей типа **huge** может оказаться, что результат имеет тип **long**. В этом случае также необходимо применить приведение типа:

```
(long)(huge_ptr1—huge_ptr2)
```

8.2 Модификация стандартной модели памяти

Работая в некоторой стандартной модели памяти, программист может в той или иной мере модифицировать ее, применяя в объявлениях модификаторы **near**, **far** и **huge**. Правила интерпретации объявлений с модификаторами рассмотрены в разделе 3.3.3.4 "Модификаторы **near**, **far**, **huge**".

8.2.1 Объявление данных

Если непосредственно за ключевым словом **near**, **far** или **huge** следует идентификатор, то это значит, что соответствующий элемент данных будет размещен в стандартном сегменте (для **near**) или может быть размещен в другом сегменте данных (для **far** или **huge**). Например, объявление

```
char far x;
```

сообщает, что адрес объекта **x** имеет тип **far**.

Если же непосредственно за ключевым словом **near**, **far** или **huge** следует признак указателя (звездочка), то это значит, что соответствующий указатель будет хранить адрес типа **near**, типа **far** или типа **huge**, соответственно. Например, объявление

```
char far *p;
```

сообщает, что указатель **p** имеет тип **far**, т. е. может указывать на объект, расположенный в любом сегменте данных (при этом тип адреса этого объекта должен быть **far**). Объявление

```
char * far p;
```

объявляет **p** как указатель на **char**, причем сам указатель **p** может находиться в любом сегменте, и его адрес имеет тип **far**. Объявление

```
char far * far p;
```

сообщает, что указатель **p** может указывать на объекты с адресом типа **far**. Адрес самого указателя **p** также имеет тип **far**.

Примеры:

```
char a[3000];          /* пример 1: малая модель */
char far b[30000];     /* пример 2: малая модель */
char a[3000];          /* пример 3: большая модель */
char near b[3000];     /* пример 4: большая модель */
char huge a[70000];    /* пример 5: малая модель */
char huge *pa;         /* пример 6: малая модель */
char *pa;              /* пример 7: малая модель */
char far *pb;          /* пример 8: малая модель */
char far **pa;         /* пример 9: малая модель */
char far **pa;         /* пример 10: большая модель */
char far *near *pb;    /* пример 11: любая модель */
char far *far *pb;     /* пример 12: любая модель */
```

В примере 1 массиву **a** выделяется память в стандартном сегменте данных; массиву **b** во втором примере память может быть выделена в любом из сегментов данных программы. Поскольку оба объявления сделаны в малой модели, то, вероятно, массив **a** содержит часто используемые данные, которые для ускорения доступа должны располагаться в стандартном сегменте, а массив **b** содержит редко используемые данные, которые могут выйти за пределы 64-Кбайтного сегмента данных. Можно было бы использовать здесь другую модель памяти, в которой адрес данных по умолчанию имел бы тип **far**, однако для сохранения быстрого доступа к массиву **a** лучше сохранить малую модель, а адрес массива **b** объявить как **far**.

В примере 2 указан большой размер массива **b**, поскольку более вероятно, что программист будет модифицировать тип адреса объекта большой длины, который может не поместиться в текущий сегмент.

В примере 3, очевидно, скорость доступа к массиву **a** не является критичной; независимо от того, попадет он в стандартный сегмент или не попадет, обращение к нему всегда будет осуществляться по 32-битовому адресу. В примере 4 массиву **b** с помощью модификатора **near** явно назначен стандартный сегмент, с целью ускорения доступа к нему в большой модели.

В примере 5 массив **a** должен быть явно объявлен как **huge**, поскольку его размер превышает 64 Кбайта. Использование модификатора **huge** вместо выбора максимальной модели памяти в качестве стандартной позволяет

сэкономить время доступа: только к массиву **a** обращение будет осуществляться по адресу типа **huge**, а все остальные данные будут размещаться в стандартном сегменте. Для обращения к массиву **a** может быть использован указатель **pa** из примера 6. Все арифметические операции над указателем **pa** (например, **pa++**) будут выполняться над всеми 32 его битами.

В примере 7 **pa** объявляется как указатель на **near char**. Указатель получает тип **near** по умолчанию, поскольку речь идет о малой модели. В примере 8 **pb** явно объявляется как указатель на **far char**. Он может быть использован, в частности, для доступа к символьному массиву, расположенному не в стандартном сегменте памяти. Например, **pa** может указывать на массив **a** из примера 1, а **pb** — на массив **b** из примера 2.

Хотя объявления **pa** в примерах 9 и 10 идентичны, в примере 9 **pa** объявляется как указатель на **near** массив указателей на тип **far char**, а в примере 10 **pa** объявляется как указатель на **far** массив указателей на тип **far char**.

В примере 11 **pb** объявляется как указатель на **near** массив указателей на тип **far char**. В примере 12 **pb** объявляется как указатель на **far** массив указателей на тип **far char**. В этих примерах употребление слов **far** и **near** изменяет действующие по умолчанию соглашения, связанные с моделями памяти; в отличие от примеров 9 и 10, объявления **pb** не зависят от выбранной модели памяти и в любой модели имеют одинаковый смысл.

8.2.2 Объявление функций

Правила применения модификаторов **near** и **far** в объявлениях функций аналогичны правилам применения их в объявлениях данных. Если непосредственно за модификатором следует имя функции, то данное ключевое слово определяет, в каком сегменте будет размещена функция. Например,

```
char far fun();
```

определяет **fun** как функцию, вызываемую по 32-битовому адресу и возвращающую тип **char**.

Если же непосредственно за специальным ключевым словом следует признак указателя (звездочка), то данное ключевое слово определяет тип адреса функций, которые могут вызываться через этот указатель. Например,

```
char (far *pfun)();
```

определяет **pfun** как указатель (32-битовый) на **far** функцию, возвращающую **char**.

Модификатор **huge** к функциям и указателям на функции неприменим.

Объявления функций должны соответствовать их определениям по набору и расположению модификаторов. Рекомендуется всегда использовать предварительные объявления функций со списками типов аргументов, чтобы компилятор мог выявить ситуации некорректного вызова функции.

Примеры:

```
char far fun();               /* пример 1: малая модель */
static char far *near fun();  /* пример 2: большая модель */
void far fun();               /* пример 3: малая модель */
void (far *pfun)() = fun;
double far * far fun();       /* пример 4: компактная модель */
double far* (far *pfun)() = fun;
```

В первом примере **fun** объявляется как функция, возвращающая **char**. Ключевое слово **far** в объявлении означает, что **fun** вызывается по 32-битовому адресу типа **far**.

Во втором примере **fun** объявляется как **near** функция класса памяти **static**, возвращающая указатель на **far char**. Такая функция в большой модели памяти может быть использована, например, как вспомогательная подпрограмма, которая вызывается часто, но только функциями из своего исходного файла. Поскольку все функции из одного исходного файла помещаются в один и тот же сегмент, они могут обращаться друг к другу по адресам типа **near**. Будет ошибкой, однако, передать адрес функции **fun** в качестве аргумента другой функции, расположенной за пределами сегмента, в котором определена **fun**, поскольку из другого сегмента функция **fun** не может быть вызвана.

В третьем примере **pfun** объявляется как указатель на **far** функцию, не возвращающую значения, а затем ему присваивается адрес функции **fun**. Фактически **pfun** может быть использован для доступа к любой функции, имеющей тип адреса **far**. Следует понимать, что если функция, вызванная через указатель **pfun**, не была объявлена с модификатором **far**, или не получила тип **far** по умолчанию, то ее вызов приведет к ошибке во время выполнения.

В примере 4 **pfun** объявляется как указатель на **far** функцию, возвращающую указатель на **far double**, после чего ему присваивается адрес функции **fun**. Такой вариант может использоваться, например, в компактной модели памяти для функции, которая используется редко, и потому необязательно должна находиться в стандартном сегменте кода. И функция, и указатель должны быть объявлены с модификатором **far**.

8.3 Модели памяти СП TC

Организация работы с моделями памяти в СП TC имеет ряд отличий от СП MSC.

В дополнение к описанным выше моделям СП TC имеет еще одну — **tiny** (минимальную). В этой модели вся программа — код, данные, стек, динамическая память — размещается в одном сегменте. Таким образом, размер программы ограничен 64 Кбайтами. Все указатели в этой модели имеют тип **near**. Программы модели памяти **tiny** могут быть преобразованы в формат выполняемых файлов COM операционной системы MSDOS.

В компактной и большой моделях статические данные занимают не несколько сегментов, а один. В этом сегменте также размещается стек. По умолчанию для данных используются указатели типа **far**, однако для статических данных лучше применять указатели типа **near**. В компактной, большой и максимальной моделях стек занимает отдельный сегмент.

Только в максимальной модели памяти СП TC позволяет статическим данным занимать более одного сегмента, т. е. более 64 Кбайтов. Однако в одном исходном файле может быть не более 64 Кбайтов статических данных. Вследствие этого недопустимы массивы, превышающие по размеру 64 Кбайта.

Указатели типа **huge** в СП TC всегда хранятся в нормализованном виде, так что значение смещения никогда не превышает 15. Нормализация требует дополнительных временных затрат при работе с указателями тип **huge**, но зато позволяет применять к ним операции отношения (с ненормализованными 32-битовыми указателями операции отношения работают некорректно).

Допустимо применение модификатора **huge** к функциям и указателям на функции.

Имеется также четыре специальных указателя типа near с именами **_cs**, **_ds**, **_ss**, **_es**. Это 16-битовые указатели, ассоциированные с сегментными регистрами микропроцессора, содержащими адреса сегментов кода, данных, стека и динамической памяти, соответственно. Например, указатель **p**, объявленный следующим образом

```
char _ss *p;
```

будет содержать 16-битовое значение, хранящееся в сегменте стека.

Объявления с модификаторами **near**, **far**, **huge** отличаются тем, что нельзя модифицировать тип адреса самого указателя. Модификатор в объявлении указателя может стоять только перед звездочкой, тем самым объявляя указатель на модифицируемый тип. А объявление

```
int * far p;
```

допустимое в СП MSC, считается ошибкой в СП TC.