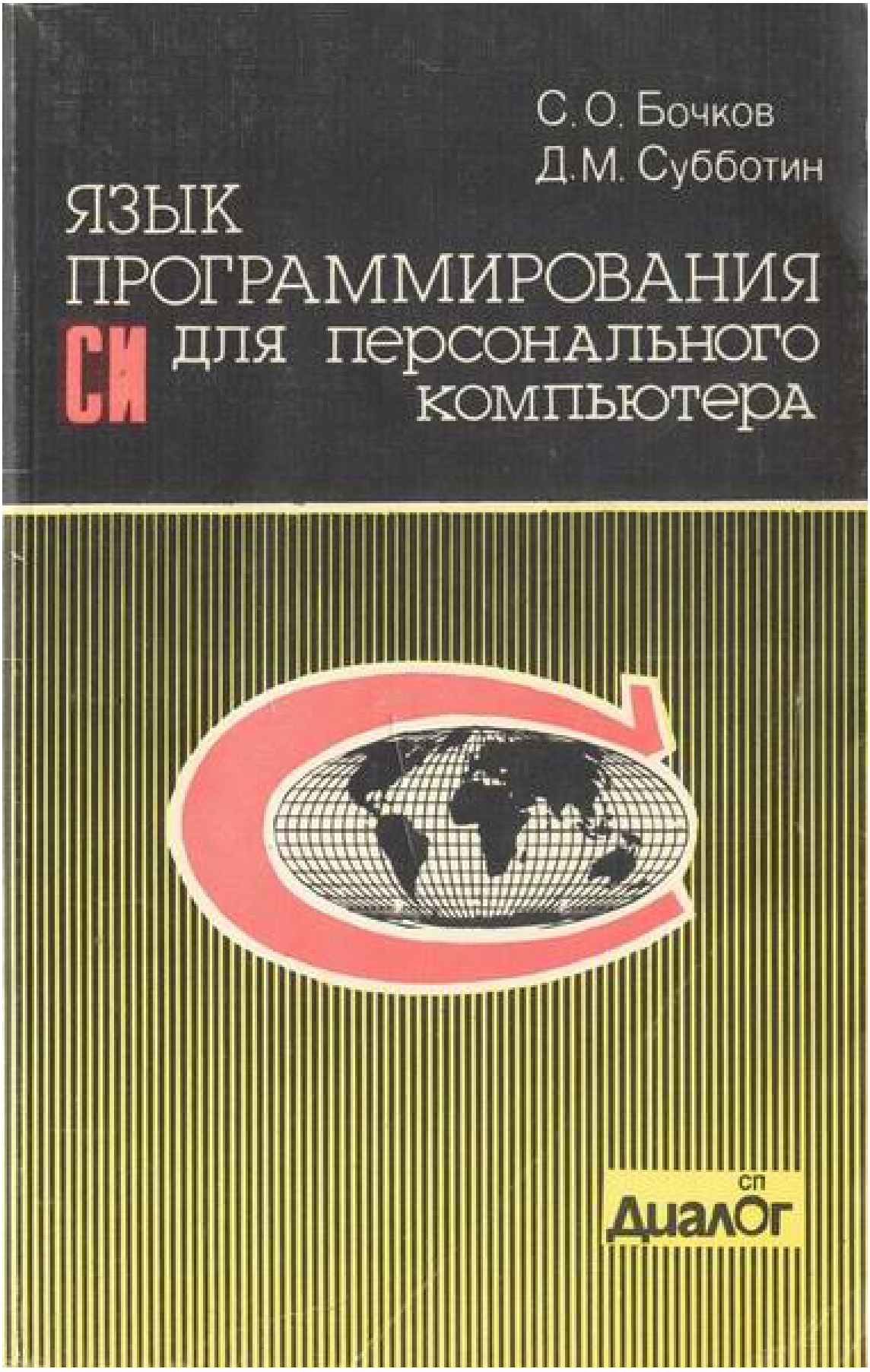

Бочков С.О., Субботин Д.М. Язык программирования Си для персонального компьютера. - М:. Радио и связь, 1990. - 384 с.

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 констант: 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—0x7FF	int
	0100000—0177777	0x8000—0xFFFF	unsigned int
32767—2147483647	02000001—017777777777	0x10000—0x7FFFFFF	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
01	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 а
'?'	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 TC 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**.

3.3.3.1Интерпретация 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

3.3.3.2Модификаторы 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 TC 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 TC 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 ТС 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 ТС 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 ТС) 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 ы 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 0xAA5a, 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 <= y         /* выражение 3 */
x >= y         /* выражение 4 */
x == y         /* 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** double 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 ТС 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 ТС 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 ТС 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 ТС 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 ТС.

ЧАСТЬ II

СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА СИ

В языке Си стандартная библиотека более сильно интегрирована с языком по сравнению с другими языками программирования высокого уровня. Без использования функций стандартной библиотеки не может быть написана ни одна серьезная программа на языке Си, в частности потому, что в самом языке Си нет никаких средств ввода/вывода информации.

Стандартную библиотеку функций языка Си можно разделить на две категории: функции, которые имеются в библиотеке любой системы программирования языка Си для различных операционных систем и различных архитектур компьютеров, и функции, которые являются уникальными в рамках какой-либо системы программирования, или обеспечивают доступ к специфическим возможностям конкретной операционной системы, или связаны с конкретной архитектурой компьютера.

Функции первой категории образуют переносимое ядро библиотеки; программы, в которых используются только такие библиотечные функции, могут быть перенесены в другую систему программирования, другую операционную систему и/или на другой тип архитектуры компьютеров с наименьшими затратами. Плата за универсальность и переносимость — невозможность воспользоваться специфическими средствами, предоставляемыми конкретной вычислительной средой.

Функции второй категории предоставляют возможность получить доступ к функциям ядра данной операционной системы, к внутренним структурам данных операционной системы, к регистрам используемых аппаратных устройств. Кроме того, ко второй категории относятся функции, которые добавлены в библиотеку, исходя из вкусовых привязанностей разработчиков конкретной системы программирования — как им видится удобный набор средств для разработки различных алгоритмов (сравните, например, функции setmem и memset). В современных системах программирования Си в рамках общей тенденции к стандартизации такие необоснованные расширения библиотек сокращаются, но в ранних системах программирования разнобой был крайне высок.

К сожалению, наборы функций второй категории не согласованы даже для различных систем программирования в рамках одной операционной системы на одном типе архитектур компьютеров. Четко прослеживается это и на примере систем программирования TC и MSC. Библиотечные функции, обеспечивающие интерфейс для вызова одной и той же функции операционной системы, могут иметь не только разные параметры, но и разные названия.

Эти несогласованности объясняются, с одной стороны, коммерческими соображениями — стремлением удержать под контролем рынок программного обеспечения, чтобы пользователи, начавшие программировать с использованием одной системы программирования, покупали затем более новые программные продукты той же фирмы, а с другой стороны, поздним появлением стандарта на язык и на его библиотеку и независимости эволюции от версии к версии каждой системы программирования. При этом, надо отметить, происходит постепенное сближение различных систем программирования по мере того, как каждая из них заимствует наиболее ценные идеи у конкурентов. Так, различия между библиотеками более поздних версий систем программирования MSC и TC отчасти сокращены по сравнению с первыми версиями.

Данную часть книги следует рассматривать в первую очередь как справочник по стандартной библиотеке языка Си двух систем программирования — MSC и TC — для компьютеров типа IBM PC. Она также будет полезна для разработчиков новых систем программирования Си, поскольку в ней проводится сравнение реализации различных библиотечных функций двух широко распространенных систем программирования.

Из-за ограничений по объему в книгу не вошло описание специальных графических библиотек Си систем программирования MSC и TC.

Структура описания библиотеки такова: сначала дается краткое описание различных групп функций и вводятся основные понятия, используемые далее при описании библиотечных функций. Затем приводится полное описание всех функций в алфавитном порядке.

Предпринята попытка дать детальное описание библиотечных функций, приводится описание используемых констант, как они описываются (с помощью директивы препроцессора #define) во включаемых файлах.

Авторы считают, что для эффективной разработки качественного программного обеспечения на языке Си программист должен представлять нижний уровень реализации языка и его стандартной библиотеки. Поэтому иногда дается описание деталей (в частности, связанных с использованием системных вызовов ОС и имен типов, введенных с помощью конструкции typedef), которые часто сознательно замалчиваются в документации по библиотекам.

По нашему мнению, пользователь должен обладать полной информацией, осознавая при этом, с какими проблемами он столкнется при переносе программ в новую версию системы программирования, в новую операционную систему или на компьютер другой архитектуры.

Обозначение ANSI, используемое в полном описании библиотеки, указывает, что отмеченная библиотечная функция включена в стандарт языка Си.

9 КРАТКОЕ ОПИСАНИЕ БИБЛИОТЕКИ

Ниже приводится краткое описание основных групп функций для быстрой ориентации в библиотеке. При этом вводятся основные понятия, используемые при описании библиотечных функций (в частности, связанные с организацией ввода/вывода). Также указывается, в каком стандартном включаемом (по директиве препроцессора #include) файле содержится описание прототипа функции, относящихся к ней структур данных и констант.

9.1 Работа с областями памяти и строками

В стандартной библиотеке есть специальная группа Функций для обработки областей памяти, которые рассматриваются как последовательности байтов.

Если размер области, с которой необходимо работать, задается явно, будем называть такую область буфером.

Другое используемое понятие — строка. Отличие строки от буфера в том, что ее размер задается не явно, а определяется первым встретившимся при просмотре строки слева направо нулевым байтом (имеющим значение '\0'), причем считается, что этот нулевой байт также принадлежит строке.

Для копирования буферов, для присваивания каждому байту в пределах указанного буфера заданного значения и для сравнения содержимого двух буферов предназначены следующие функции:

Функция	Краткое описание
memccpy	копирует символы из одного буфера в другой до тех пор, пока не будет скопирован заданный символ или не будет скопировано определенное число символов
memchr	возвращает указатель на первое вхождение заданного символа в буфере
memcmp	сравнивает указанное число символов из двух буферов
memlcmp	сравнивает указанное число символов двух буферов, считая строчные и прописные буквы эквивалентными
memcpy	копирует указанное количество символов из одного буфера в другой
memset	инициализирует заданным значением указанное количество байтов в буфере
movedata	копирует определенное количество символов из одного буфера в другой, даже когда буфера находятся в разных сегментах

Прототипы перечисленных функций содержатся в файле memory.h (MSC) и в файлах mem.h и string.h (TC).

Система программирования TC предоставляет дополнительно следующие функции для работы с буферами:

Функция	Краткое описание
memove	копирует указанное количество символов из одного буфера в другой
movmem	копирует указанное количество символов из одного буфера в другой
setmem	инициализирует заданным значением указанное количество байтов в буфере

Прототипы функций memmove и movmem содержатся в файлах mem.h и string.h. Прототип функции setmem содержится в файле mem.h.

Для работы со строками существуют следующие библиотечные функции (TC & MSC):

Функция	Краткое описание
strcat	катенация (склеивание) строк
strchr	найти первое вхождение заданного символа в строке
strcmp	сравнить две строки
strcpy	копировать одну строку в другую
strcspn	найти первое вхождение символа из заданного набора символов в строке
strdup	дублирование строки
strerror	сформировать в строке сообщение об ошибке, состоящее из двух частей: системной диагностики и необязательного добавочного пользовательского сообщения
stricmp	сравнить две строки, считая символы нижнего и верхнего регистров эквивалентными
strlen	вычислить длину строки

strlwr	преобразовать строку в нижний регистр (строчные буквы)
strncat	добавить n символов в строку
strncmp	сравнение n символов в двух строках
strncpy	скопировать n символов из одной строки в другую
strnicmp	сравнение n символов двух строк
strnset	установить n символов в строке в заданное значение
strpbrk	найти первое вхождение любого символа из заданного набора в строке
strrchr	найти последнее вхождение заданного символа в строке
strrev	инвертировать (перевернуть) строку
strset	установить все символы строки в заданное значение
strspn	найти первую подстроку из заданного набора символов в строке
strstr	найти первую подстановку одной строки (более короткой) в другой
strtok	найти следующую точку в строке
strupr	преобразовать строку в верхний регистр (заглавные буквы)

Кроме того, система программирования MSC предоставляет дополнительно функцию strcmpi (идентична функции stricmp), а система программирования TC предоставляет функцию stpcpy (идентична функции strcpy, но возвращает в точку вызова другое значение).

Прототипы всех функций работы со строками содержатся в файле string.h. Все функции работают со строками, завершающимися нулевым байтом ('\0'). Для работы с массивом символов, не имеющим в конце нулевого байта, вы можете использовать функции преобразования буферов, описанные выше.

9.2 Определение класса символов и преобразование символов

Функция	Краткое описание
isalnum	проверка на букву или цифру
isalpha	проверка на букву
isascii	проверка на символ из набора кодировки ASCII
iscntrl	проверка на управляющий символ
isdigit	проверка на десятичную цифру
isgraph	проверка на печатный символ, исключая пробел
islower	проверка на малую букву
isprint	проверка на печатный символ
ispunct	проверка на знак пунктуации
isspace	проверка на пробельный символ
isupper	проверка на заглавную букву
isxdigit	проверка на шестнадцатеричную цифру
toascii	преобразование символа в код ASCII
tolower	проверка и преобразование в малую букву, если заглавная буква
toupper	проверка и преобразование малой буквы в заглавную
_tolower	преобразование буквы в малую (без проверки)
_toupper	преобразование буквы в заглавную (без проверки)

Все эти функции реализованы как макроопределения, заданные в файле ctype.h

9.3 Форматные преобразования данных

Функция	Краткое описание
atof	преобразование строки, в представляемое ей число типа float
atoi	преобразование строки в число типа int (целое)
atol	преобразование строки в число типа long (длинное целое)
ecvt	преобразование числа типа double в строку
fcvt	преобразование числа типа double в строку
gcvt	преобразование числа типа double в строку
itoa	преобразование числа типа int в строку
ltoa	преобразование числа типа long в строку
ultoa	преобразование числа типа unsigned long в строку

Система программирования TC предоставляет также следующие функции:

Функция	Краткое описание
strtod	преобразование строки в число типа double (покрывает возможности atof)
strtol	преобразование строки в число типа long (покрывает возможности atol)
strtoul	преобразование строки в число типа unsigned long

Прототипы всех перечисленных функций содержатся в файле stdlib.h. Прототип функции atof содержится, кроме того, в файле math.h.

9.4 Работа с каталогами файловой системы

Функция	Краткое описание
chdir	изменение текущего рабочего каталога
getcwd	получить имя текущего рабочего каталога
mkdir	создать новый каталог
rmdir	удаление каталога

Система программирования TC предоставляет, кроме перечисленных, следующие функции:

Функция	Краткое описание
findfirst	начало поиска файла по шаблону имени
findnext	продолжение поиска файла по шаблону имени
fnmerge	создание имени файла из отдельных компонент
fnsplit	разбиение имени файла на отдельные компоненты
getcurdir	узнать текущий каталог
getdisk	узнать текущее устройство
searchpath	поиск файла в различных каталогах
setdisk	задать текущее устройство

В системе программирования MSC прототипы функций содержатся в файле direct.h, в системе программирования TC прототипы функций содержатся в файле dir.h.

9.5 Операции над файлами

Функция	Краткое описание
access	определение прав доступа к файлу
chmod	изменение прав доступа к файлу
filelength	измерение длины файла
isatty	проверка, является ли устройство символьным
mktemp	генерация уникального имени файла
remove	уничтожение файла
rename	переименование файла
setmode	установить новые значения для параметров файла

Система программирования MSC предоставляет дополнительно следующие функции:

Функция	Краткое описание
chsize	изменение размера файла
fstat	получение информации о файле
locking	запирает область в файле (работает с версией ОС MS-DOS 3.0 и выше), временно запрещая к ней доступ со стороны других процессов, или отпирает эту область
slat	получение информации о файле
umask	установка маски для выбора режима работы по умолчанию
unlink	удаление файла

Система программирования TC предоставляет дополнительно функции:

Функция	Краткое описание
chmod	изменение прав доступа к файлу
lock	запирает область файла для доступа со стороны других процессов (работает с версией MS-DOS 3.0 и выше)
unlock	отпирает область (работает с версией MS-DOS 3.0 и выше)

Прототипы функций, за исключением функций **fstat** и **stat**, содержатся в файле **io.h**; прототипы функций **fstat** и **stat** описаны в файле **sys\stat.h**.

Функции **access**, **chmod**, **rename**, **stat** и **unlink** оперируют с файлами, которые задаются через имя пути (**pathname**) -или через имя файла.

Функции **chsize**, **filelength**, **isatty**, **locking**, **setmode**, **fstat** работают с уже открытыми файлами, которые определяются дескрипторами (**handle**) (смотри далее описание функций ввода/вывода нижнего уровня).

9.6 Ввод и вывод

Функции ввода и вывода в стандартной библиотеке Си позволяют читать данные из файлов или получать их с устройств ввода (например, с клавиатуры) и записывать данные в файлы, или выводить их на различные устройства (например, на принтер).

Функции ввода/вывода делятся на три класса:

1) Ввод/вывод верхнего уровня (с использованием понятия "поток").
2) Ввод/вывод для консольного терминала путем непосредственного обращения к нему.
3) Ввод/вывод нижнего уровня (с использованием понятия "дескриптор").

В библиотеке есть также функции для работы с последовательным портом (COM), они отнесены условно ко второй группе.

функции ввода/вывода верхнего уровня обеспечивают буферизацию работы с файлами. Это означает, что, когда производится чтение информации из файла или запись информации в файл, обмен информацией осуществляется не между программой и указанным файлом, а между программой и промежуточным буфером, расположенным в оперативной памяти.

Если производится операция записи в файл, то информация из буфера записывается в файл при заполнении буфера или при закрытии файла (или при выполнении каких-то других условий, смотри ниже). Если информация считывается из файла, то она на самом деле берется из буфера, а в буфер информация считывается из файла при открытии файла и впоследствии каждый раз при исчерпании (опустошении) буфера. Буферизация ввода/вывода выполняется автоматически, она позволяет ускорить выполнение программы за счет уменьшения количества обращений к сравнительно медленно работающим внешним устройствам.

Для пользователя файл, открытый на верхнем уровне, представляется как последовательность считываемых или записываемых байтов. Чтобы отразить эту особенность организации ввода/вывода, предложено понятие "поток" (соответствует английскому слову **stream**). Когда файл открывается, с ним связывается поток, выводимая информация записывается "в поток", считываемая информация берется "из потока".

Когда поток открывается для ввода/вывода, он связывается со структурой типа FILE (имя типа FILE определяется с помощью конструкции **typedef** в файле **stdio.h**). Структура содержит разнообразную информацию о файле. При открытии файла с помощью функции **fopen** возвращается указатель на структуру типа FILE. Этот указатель (указатель потока) используется для последующих операций с файлом, пользователь не обязан вникать в способ организации потока, он только должен сохранить полученный указатель и передавать его значение всем библиотечным функциям, используемым для ввода/вывода через этот поток.

Функции в/в верхнего уровня дают возможность для буферизованного форматированного и неформатированного ввода/вывода.

Функции в/в верхнего уровня относятся к числу функций, одинаково реализуемых в различных ОС и на разных компьютерах, с их помощью пользователь имеет возможность писать переносимые программы.

Функции ввода/вывода для консоли и порта распространяют возможности функций ввода/вывода верхнего уровня на этот класс устройств, добавляя новые возможности.

Они позволяют читать или записывать на консоль (терминал) или в порт ввода/вывода (например, порт принтера). Функции в/в с портом читают или записывают данные побайтно. Некоторые дополнительные режимы устанавливаются для в/в с консоли (например: ввод с эхо-печатью символов и без эхо-печати).

Функции в/в для консоли и порта являются уникальными для компьютеров типа IBM/PC.

Функции в/в низкого уровня не выполняют буферизацию и форматирование данных; они позволяют непосредственно пользоваться средствами ввода/вывода операционной системы.

При низкоуровневом открытии файла (при помощи функции **open**) с ним связывается дескриптор (**handle**). Дескриптор является целым значением, характеризующим размещение информации об открытом файле во внутренних таблицах системы. Дескриптор используется при последующих операциях с файлом.

Функции в/в нижнего уровня из стандартной библиотеки целесообразно использовать при разработке своей собственной подсистемы ввода/вывода.

Функции в/в нижнего уровня переносимы в рамках некоторых систем программирования Си, в частности относящихся к ОС UNIX.

9.6.1 Функции вода/вывода высокого уровня

Функция	Краткое описание
clearerr	очистка флажка ошибки для потока
fclose	закрытие потока
fcloseall	закрытие всех открытых (на верхнем уровне) файлов

fdopen	создание потока для файла, ранее открытого на нижнем уровне, используя дескриптор
feof	проверка на конец потока
ferror	проверка флажка ошибок потока
flush	сброс буфера потока на связанное с ним внешнее устройство
fgetc	чтение символа из потока
fileno	получение дескриптора файла, связанного с потоком
fgetchar	чтение символа из стандартного потока ввода stdin
fgets	чтение строки из потока
flushall	сброс буферов всех потоков
fopen	открытие потока (открыть файл и связать его с потоком)
fprint	запись данных в поток по формату
fputc	запись символа в поток
fputchar	запись символа в стандартный поток вывода stdout
fputs	запись строки в поток
fread	неформатированное чтение данных из потока
freopen	повторное открытие потока в новом режиме
fscanf	чтение из потока по формату
fseek	перемещение указателя файла в заданную позицию
ftell	получение текущей позиции указателя файла
fwrite	неформатированная запись данных в поток
getc	чтение символа из потока (реализуется через макроопределение)
getchar	чтение символа из потока stdin (версия макро)
gets	чтение строки из потока stdin
getw	чтение двух байтов (по размеру int) в формате слова из потока
printf	запись данных в поток stdout по формату
putc	запись символа в поток (версия макро)
putchar	запись символа в поток stdout (версия макро)
puts	запись строки в поток
putw	запись двух байтов (по размеру int) в формате слова в поток
rewind	установка указателя по файлу на начало файла
scanf	чтение данных из потока stdin по формату
setbuf	управление буферизацией потока
setvbuf	управление буферизацией потока и размером буфера
sprintf	запись данных в строку по формату
sscanf	чтение данных из строки по формату
tempnam	сгенерировать имя временного файла в заданном каталоге
tmpfile	создать временный файл
ungetc	вернуть символ в поток
vfprintf	запись данных в поток по формату
vsprintf	запись данных в строку по формату

Система программирования MSC дополнительно предоставляет следующие функции:

Функция	Краткое описание
rmtemp	удаление временных файлов, созданных посредством функции tmpfile
tmpnam	сгенерировать имя временного файла
vprintf	запись данных в поток stdout по формату

Система программирования TC дополнительно предоставляет следующие функции:

Функция	Краткое описание
vfscanf vscanf vsscanf	эти функции подобны функциям fscanf, scanf и sscanf, но принимают как параметр указатель на список аргументов — адресов переменных, которым присваиваются вводимые значения

Прототипы всех функций ввода/вывода верхнего уровня содержатся в файле stdio.h.

Некоторые константы, определенные в stdio.h, могут быть полезны в программе:

константа EOF	код, возвращаемый как признак конца файла
константа NULL	значение указателя, который не содержит адрес никакого реально размещенного в оперативной памяти объекта
константа BUFSIZ	определяет размер буфера потока в байтах
имя типа FILE	структура, которая содержит информацию о потоке

9.6.1.1 Высокоуровневое открытие файлов

Функции открытия потока возвращают указатель на тип FILE (этот указатель называют также указателем потока), этот указатель используется при дальнейших обращениях к потоку.

9.6.1.2 Стандартные потоки: stdin, stdout, stdeir, stdaux, stdprn.

Когда программа начинает выполняться, автоматически открываются пять потоков. Эти потоки — стандартный ввод (**stdin**), стандартный вывод (**stdout**), стандартный вывод для сообщений об ошибках (**stderr**), стандартный последовательный порт (**stdaux**) и стандартное устройство печати (**stdprn**).

По умолчанию стандартный ввод/вывод и стандартный вывод сообщений об ошибках связывается с консольным терминалом.

Назначения по умолчанию для стандартного порта и стандартного устройства печати зависят от конфигурации аппаратуры компьютера; эти потоки обычно связываются с последовательным портом и принтером, но могут быть и не установлены в отдельных системах.

Следующие указатели на структуру типа. FILE определяются в файле **stdio.h** и могут использоваться в любом месте как указатели потоков:

```
extern FILE * stdin; — стандартный ввод
extern FILE * stdout;— стандартный вывод
extern FILE * stderr;— стандартный вывод сообщений об ошибках
extern FILE * sidaux;— стандартный порт
extern FILE * stdprn;— стандартное устройство печати
```

При запуске оттранслированной программы на выполнение можно использовать символы перенаправления в/в из командного языка MS-DOS (< , > или >>) для переопределения стандартного ввода и вывода программы.

Можно переопределить **stdin**, **stdout**, **stderr**, **stdaux** или **stdprn** так, что они будут относиться к файлу на диске или устройству. Такие возможности предоставляет функция **freopen**.

9.6.1.3 Управление буферизацией потоков

Открытые файлы, для которых осуществляется высокоуровневый ввод/вывод, буферизуются по умолчанию, за исключением потоков **stdin**, **stdout**, **stderr**, **stdaux**, **stdprn**.

Потоки **stderr** и **stdaux** — не буферизованы. Если к ним применяется функция **printf** или **scanf**, создается временный буфер. Для обоих потоков может задаваться буферизация с помощью функций **setbuf** или **setvbuf**.

Буферизация для потоков **stdin**, **stdout**, **stdprn** выполняется следующим образом: буфер сбрасывается при его заполнении или когда вызванная библиотечная функция ввода/вывода завершает работу.

Использованием функции **setbuf** или **setvbuf** можно сделать поток небуферизованным или связать буфер с небуферизованным до этого потоком. Буфера, размещенные в системе, недоступны пользователю, кроме буферов, полученных с помощью **setbuf** или **setvbuf**.

Буфера должны иметь постоянный размер, равный константе BUFSIZ в **stdio.h**.

Если используется **setvbuf**, размер буфера устанавливает пользователь. Буфера автоматически сбрасываются при их наполнении, или когда связанный с буфером файл закрывается, или когда происходит нормальное завершение программы.

Можно сбросить буфера в произвольный момент времени, используя функции **fflush** и **flushall**. Функция **fflush** сбрасывает буфер одного заданного потока, а функция **flushall** сбрасывает буфера всех потоков, которые открыты и буферизованы в данный момент.

9.6.1.4 Закрытие потоков

Функции **fclose** и **fcloseall** закрывают поток или потоки. Функция **fclose** закрывает один заданный поток, **fcloseall** — все потоки, кроме потоков **stdin**, **stdout**, **stderr**, **stdaux**, **stdprn**.

Если программа не выполняет закрытия потоков, потоки автоматически закрываются, когда программа завершается неаварийно. Однако следует закрывать потоки по завершении работы с ними, так как число потоков, которые могут быть открыты одновременно, ограничено.

9.6.1.5 Чтение и запись данных

Функции ввода/вывода верхнего уровня позволяют передавать данные различными способами.

Операции чтения и записи в потоках начинаются с текущей. позиции в потоке, идентифицируемой как "file pointer"

(указатель файла) для потока. Указатель файла изменяется после выполнения операции чтения или записи.

Например, если Вы читаете один символ из потока, указатель файла продвигается на 1 байт, так что следующая операция начнется с первого несчитанного символа. Если поток открыт для добавления, указатель файла автоматически позиционируется на конец файла перед каждой операцией записи.

Поток, связанный с таким устройством, как консольный терминал, не имеет указателя файла. Программы, которые перемещают указатель файла (с помощью функции **fseek**), будут иметь в этом случае неопределенный результат.

9.6.1.6 Обнаружение ошибок

Когда происходит ошибка в операции с потоком, устанавливается в ненулевое значение флажок ошибки для потока. Можно использовать макроопределение **ferror**, чтобы определить, произошла ли ошибка.

После каждой ошибки флажок ошибки остается установленным до тех пор, пока не будет сброшен вызовом функции **clearerr** или **rewind**.

9.6.2 Функции вода/вывода нижнего уровня

Функция	Краткое описание
close	закрыть файл
creat	создать файл
dup	создать второй дескриптор (handle) для файла
dup2	переназначить дескриптор (handle) для файла
eof	проверка на конец файла
lseek	позиционирование указателя файла в заданное место
open	открыть файл
read	читать данные из файла
sopen	открыть файл в режиме разделения
tell	получить текущую позицию указателя файла
write	записать данные в файл

Система программирования TC предоставляет дополнительно следующие функции:

Функция	Краткое описание
_creat	создать файл
creatnew	создать новый файл
creattemp	создать новый файл
_open	открыть файл
_read	чтение данных из файла
_write	запись данных в файл

Нижний уровень ввода и вывода не работает с буферизованными или форматированными данными. Для работы с файлами, открытыми посредством функции нижнего уровня, используется дескриптор файла (**handle**).

Для открытия файлов используются функции **open** и **_open**; В ОС MS/DOC версии 3.0 или выше может быть использована функция **sopen** для открытия файлов с атрибутами режима разделения файла.

функции нижнего уровня, в отличие от функций верхнего уровня, не требуют включения файла **stdio.h**. Тем не менее нескольких общих констант, определенных в файле **stdio.h**, как, например, признак конца файла EOF, могут оказаться полезными. Если программа использует эти константы, необходимо включить файл **stdio.h**.

Прототипы функций нижнего уровня содержатся в файле **io.h**.

9.6.2.1 Открытие файлов

Файл должен быть открыт функциями **open**, **sopen** или **creat** до выполнения первой операции ввода или вывода с использованием функций нижнего уровня для этого файла.

Файл может быть открыт для чтения, записи, или для чтения и записи, может быть открыт в текстовом или в двоичном режиме.

Файл **fcntl.h** должен быть включен при открытии файла, так как содержит определения для флагов, используемых в функции **open**. В некоторых случаях также должны быть включены файлы **sys\types.h** и **sys\stat.h**.

Перечисленные функции возвращают дескриптор файла, который используется при последующих операциях с файлом. При вызове одной из функций открытия файла необходимо

возвращаемое функцией значение присвоить целочисленной переменной и потом использовать значение этой переменной, чтобы обращаться к открытому файлу.

9.6.2.29.6.2.2. Переопределение дескрипторов (handle)

Когда программа начинает выполняться, пять дескрипторов (**handle**), соответствующих стандартным вводу, выводу, выводу сообщений об ошибках, порту и устройству печати, уже назначены. Пользователь может использовать значения этих дескрипторов при вызове функций ввода/вывода нижнего уровня.

Каждый из этих дескрипторов соответствует одному из стандартных потоков, значения этих дескрипторов таковы:

поток	значение дескриптора
stdin	0
stdout	1
stderr	2
stdaux	3
stdprn	4

Можно использовать эти дескрипторы файлов в программе без предварительного открытия этих файлов. Они автоматически открываются при запуске программы.

Так же, как с функциями для потоков, Вы можете использовать перенаправление, чтобы переопределить стандартный ввод и вывод.

Функции **dup** и **dup2** позволяют назначать несколько **handle** для одного файла; эти функции обычно используются, чтобы связать дополнительные дескрипторы с уже используемыми файлами.

9.6.2.3Чтение и запись данных

Функции **read** и **write**, как и функции ввода/вывода верхнего уровня, начинают выполнение очередной операции с текущей позиции в файле. Текущая позиция изменяется при каждой операции чтения или записи.

Функция **eof** может быть использована для проверки на конец файла.

Когда происходит ошибка, программы в/в нижнего уровня присваивают код ошибки переменной **errno**. Можно использовать функцию **perror** для печати информации об ошибках в/в. Можно позиционировать указатель файла на определенную позицию в файле, используя функцию **lseek**. Используя функцию **tell**, можно определить текущую позицию указателя файла. Устройства типа консольного терминала не имеют указателя файла. Результат функций **lseek** и **tell** не определен, если они применяются к дескриптору, связанному с таким устройством.

9.6.2.4Закрытие файлов

Функция **close** закрывает открытые файлы. Открытые файлы также автоматически закрываются при неаварийном завершении программы.

9.6.3 Функции вода/вывода с консольного терминала и порта

Функции ввода/вывода для консольного терминала выделены в отдельную группу, потому что они используют специфические особенности компьютера IBM/PC (наличие специального видеоадаптера) и не являются переносимыми на другие типы компьютеров.

функция	Краткое описание
cgets	чтение строки с консоли
cprintf	запись данных на консольный терминал по формату
cputs	вывод строки на консольный терминал
getch	чтение символа с консоли
getche	чтение символа с консоли с эхо-печатью
kbhit	проверка нажатия клавиши на консоли
putch	вывод символа на консольный терминал
ungetch	возврат последнего прочитанного символа с консольного символа обратно с тем, чтобы он стал следующим символом для чтения

Система программирования MSC предоставляет дополнительно функцию cscanf - чтение данных с консоли по формату.

Система программирования TC предоставляет дополнительно функцию getpass - ввод с терминала пароля без эхо-печати

Прототипы функций содержатся в файле **conio.h**. Устройства: консольный терминал и порт не могут быть открыты или закрыты перед выполнением в/в, поэтому функции **fopen** и **fclose** не вызываются. Функции в/в с консольного терминала позволяют читать и записывать строки (**cgets** и **cputs**), форматированные данные (**cscanf** и **cprintf**) и

символы. Функция **kbhit** определяет: было ли нажатие клавиши на консольном терминале. Эта функция позволяет определить наличие символов для ввода с клавиатуры до попытки чтения.

9.7 Математические функции

Функция	Краткое описание
abs	нахождение абсолютного значения выражения типа int
acos	вычисление арккосинуса
asin	вычисление арксинуса
atan	вычисление арктангенса x
atan2	вычисление арктангенса от y/x
cabs	нахождение абсолютного значения комплексного числа
ceil	нахождение наименьшего целого, большего или равного x
_clear87	получение значения и инициализация слова состояния сопроцессора и библиотеки арифметики с плавающей точкой
_control87	получение старого значения слова состояния для функций арифметики с плавающей точкой и установка нового состояния
cos	вычисление косинуса
cosh	вычисление гиперболического косинуса
exp	вычисление экспоненты
fabs	нахождение абсолютного значения типа double
floor	нахождение наибольшего целого, меньшего или равного x
fmod	нахождение остатка от деления x/y
_fpreset	повторная инициализация пакета плавающей арифметики
frexp	разложение x как произведения мантиссы на экспоненту 2^n
hypot	вычисление гипотенузы
labs	нахождение абсолютного значения типа long
ldexp	вычисление $x*2^{exp}$
log	вычисление натурального логарифма
log10	вычисление логарифма по основанию 10
matherr	управление реакцией на ошибки при выполнении функций математической библиотеки
modf	разложение x на дробную и целую часть
pow	вычисление x в степени y
sin	вычисление синуса
sinh	вычисление гиперболического синуса
sqrt	нахождение квадратного корня
_status87	получение значения слова состояния с плавающей точкой
tan	вычисление тангенса
tanh	вычисление гиперболического тангенса

Система программирования MSC предоставляет дополнительно функции:

Функция	Краткое описание
bessel	вычисление функции Бесселя
dieeetomsbin	преобразование плавающего числа двойной точности из IEEE-формата в Microsoft-формат
dmsbintoieee	преобразование плавающего числа двойной точности из Microsoft-формата в IEEE-формат
fieeetomsbin	преобразование числа с плавающей точкой из IEEE-формата в Microsoft-формат
fmsbintoieee	преобразование числа с плавающей точкой из Microsoft-формата в IEEE-формат

Система программирования TC предоставляет дополнительно функции:

Функция	Краткое описание
_matherr	управление реакцией на ошибки при выполнении функций из математической библиотеки
pow10	вычисление десятичной степени

Прототипы функций содержатся в файле **math.h**, за исключением прототипов функций **_clear87**, **_control87**, **_fpreset**, **status87**, которые определены в файле **float.h**. Функция **matherr** (ее пользователь может задать сам в своей программе) вызывается любой библиотечной математической функцией при возникновении ошибки. Эта программа определена в библиотеке, но может быть переопределена пользователем, если она необходима, для установки различных процедур обработки ошибок.

9.8 Динамическое распределение памяти

Библиотека языка Си предоставляет механизм распределения динамической памяти (**heap**). Этот механизм позволяет динамически (по мере возникновения необходимости) запрашивать из программы дополнительные области оперативной памяти.

Работа функций динамического распределения памяти различается для различных моделей памяти, поддерживаемых системой программирования (смотри первую части книги).

В малых моделях памяти (**tiny**, **small**, **medium**) доступно для использования все пространство между концом сегмента статических данных программы и вершиной программного стека, за исключением 256-байтной буферной зоны непосредственно около вершины стека.

В больших моделях памяти (**compact**, **large**, **huge**) все пространство между стеком программы и верхней границей физической памяти доступно для динамического размещения памяти.

Следующие функции используются для динамического распределения памяти:

Функция	Краткое описание
calloc	выделить память для массива
free	освободить блок, полученный посредством функции calloc, malloc или realloc
malloc	выделить блок памяти
realloc	переразместить ранее выделенный блок памяти, изменив его размер
sbrk	переустановить адрес первого байта оперативной памяти, недоступного программе (начала области памяти вне досягаемости программы)

Система программирования MSC предоставляет дополнительно функции:

Функция	Краткое описание
alloca	выделение блока памяти из программного стека
_expand	изменение размера блока памяти, не меняя местоположения блока
_ffree	освобождение блока, выделенного посредством функции fmalloc
_fmalloc	выделение блока памяти вне данного сегмента
_freect	определить примерное число областей заданного размера, которые можно выделить
_fmsize	возвращает размер блока памяти, на который указывает дальний (far) указатель
halloc	выделить память для большого массива (объемом более 64 Кбайтов)
hfree	освободить блок памяти, выделенный посредством функции halloc
_memavl	определить примерный размер в байтах памяти, доступной для выделения
_msize	определить размер блока, выделенного посредством функций calloc, malloc, realloc
_nfree	освобождает блок, выделенный посредством _nmalloc
_nmalloc	выделить блок памяти в заданном сегменте
_nmsize	определить размер блока, на которой указывает близкий (near) указатель
stackavail	определить объем памяти, доступной для выделения посредством функции alloca

Система программирования TC предоставляет дополнительно функции:

Функция	Краткое описание
brk	переустановить адрес первого байта оперативной памяти, недоступного программе (начала области памяти вне досягаемости программы)
allocmem	низкоуровневая функция выделения памяти
freemem	низкоуровневая функция возврата памяти операционной системе
coreleft	узнать, сколько осталось памяти для выделения в данном сегменте
farcalloc	выделить блок памяти вне данного сегмента
farcoreleft	определить, сколько памяти для размещения осталось вне данного сегмента
farmalloc	выделить блок памяти вне данного сегмента
farrealloc	изменить размер блока, ранее выделенного функцией farmalloc или farcalloc
farfree	освободить блок, ранее выделенный функцией farmalloc или farcalloc

Прототипы функций содержатся в файле **malloc.h** для системы программирования MSC и в файле **alloc.h** для системы программирования TC.

Функции **calloc** и **malloc** выделяют блоки памяти, функция **malloc** выделяет заданное число байтов, тогда как **calloc** выделяет и инициализирует нулями массив элементов заданного размера.

Функции **_fmalloc** и **_nmalloc** подобны **malloc**, за исключением того, что **_fmalloc** и **_nmalloc** позволяют выделить блок байтов в том случае, когда существуют ограничения

адресного пространства текущей модели памяти. Функция **halloc** выполняется аналогично **calloc**, но **halloc** выделяет память для большого массива (больше 64 К).

Функции **realloc** и **_expand** изменяют размер полученного блока.

Функция **free** (для **calloc**, **malloc** и **realloc**), функция **ffree** (для **_fmalloc**), функция **_nfree** (для **_nmalloc**) и функция **hfree** (для **halloc**) освобождают память, которая была выделена ранее, и делают ее доступной для последующего распределения.

Функции **_freect** и **_memavl** определяют: сколько памяти доступно для динамического выделения в заданном сегменте; **_freect** возвращает примерное число областей заданного размера, которые могут быть выделены; **_memavl** возвращает общее число байтов, доступных для выделения.

Функции **_msize** (для **calloc**, **malloc**, **realloc** и **_expand**), **_fmsize** (для **_fmalloc**) и **_nmsize** (для **_nmalloc**) возвращают размер ранее выделенного блока памяти.

Функция **sbrk** — это функция нижнего уровня для получения памяти. Вообще говоря, программа, которая использует функцию **sbrk**, не должна использовать другие функции выделения памяти, хотя их использование не запрещено.

Все выше описанные функции распределяли области памяти из общей памяти. Система программирования MSC предоставляет 2 функции, **alloca** и **stackavail**, для выделения памяти из стека и определения количества доступной памяти в стеке.

9.9 Использование системных вызовов операционной системы MS-DOS

Функция	Краткое описание
bdos	вызов системы MS-DOS; используются только регистры DX и AL
dosexterr	получение значений регистров из системы MS-DOS вызовом 59H
FP_OFF	возвращает смещение far-указателя
FP_SEG	возвращает сегмент far-указателя
int86	вызов прерывания MS-DOS
int86x	вызов прерывания MS-DOS
intdos	системный вызов MS-DOS
intdosx	системный вызов MS-DOS
segread	возвращает текущее значение сегментных регистров

Прототипы функций и макроопределения содержатся в файле **dos.h**.

Система программирования MSC предоставляет дополнительно функции:

Функция	Краткое описание
inp	чтение с указанного порта в/в
outp	вывод в указанный порт в/в

Прототипы функций **inp** и **outp** содержатся в файле **conio.h**.

Система программирования TC предоставляет дополнительно следующие функции:

Функция	Краткое описание
absread	чтение с диска по номеру сектора
abswrite	запись на диск по номеру сектора
bdosptr	вызов системы MS-DOS
country	определение способа записи времени в данной стране
ctrlbrk	установить реакцию на <CTRL/BREAK>
disable	отменить прерывания
enable	разрешить прерывания
freemem	освободить память
getinterrupt	возбудить прерывание
getcbrk	узнать установленную реакцию на <CTRL/BREAK>
getdfree	узнать объем свободного места на диске
getdta	узнать адрес области передачи данных диска
getfat	получить информацию из таблицы размещения файлов
getfatd	получить информацию из таблицы размещения файлов
getpsp	получить сегментный префикс для текущего программного адреса текущего выполняемого процесса
getvect	узнать значение вектора прерывания
getverify	узнать режим проверки записи на диск
harderr	регистрация функции обработки аппаратных ошибок
hardresume	возврат из функции обработки аппаратных ошибок
hardretn	возврат из функции обработки аппаратных ошибок
inport	ввести слово из порта
inportb	ввести байт из порта
intr	аналог функции int86
keep	зафиксировать программу в памяти

MK_FP	составить far-указатель из компонент
outport	вывести слово в порт
outportb	вывести байт в порт
parsfnm	выделение имени файла из командной строки MS-DOS
peek	получить значение слова по адресу
peekb	получить значение байта по адресу
poke	записать слово в память по адресу
pokeb	записать байт в память по адресу
randbrd	чтение с диска
randbwr	запись на диск
setdta	установить адрес области передачи данных диска
setvect	задать значение вектора прерывания
setverify	включить режим проверки записи на диск
sleep	задержка
unlink	удаление файла

Прототипы функций и макроопределения содержатся в файле **dos.h.**

Система программирования TC предоставляет также следующие функции для обращения к BIOS (базовой подсистеме ввода/вывода операционной системы):

Функция	Краткое описание
bioscom	управление последовательным каналом
biosdisk	управление диском
biosequip	узнать конфигурацию аппаратуры
bioskey	управление клавиатурой
biosmemory	узнать объем оперативной памяти
biosprint	управление устройством печати
biostime	управление BIOS-таймером

Прототипы функций обращения к BIOS содержатся в файле **bios.h.**

9.10 Управление процессами

Функция	Краткое описание
abort	завершить процесс
execl	выполнить порождаемый процесс со списком аргументов
execle	выполнить порождаемый процесс со списком аргументов и заданным окружением (контекстом имен командного языка операционной системы)
execlp	выполнить порождаемый процесс, используя переменную PATH и список аргументов
execlpe	выполнить порождаемый процесс, используя переменную PATH, заданное окружение и список аргументов
execv	выполнить порождаемый процесс с массивом аргументов
execve	выполнить порождаемый процесс с массивом аргументов и заданным окружением
execvp	выполнить порождаемый процесс, используя переменную PATH и массив аргументов
execvpe	выполнить порождаемый процесс, используя переменную PATH, заданное окружение и массив аргументов
exit	завершить процесс
_exit	завершить процесс без скидывания буферов
signal	управление сигналом прерывания
spawnl	выполнить порождаемый процесс со списком аргументов
spawnle	выполнить порождаемый процесс со списком аргументов и заданным окружением
spawnlp	выполнить порождаемый процесс, используя переменную PATH и список аргументов
spawnlpe	выполнить порождаемый процесс, используя переменную PATH, заданное окружение и список аргументов
spawnv	выполнить порождаемый процесс с массивом аргументов
spawnve	выполнить порождаемый процесс с массивом аргументов и задан ным окружением
spawnvp	выполнить порождаемый процесс, используя переменную PATH и массив аргументов
spawnvpe	выполнить порождаемый процесс, используя переменную PATH, заданное окружение и массив аргументов
system	выполнение команды MS-DOS

Система программирования MSC предоставляет дополнительно функции:

Функция	Краткое описание
getpid	получить номер процесса
onexit	выполнить функцию при завершении программы

Термин "процесс" относится к программе, которая выполняется под управлением операционной системы. Процесс состоит из кодов программы и данных, а также информации о состоянии процесса, такой, как число открытых файлов. Где бы ни выполнялась программа на уровне MS-DOS, запускается процесс. Можно запустить, остановить и управлять процессом из программы, используя функции управления процессом. Прототипы всех функций управления процессами объявлены в файле **process.h** (исключая функцию **signal**). Прототип функции **signal** содержится в файле **signal.h**. Функции управления процессом позволяют следующее:

1) Узнать уникальный номер процесса (**getpid**).
2) Завершить процесс (**abort**, **exit**, **_exit**).
3) Управлять сигналами прерывания (**signal**).
4) Начать новый процесс (разновидности **exec** и **spawn** функции, **system** функция).

Функции **abort** и **_exit** осуществляют немедленное завершение без скидывания буферов потоков, функция **exit** осуществляет выход после скидывания буферов потоков. Функция **system** вызывает на выполнение заданную команду MS-DOS. Функции **exec** и **spawn** создают новый процесс, называемый порождаемым процессом. Разница между функциями **exec** и **spawn** в том, что **spawn** способна возвращать управление из порождаемого процесса к его родителю. Оба, и родитель, и порождаемый процесс, размещаются в памяти (если не указан флаг P_OVERLAY).

В функции **exec** порождаемый процесс перекрывает порождающий процесс, так что возврат управления в родительский процесс невозможен (если не произошла ошибка во время попытки запуска на выполнение порождаемого процесса).

В таблице описывается способ формирования **exec** и **spawn**. Имя функции задается в первом поле. Второе поле определяет: используется ли переменная PATH для поиска файла для выполнения, который определяет порождаемый процесс.

Третье поле описывает метод передачи аргументов порождаемому процессу. Передача аргументов списком означает, что аргументы в порождаемый процесс передаются один за одним, в том порядке, как пользователь перечислил их в обращении к функции **exec** или **spawn**. Передача аргументов массивом означает, что аргументы помещаются в массив и указатель на массив передается порождаемому процессу. Передача списком обычно используется, когда число аргументов постоянно и известно заранее, а метод передачи аргументов массивом полезен, когда число аргументов должно быть определено во время работы. Последнее поле определяет: унаследует ли порождаемый процесс от родителя окружение, или оно будет изменено для него.

Таблица 9.1.

функция	Использование PATH переменной	Способ передачи аргументов	Окружение
execl spawnl	не использует PATH	список аргументов	наследует от родителя
execle spawnle	не использует PATH	список аргументов	указатель на таблицу окружения (последний аргумент)
execlp spawnlp	использует PATH	список аргументов	наследует от родителя
execlpe spawnlpe	использует PATH	список аргументов	указатель на таблицу окружения (последний аргумент)
execv spawnv	не использует PATH	массив аргументов	наследует от родителей
execve spawnve	не использует PATH	массив аргументов	указатель на таблицу окружения (последний аргумент)
execvp spawnvp	использует PATH	массив аргументов	наследует от родителя
execvpe spawnvpe	использует PATH	массив аргументов	указатель на таблицу окружения (последний аргумент)

9.11 Поиск и сортировка

Следующие библиотечные функции предназначены для поиска и сортировки в массиве:

Функция	Краткое описание
bsearch	выполняет двоичный поиск

lfind	выполняет линейный поиск для заданного значения
lsearch	выполняет линейный поиск для заданного значения, которое добавляется в массив, если не найдено
qsort	выполняет быструю сортировку

Прототипы функций содержатся в файле **search.h** в системе программирования MSC, в файле **stdlib.h** в системе программирования TC.

9.12 Функции работы со временем

Функция	Краткое описание
asctime	преобразование времени из структуры (внутренней формы) в символьную строку
ctime	преобразование времени из длинного целого (long int) в строку символов
gmtime	преобразование времени из целого (int) в структуру
localtime	преобразование времени из целочисленного (int) в структуру с локальной поправкой
tzset	установить переменную времени из переменной времени среды

Система программирования MSC предоставляет дополнительные функции:

Функция	Краткое описание
ftime	получить текущее время системы как структуру
time	получить текущее системное время как длинное целое (long int)
utime	установить время изменения файла

Система программирования TC предоставляет дополнительные функции:

Функция	Краткое описание
difftime	вычислить разность по времени
dostounix	преобразование времени из формате ОС MS-DOS в формат ОС UNIX
getdate	получить системную дату как структуру
getftime	получить системную дату
gettime	получить системное время как структуру
setdate	установить системную дату
setftime	установить системное время
settime	установить системное время
stime	установить системное время
unixtodos	преобразовать время из формата ОС UNIX в формат ОС MS-DOS

Функции **time** и **ftime** возвращают текущее время как число секунд, прошедших с 1 января 1970 Гринвичского Всемирного времени. Эта величина может быть преобразована, скорректирована и сохранена посредством функций **asctime**, **ctime**, **gmtime** и **localtime**.

Функция **utime** устанавливает время модификации для указанного файла, используя текущее время или значение времени, заданное в структуре.

Функция **ftime** требует включения двух файлов: **sys\types.h** и **sys\timeb.h**. Прототип функции **ftime** содержится в **sys\timeb.h**.

Функция **utime** также требует включения двух файлов: **sys\types.h** и **sys\utime.h**. Прототип функции **utime** содержится в файле **sys\utime.h**.

Прототипы функций **dostounix**, **getdate**, **gettime**, **setdate**, **settime**, **unixtodos** содержатся в файле **dos.h**.

Прототипы функций **getftime** и **setftime** определены в файле **io.h**.

Прототипы остальных функций работы со временем времени содержатся в файле **time.h**.

При использования функции **ftime** или **localtime**, чтобы сделать поправку для местного времени, необходимо определить переменную командного языка операционной системы TZ.

9.13 Функции работы со списком аргументов

Функция	Краткое описание
va_arg	выбрать аргумент из списка
va_end	переустановить указатель
va_start	установить указатель на начало списка аргументов

Эти макроопределения дают возможность получить доступ к аргументам функции, когда число аргументов переменно.

В системе программирования MSC для совместимости с ОС UNIX System V можно использовать включаемый файл **vararg.h**, для совместимости со стандартом ANSI на язык Си можно использовать включаемый **stdarg.h**. В этих файлах содержится две различных версии макроопределений.

В системе программирования TC доступна только версия **stdarg.h**.

9.14 Другие функции

Функция	Краткое описание
assert	проверка утверждения о состоянии переменных
getenv	получить значение переменной среды (окружения)
perror	напечатать сообщение об ошибке
putenv	изменить значение переменной среды
swab	поменять местами два смежных байта
rand	получить псевдо-случайное число
srand	инициализация датчика случайных чисел
setjmp	запоминание точки для многоуровневого возврата
longjmp	многоуровневый возврат из функции

Прототипы всех функций, исключая **assert**, **longjmp** и **setjmp**, описаны в **stdiib.h**.

Assert — это макроопределение из файла **assert.h**.

Прототипы функций **setjmp** и **longjmp** содержатся в файле **setjmp.h**.

Программы **getenv** и **putenv** предоставляют доступ к таблице среды процесса. Глобальная переменная **environ** также указывает на таблицу среды, но рекомендуется использование функций **getenv** и **putenv** для доступа и изменения установленной среды вместо обращения к таблице среды напрямую.

Функция **perror** печатает диагностическое сообщение о последней ошибке, произошедшей при вызове какой-либо библиотечной функции.

Функция **swab** обычно используется для преобразования данных в формат других компьютеров, где используется иной порядок следования байтов в слове в оперативной памяти.

10. ПОЛНОЕ ОПИСАНИЕ БИБЛИОТЕКИ

ABORT (MSC & TC)

#include <process.h> /*используется только для описания функции*/

#include <stdlib.h> /*используйте либо process.h, либо stdlib.h*/

void abort();

Описание (TC & MSC)

Функция **abort** выводит сообщение:

Abnormal program termination (ненормальное завершение программы) в файл *stderr*, затем прерывает программу посредством вызова функции **_exit()** с кодом завершения, равным 3.

Функция **abort** не сбрасывает буфера файлов. Управление возвращается процессу, который вызвал данный процесс (обычно операционной системе).

Описание (для MSC 5.1)

Функция **abort** выводит сообщение:

Abnormal program termination (ненормальное завершение программы) в файл *stderr*, затем производит вызов функции **raise**(SIGABRT). Действия, выполняемые при возбуждении сигнала SIGABRT, могут быть переопределены заранее с помощью функции **signal()**. По умолчанию действия для SIGABRT — завершение программы с кодом завершения, равным 3, и передача управления вызвавшему процессу.

Функция **abort** не сбрасывает буфера файлов, а также не вызывает обработки завершения функциями **atexit()** и **onexit()**. Управление возвращается процессу, который инициализировал вызванный процесс (обычно операционной системе).

Возвращаемое значение

Возврат управления из функции в точку вызова не происходит. Статус 3 возвращается процессу, из которого вызван данный процесс (операционной системе).

Смотри также

signal, raise, exit, _exit; функции группы exec; функции группы spawn

ABS (TC & MSC & ANSI)

#include <stdlib.h> /*(TC & MSC)*/

#include <math.h> /*(TC)*/

int abs(n);

int n;

Описание (TC)

Функция **abs** возвращает абсолютное значение целочисленного аргумента.

Функция **abs** определяется как макрорасширение в файле *stdlib.h*, если Вы все же хотите использовать **abs** как функцию и подключаете файл *stdlib.h*, то необходимо отменить это макроопределение:

#include <stdlib.h>

#undef abs

Описание (MSC)

Функция **abs** возвращает абсолютное значение целочисленного аргумента.

Возвращаемое значение

Функция возвращает абсолютное значение аргумента, значение в диапазоне от 0 до 32767. Исключение: **abs**(-32768)=-32768.

Смотри также

cabs, fabs, labs

ABSREAD (TC)

#include <dos.h>

*int absread(int drive, int nsects, int sectno, void *buffer);*

Описание

Функция считывает информацию из секторов на диске с учетом только формата диска. Игнорируется логическая структура диска и не обращается внимания на расположение файлов, FATы или каталоги.

Функция **absread** считывает сектора диска через прерывание 0x25.

Значение параметров следующее:

drive — номер, идентифицирующий устройство, с которого производится чтение (0=A, 1=B и т.д.);

nsects — количество считываемых секторов;

sectno — номер логического сектора, с которого начинается чтение;

buffer — адрес буфера памяти, в который читаются данные.

Количество считываемых секторов ограничено объемом памяти в сегменте, от адреса *buffer* до конца сегмента. Т.о., 64 Кбайта — максимальный объем информации, которая может быть считана при одном вызове функции **absread**.

Возвращаемое значение

При успешном завершении операции чтения функция возвращает значение 0. В случае ошибки возвращается значение -1, и переменная *errno* принимает значение регистра AX, возвращаемое после системного вызова (см. документацию по ОС MS-DOS).

Смотри также

abswrite, biosdisk

Переносимость

Только для MS-DOS.

ABSWRITE (TC)

#include <dos.h>

*int abswrite(int drive, int nsects, int sectno, void *buffer);*

Описание

Функция записывает информацию в сектора на диске, учитывая только формат диска. Игнорируется логическая структура диска и не обращается внимания на расположение файлов, FATы или каталога.

abswrite пишет сектора на диск, используя системное прерывание 0x26. Значение параметров следующее:

drive — номер, идентифицирующий устройство, на которое производится запись (0=A, 1=B и т.д.);

nsects — количество записываемых секторов;

sectno — номер логического сектора, с которого начинается запись;

buffer — адрес буфера памяти, откуда читаются данные для записи на диск.

Количество записываемых секторов ограничено объемом памяти в сегменте, от адреса *buffer* до конца сегмента. Т.о., 64 Кбайта — максимальный объем информации, которая может быть записана при одном вызове функции **abswrite**.

Возвращаемое значение

При успешном завершении операции записи функция возвращает значение 0. В случае ошибки возвращается значение -1, и переменная *errno* принимает значение регистра AX, возвращаемое после системного вызова (см. документацию по ОС MS-DOS).

Смотри также

absread, biosdisk

Переносимость

Только для MS-DOS.

ACCESS (TC & MSC)

#include <io.h> /*используется только для описания функции*/

int access(pathname, mode);

*char *pathname;*

int mode;

Описание

Функция устанавливает: существует ли указанный файл и разрешен ли к нему доступ в данном режиме *mode*. Возможные значения *mode*:

06 Проверить возможность чтения и записи;

04 Проверить возможность чтения;

02 Проверить возможность записи;

00 Проверить существование файла.

В MS DOS все существующие файлы доступны для чтения. Таким образом, значения параметра *mode* 00 и 04, а также 02 и 06 идентичны.

Возвращаемое значение

0 — доступ разрешен

1 — доступ запрещен или файл не существует, при этом переменная *errno* устанавливается в одно из следующих значений:

EACCES — запрещен доступ в требуемом режиме;

ENOENT — файл или путь к нему не найден.

Смотри также

chmod, fstat, open, stat

ACOS (TC & MSC & ANSI)

#include <math.h>

double acos(x);

double x;

Описание

Функция **acos** вычисляет значение арккосинуса от *x*. Значение аргумента должно быть в пределах от -1.0 до 1.0.

Возвращаемое значение

Значение арккосинуса от аргумента.

Возвращаемое значение равняется 0.0, если значение аргумента *x* не находится в пределах от -1.0 до 1.0; при этом переменная *errno* устанавливается в EDOM и печатается сообщение об ошибке DOMAIN в файл *stderr*.

Обработку ошибочной ситуации можно изменить, используя функцию **matherr**.

Смотри также

asin, atan, atan2, cos, matherr, sin, tan

ALLOCMEM (TC), _DOS_ALLOCMEM (MSC 5.1)

Использование (TC)

#include <dos.h>

*int allocmem(unsigned size, unsigned *seg);*

Использование (MSC 5.1)

#include <dos.h>

*unsigned dos_allocmem(unsigned size, unsigned *seg);*

Описание

Функция запрашивает сегмент памяти у операционной системы. Используется системный вызов 0x48 MS-DOS для выделения блока свободной памяти и возвращается адрес сегмента выделенного блока.

Значение параметров:

size — требуемый размер памяти в параграфах (один параграф равняется 16 байтам);

seg — указатель на переменную, куда будет записано значение адреса сегмента выделенного блока памяти.

(TC) — присвоения значения переменной, на которую указывает *seg*, не производится, если нет достаточного свободного места.

(MSC) — если запрос не может быть удовлетворен, то в **seg* записывается размер свободной памяти в параграфах.

Все выделяемые блоки выравниваются на границу параграфа.

Возвращаемое значение

В случае успешного завершения **allocmem** возвращает -1. В случае ошибки возвращается (число) (размер наибольшего свободного блока), а переменной *_doserrno* присваивается значение ENOMEM Недостаточно памяти.

Переносимость

Только для MS-DOS.

Смотри также

coreleft, freemem, malloc, setblock

ASCTIME (TC & MSC & ANSI)

#include <time.h>

*char *asctime(struct tm *ptm);*

Описание

Функция преобразует время из внутреннего представления, хранящегося в структуре (вид структуры описан в файле *time.h*), в строку символов. Строка-результат после выполнения **ctime** содержит 26 символов и имеет вид, показанный на следующем примере:

"Mon Dec 04 01:02:55 1989\n\0"

Каждое поле имеет символьный формат. Символ новой строки ('\n') и нулевой символ ('\0') занимают две последние позиции в строке.

Возвращаемое значение

Указатель на строку-результат.

Нет кодов ошибок.

Смотри также

ctime, ftime, gmtime, localtime, time

ASIN (TC & MSC & ANSI)

#include <math.h>

double asin(x);

double x;

Описание

Функция **asin** вычисляет значение арксинуса от *x*. Значение аргумента должно быть в пределах от -1.0 до 1.0.

Возвращаемое значение

Значение арксинуса от аргумента.

Возвращаемое значение равняется 0.0, если значение аргумента *x* не находится в пределах от -1.0 до 1.0; при этом переменная *errno* устанавливается в EDOM и печатается сообщение об ошибке DOMAIN в файл *stderr*.

Обработку ошибочной ситуации можно изменить, используя функцию **matherr**.

Смотри также

acos, atan, atan2, cos, matherr, sin, tan

ASSERT (TC & MSC & ANSI)

#include <assert.h>

void assert(expression);

Описание

Если значение выражения равно 0, то печатается диагностическое сообщение и завершается выполнение программы (для системы программирования MSC версии 5.1 в этом случае вызывается функция **abort()**).

Если значение выражения отлично от 0, то никаких действий не выполняется.

Диагностическое сообщение имеет следующую форму. Для системы программирования MSC версии 4.0 и системы программирования TC версии 1.5:

Asseption failed: file <*имя файла*>, line <*номер строки*>

Для системы программирования TC версии 2.0 и системы программирования MSC версии 5.1:

Asseption failed: <*выражение*>, file <*имя файла*>, line <*номер строки*>

Функция **assert** обычно используется, чтобы определить в программе логические ошибки (поддержка "защитного" программирования). Заданное выражение должно иметь значение, отличное от нуля, только в том случае, когда программа выполняется правильно (как задумано).

После того как программа отлажена, можно использовать идентификатор **NDEBUG**, чтобы не выполнять **assert** (не генерировать машинный код для проверок условия **assert**).

Если имя **NDEBUG** определено с помощью параметра **/D** в командной строке компилятора или с

помощью директивы препроцессора *#define*, то процессор не будет обрабатывать **assert** в исходной программе.

Если **NDEBUG** определяется в файле (т.е. используется второй вариант), то директива препроцессора

#define NDEBUG

должна стоять раньше, чем директива препроцессора

#include <assert.h>

Функция **assert** реализуется как макроопределение.

ATAN, ATAN2 (TC & MSC & ANSI)

#include <math.h>

double atan(x);

double(x);

double atan2(y, x); /*обратите внимание на порядок аргументов*/

double x;

double y;

Описание

Функции **atan** и **atan2** вычисляют арктангенс *x* и *y/x* соответственно: **atan** возвращает значение в пределах от *–pi/2* до *pi/2*; **atan2** возвращает значение в пределах от *-pi* до *pi*.

Возвращаемое значение

atan и **atan2** возвращают значение арктангенса.

Если оба аргумента функции **atan2** нулевые, то возвращается 0.0, при этом *errno* устанавливается в EDOM и печатается сообщение об ошибке DOMAIN в *stderr*.

Обработку ошибки можно изменить, используя функцию **matherr**.

Смотри также

acos, asm, cos, matherr, sin, tan

ATEXIT (TC & MSC 5.1 & ANSI)

Использование (TC)

#include <stdlib.h>

int atexit(func);

atextt_t func;

Использование (MSC 5.1)

#include <stdlib.h>

int atexit(func);

*void (*func) (void);*

Описание

Функция **atexit** регистрирует функцию, на которую указывает *func*, как функцию завершения. Эта функция будет вызвана в момент нормального (неаварийного) завершения программы.

В случае нормального окончания программы функция **exit** (вызываемая автоматически) вызывает функцию **func* (без аргументов) перед возвратом в операционную систему.

В системе программирования TC параметр имеет тип *atexit_t*, этот тип определен через *typedef* в файле *stdlib.h*. Каждый вызов функции **atexit** регистрирует новую завершающую функцию. Может быть зарегистрировано до 32 функций, которые будут выполнены в порядке "последний зарегистрирован — первый выполнен" (LIFO); это значит, что функция, зарегистрированная последней, выполнится первой.

Возвращаемое значение

В случае нормального окончания **atexit** возвращает 0, в случае ошибки (нет свободной памяти для регистрации функции) — ненулевое значение.

Смотри также

abort, onexit, exit, _exit, spawn...

ATOF, ATOI, ATOL (TC & MSC & ANSI)

#include <math.h> /*для функции atof*/

#include <stdlib.h> /*используйте math.h или stdlib.h*/

double atof(string);

*char *string;*

#include <stdlib.h> /*для функций atoi и atoll — только <stdlib.h>*/

int atoi(string);

long atol(string);

*char *string;*

Описание

Эти функции преобразуют строку символов в число с плавающей точкой двойной точности (функция **atof**), в целое (функция **atoi**) или в длинное целое (функция **atol**).

Строка символов является последовательностью символов, которая может быть интерпретирована как числовое значение определяемого типа. Функция заканчивает чтение символов из входной строки, как только встретит символ, который не может быть распознан как часть числа (этот символ может быть символом конца строки '\0').

Для функции **atof** строка должна иметь форму:

[пробелы][знак][цифры][.цифры][e[знак]цифры]

Первое поле может состоять из пробелов и табуляций, которые игнорируются. Знак — это символ "+" или "-".

Цифры — это одна или более десятичных цифр. Если нет цифр до десятичной точки, то они должны присутствовать после десятичной точки.

Знак *'e'* или *'E'* является признаком экспоненты.

Для функций **atoi** и **atol** строка должна иметь форму:

[пробелы][знак][цифры]

Возвращаемое значение

Функции возвращают: значение с плавающей точкой двойной точности (функция **atof**), целое число (функция **atoi**) или длинное целое число (функция **atol**).

Возвращается значение 0 (соответствующего типа), если не удалось преобразовать входную строку.

В случае переполнения **atof** возвращает плюс или минус значение HUGE_VAL, и функция **matherr** не переопределяет реакцию на такую ситуацию.

Смотри также

ecvt, fcvt, gcvt, scanf

BDOS, BDOSPTR (TC)

#include <dos.h>

int bdos(dosfun, dosdx, dosal);

int dosfun;

unsigned dosdx;

unsigned dosal;

int bdosptr(dosfun, argument, dosal);

int dosfun;

*void *argument;*

unsigned dosal;

Описание

Функции **bdos** и **bdosptr** обеспечивают непосредственный доступ ко многим системным вызовам ОС MS-DOS.

Доступ к системным вызовам, требующим параметра типа **int**, может быть осуществлен с помощью функции **bdos**. Доступ к системным вызовам, использующим параметр, являющийся указателем, осуществляется с помощью функции **bdosptr**.

Для малых моделей памяти (**tiny**, **small**, **medium**) эти две функции работают одинаково. Отличия проявляются при работе с большими моделями памяти, где существенно использование именно функции **bdosptr** для обращения к системному вызову, требующему указатель в виде параметра.

Параметр *dosfun* — это номер системного вызова.

Смотри справочное руководство по системе MS-DOS для получения информации о различных

системных вызовах.

В малых моделях памяти параметр *argument* определяет значение регистра DX; в больших моделях памяти этот параметр определяет значения регистров DS:DX.

Параметр *dosdx* задает значение регистра *dx*.

Параметр *dosal* задает значение регистра *al*.

Возвращаемое значение

Функция **bdos** возвращает содержимое регистра AX после выполнения системного вызова.

Функция **bdosptr** возвращает содержимое регистра AX после выполнения системного вызова в случае его успешного завершения. В случае ошибки возвращается значение -1 и устанавливается код ошибки в переменных *_doserrno* и *errno*.

Смотри также

intdos, intdosx, int86, int86x

BESSEL (MSC)

#include <math.h>

double j0(x);

double j1(x);

double jn(n, x);

double y0(x);

double y1(x);

double yn(n, x);

double x;

int n;

Описание

Функции *j0*, *j1* и *jn* вычисляют первую функцию Бесселя соответственно нулевого, первого и *n*-го порядка.

Функции *y0*, *y1* и *yn* вычисляют вторую функцию Бесселя соответственно нулевого, первого и *n*-го порядка.

Аргумент *x* должен быть положительным.

Возвращаемое значение

Эти функции возвращают значение функции Бесселя от аргумента *x*.

Для функций *j0*, *j1*, *y0*, *y1*, если *x* слишком велико, возвращается значение 0.0, в переменную *errno* засылается значение ERANGE и выводится сообщение об ошибке TLOSS в файл *stderr*.

Для *y0*, *y1* или *yn*, если значение *x* отрицательно, возвращается значение HUGE, переменной *errno* присваивается значение EDOM и выводится сообщение об ошибке DOMAIN в файл *stderr*.

Обработку ошибок можно изменить, используя функцию **matherr**.

Смотри также

matherr

BIOSCOM (TC), _BIOS_SERIALCOM (MSC 5.1)

Использование (TC)

#include <bios.h>

int bioscom(cmd, byte, port);

int cmd;

char byte;

int port;

Использование (MSC 5.1)

#include <bios.h>

unsigned _bios_serlalcom(cmd, port, byte);

unsigned cmd;

unsigned port;

unsigned byte;

/*обратите внимание на различный порядок следования параметров*/

Описание

Функция предназначена для работы с последовательным каналом связи (адаптером RS232). Используется прерывание BIOS номер 0x14.

Значение переменной *port*, равное 0, соответствует устройству COM1, равное 1 — соответствует COM2 и так далее. Всего в компьютере типа IBM PC/XT или /AT может быть до четырех последовательных каналов, но при загрузке автоматически проверяются и инициализируются только первые два из них. Регистры этих каналов начинаются с адреса 0000:0400H.

Значение параметра *cmd* может быть одним из следующих:

0 — установить параметры передачи, задаваемые параметром *byte*;

1 — передать символ, заданный параметром *byte*, в линию;

2 — принять символ из линии;

3 — получить текущий статус коммуникационного порта. Значение параметра *byte* представляет собой комбинацию следующих констант:

```
0x02     7-битный канал
0x03     8-битный канал
0x00     передача с одним стоп-битом
0x04     передача с двумя стоп-битами
0x00     Отсутствие проверки правильности передачи данных
```

```
0x08     Проверка по нечетности (odd parity)
0x18     Проверка по нечетности (even parity)
0x00     скорость передачи 110 бод (бит в секунду)
0x20     150 бод
0x40     300 бод
0x60     600 бод
0x80     1200 бод
0xA0     2400 бод
0xC0     4800 бод
0xC0     9600 бод
```

Возвращаемое значение

Для всех значений параметра *cmd* возвращаемое является 16-битным словом, где старшие 8 бит — это биты статуса (состояния), а младшие — возвращаемый байт.

Старшие биты могут принимать следующие значения:

```
15     Time out (нет ответа в установленный интервал времени);
14     Transmit shift register empty (конец передачи очередного байта);
13     Transmit holding register empty (разрешение загрузки очередного байта для передачи);
12     Break detect (обнаружено прерывание передачи);
11     Framing error (неверный формат принятой посылки);
10     Parity error (ошибка четности);
9      Overrun error (наложение данных);
8      Data ready (готовность данных).
```

Если значение *cmd* было равно 1 и бит 15 установлен в 1, значение *byte* не было передано по каналу.

Если значение *cmd* было равно 2, считанный байт записан в младших битах возвращаемого слова, если не было ошибок. Ошибкой считается, если хотя бы один бит старшего байта установлен в 1.

Если значение *cmd* было 0 или 3, старший байт может принимать значения в соответствии с тем, как это было описано выше, а биты младшего байта имеют следующее значение:

```
7     Received line signal detect (обнаружение несущей);
6     Ring indicator;
5     Data set ready (окончание всех операций);
4     Clear to send (разрешение передачи);
3     Delta receive line signal detector;
2     Trailing edge ring indicator;
1     Delta data set ready;
0     Delta clear to send.
```

BIOSDISK (TC), _BIOS_DISC (MSC 5.1)

Использование (TC)

#include <bios.h>

int biosdisk(cmd, drive, head, track, sector, nsect, buffer);

int cmd;

int drive;

int head;

int track;

int sector;

int nsect;

*void *buffer;*

Использование (MSC 5.1)

#include <bios.h>

int _bios_disk(cmd, diskinfo)

unsigned cmd;

struct diskinfo_t

{

unsigned drive;

unsigned head;

unsigned track;

unsigned sector;

unsigned nsectors;

*void far *buffer;*

*} *diskinfo;*

Описание

Эта функция позволяет обратиться к прерыванию BIOS номер 0x13 для работы с диском.

Параметр *drive* определяет номер диска. Значение 0 соответствует первому дисковому устройству типа накопителя на гибких магнитных дисках, 1 — второму и т.д. Для работы с постоянным жестким ("винчестерским") диском нумерация начинается с 0x80: 0x80 — первый, 0x81 — второй и так далее. Для жестких дисков задается физический номер устройства, а не номер раздела на диске. Прикладные программы сами должны интерпретировать информацию из таблицы разделов (*partition*), если им это необходимо.

Параметр *cmd* указывает необходимую операцию. Он может принимать следующие значения:

Для IBM PC, XT или AT, PS/2:

0 Сброс. Все остальные параметры игнорируются.

1 Возвращает статус последней дисковой операции. Все остальные параметры игнорируются.

2 Читает один или более секторов в память. Начальный сектор задается параметрами: *head*, *track* и *sector*. Число секторов задается параметром *nsect*. Данные читаются по 512 байтов в секторе в *buffer*.

3 Запись одного или более секторов из памяти. Начальный сектор задается параметрами: *head*, *track* и *sector*. Число секторов задается параметром *nsect*. Данные записываются по 512 байтов в каждый сектор из *buffer*.

4 Проверка одного или более секторов. Начальный сектор задается параметрами: *head*, *track* и *sector*. Число секторов задается параметром *nsect*.

5 Форматирование дорожки. Порожка определяется параметрами *head* и *track*. *Buffer* указывает на таблицу соответствия секторов и головок. Смотрите руководство "Technical Reference Manual" для описания этой таблицы и операции форматирования.

Только для XT, AT, PS/2:

6 Форматирование дорожки и установка флагов для плохих секторов

7 Форматирование начинается с определенной дорожки

8 Получить текущие параметры устройства. Информация возвращается в *buffer* в первых четырех байтах.

9 Инициализация параметров *disk-pair*

10 Длинное чтение, при котором читаются 512 байтов + 4 дополнительных байта из сектора

11 Длинная запись, при которой пишутся 512 байтов + 4 дополнительных байта в сектор

12 Позиционирование (*disk seek*)

13 Альтернативный сброс диска

14 Считать секторный буфер

15 Записать секторный буфер

16 Проверить на готовность указанное устройство

17 Перекалибровка устройства

18 Проверка RAM контроллера

19 Диагностика устройства

20 Внутренняя диагностика контроллера

Возвращаемое значение

При успешном завершении возвращается значение 0, при ошибке возвращается значение, отличное от 0.

Смотрите руководство "Technical Reference Manual" для подробного описания возможных ошибок.

BIOSEQUIP (TC), _BIOS_EQUIPLIST (MSC 5.1)

Использование (TC)

#include <bios.h>

int biosequip(void);

Использование (MSC 5.1)

#include <bios.h>

unsigned _bios_equlplist(void);

Описание

Эта функция возвращает целое значение, описывающее конфигурацию подключенных к компьютеру внешних устройств. Используется прерывание 0x11 BIOS.

Возвращаемое значение

Биты возвращаемого значения, если они установлены, указывают следующее:

15, 14 — число параллельных принтеров

13 — подключен последовательный принтер

12 — Джойстик (игровой порт в/в)

11-9 — число портов RS232

8 — отсутствие прямого доступа к памяти ПДП (DMA), 1 — когда отсутствует. (Этот бит используется не для всех версий BIOS)

7-6 —количество подключенных дисководов для гибких магнитных дисков:

00 — 1 устройство

01 — 2 устройства

10 — 3 устройства

11 — 4 устройства, только если нулевой бит равен 1

5-4 — видеорежим

01 — 40x25 BW with color card

10 — 80x25 BW with color card

11 — 80x25 BW with mono card

3-2 — размер системного RAM

00 — 16 К

01 — 32 К

10 — 48 К

11 — 64 К

1 — сопроцессор с плавающей точкой

0 — подключен ли хотя бы один жесткий диск (другая версия — признак загрузки с гибкого диска)

BIOSKEY (TC), _BIOS_KEYBRD (MSC 5.1)

Использование (TC)

#include <bios.h>

int bioskey(int cmd);

Использование (MSC 5.1)

#include <bios.h>

unsigned _bios_keybrd(unsigned cmd);

Описание

Эта функция позволяет работать с клавиатурой, используя прерывание 0x14 BIOS.

Параметр *cmd* определяет следующие операции:

0 - вернуть код клавиши, нажатой на клавиатуре.

Если младшие восемь разрядов содержат ненулевое значение, значит, введен ASCII-символ.

Если младшие восемь битов содержат нулевое значение, введен символ из расширенной кодировки, определенной в руководстве "Technical Reference Manual for the IBM PC".

1 — проверить, есть ли символ в буфере ввода (можем ли немедленно считать символ). Если нет, то возвращается ноль.

Если есть готовый символ для ввода, возвращается его код, однако этот символ не изымается из буфера и будет считан при следующем обращении к функции с параметром *cmd*=0.

2 — опросить текущее состояние клавиш-модификаторов (нажаты или нет). Установленные биты в младшем байте означают:

0x80 включен режим Insert

0x40 включен режим Caps

0x20 включен режим Num Lock

0x10 включен режим Scroll Lock

0x08 нажата клавиша Alt

0x04 нажата клавиша Ctrl

0x02 нажата клавиша Left Shift

0x01 нажата клавиша Right Shift

Возвращаемое значение

Описано выше.

BIOSMEMORY (TC), _BIOS_MEMSIZE (MSC 5.1)

Использование (TC)

 # include <bios.h>

int biosmemory(void);

Использование (MSC 5.1)

#include <bios.h>

unsigned _blos_memsize(void);

Описание

Функция использует прерывание 0x12 BIOS, чтобы определить общий объем оперативной памяти в компьютере.

Возвращаемое значение

Функция возвращает размер ОЗУ в блоках по 1 Кбайту (не учитывается расширенная память (*expended* и *extended*)).

Максимальное значение — 640.

BIOSPRINT (TC), _BIOS_PRINTER (MSC 5.1)

Использование (TC)

#include <bios.h>

int biosprtnt(cmd, byte, port);

int cmd;

int byte;

int port;

Использование (MSC 5.1)

#include <bios.h>

unsigned _bios_printer(cmd, port, byte);

unsigned cmd;

unsigned port;

unsigned byte;

/*обратите внимание на различный порядок следования аргументов*/

Описание

Функция предназначена для работы с принтером, указанным параметром *port*. Используется прерывание BIOS 0x17.

Если значение параметра *port* равно 0, это соответствует устройству LPT1, 1 — LPT2 и т. д.

Параметр *cmd* может принимать следующие значения:

0 напечатать символ, код которого задается значением *byte*;

1 инициализация принтера;

2 прочитать статус (состояние) принтера.

Возвращаемое значение

Если возвращаемое значение для операции *cmd=0* не равно 0, это свидетельствует об ошибке вывода символа.

Для всех операций возвращаемое значение - это текущее состояние принтера, которое определяется значениями следующих битов в возвращаемом значении:

0x01 — устройство не готово (*time out*)

0x08 — ошибка ввода/вывода

0x10 — выбор (*selected*)

0x20 — обрыв бумаги (*out of paper*)

0x40 — подтверждение (*acknowledge*)

0x80 — не занято (*not busy*)

Детальное описание смотри в руководстве "Technical Reference Manual for the IBM PC".

BIOSTIME (TC), _BIOS_TIMEOFDAY (MSC 5.1)

Использование (TC)

#include <bios.h>

long biostime(int cmd, long newtime);

Использование (MSC 5.1)

#include <bios.h>

unsigned _bios_timeofday(int cmd, long newtime);

Описание

Эта функция читает или устанавливает таймер BIOS. Используется прерывание BIOS 0x1A. Время определяется в тиках от полуночи. В каждой секунде содержится 18.2 тика.

Если значение *cmd* равно 0, функция возвращает текущее время в таймере.

Если значение *cmd* равно 1, устанавливается новое время в таймере.

Возвращаемое значение

Если *cmd*=0, возвращается текущее время.

BSEARCH (TC & MSC & ANSI)

Использование (TC)

#include <stdlib.h> /*используется только для описания функции*/

*void *bsearch(key, base, num, width, compare);*

*const void *key;*

*const void *base;*

size_t num, width;

*int(*compare)(const void *, const void *);*

Использование (MSC)

#include <stdlib.h> /*для совместимости с ANSI*/

#include <search.h> /*используется только для описания функции*/

*void *bsearch(key, base, num, width, compare);*

*char *key;*

*char *base;*

size_t num, width;

*int (*compare)();*

Описание

Функция **bsearch** выполняет поиск в отсортированном массиве из *num* элементов, где каждый из элементов массива имеет длину *width* байтов.

Значение параметра *base* задает адрес начала массива, в котором будет вестись поиск. Поиск ведется по ключу-значению, которое хранится по адресу, задаваемому значением параметра *key*.

Параметр *compare* определяет функцию, которая будет использоваться для определения, удовлетворяет ли текущий элемент массива условию поиска или нет. Подразумевается, что эта функция может сравнивать два элемента массива (на самом деле она сравнивает элемент массива и ключ-значение для поиска) и возвращает значение, определяющее их соотношение.

Функция **bsearch** будет вызывать функцию, определяемую параметром *compare*, один или более раз в течение поиска, на каждом шаге выбирая из оставшегося участка массива ту половину, где может располагаться искомый элемент.

Функция ***compare** должна сравнивать элементы, потом возвращать одно из следующих значений:

<0, если элемент 1 меньше элемента 2

0, если элемент 1 равен элементу 2

>0, если элемент 1 больше элемента 2

Отметим, в частности, что функции **strcmp** и **stricmp** удовлетворяют этим требованиям.

Возвращаемое значение

Указатель на первое месторасположение искомого элемента в массиве. Значение NULL, если элемент не найден.

Смотри также

lfind, lsearch, qsort

CABS (TC & MSC)

#include <math.h>

double cabs(z);

struct complex z;

Описание

Функция **cabs** вычисляет абсолютное значение комплексного числа.

Комплексное число представляется структурой типа *complex*, определенной в файле *math.h* в следующем виде:

struct complex

{

double x,y;

};

Вызов функции **cabs** эквивалентен вызову функции

sqrt(z.x*z.x+x.y*z.y)

Возвращаемое значение

Функция **cabs** возвращает абсолютное значение комплексного числа.

При переполнении возвращается значение HUGE_VAL, переменной *errno* присваивается значение:

ERANGE — результат вне допустимого диапазона значений.

Смотри также

abs, fabs, labs

CALLOC (TC & MSC & ANSI)

#include <stdlib.h> /*используется в СП TC, MSC*/

#include <malloc.h> /*используется в СП MSC*/

#include <alloc.h> /*используется в СП TC*/

*void *calloc(n, size);*

size_t n;

size_t size;

Описание

Библиотека языка Си предоставляет механизм распределения динамической памяти (*heap*). Этот механизм позволяет динамически (по мере возникновения необходимости) запрашивать из программы дополнительные области оперативной памяти.

В малых моделях памяти (*tiny*, *small*, *medium*) доступно для использования все пространство между концом сегмента статических данных программы и вершиной программного стека, за исключением 256-байтной буферной зоны непосредственно около вершины стека.

В больших моделях памяти (*compact*, *large*, *huge*) все пространство между стеком программы и верхней границей физической памяти доступно для динамического размещения памяти.

Функция **calloc** распределяет область памяти для массива из *n* элементов, каждый из которых имеет длину *size* байтов. Каждый элемент инициализируется нулем.

Имя типа *size_t* определяется следующим образом: *typedef unsigned size_t*;

Возвращаемое значение

Указатель на выделенную область памяти. NULL, если нет доступной памяти.

Смотри также

free, halloc, hfree, malloc, realloc, farcalloc

CEIL (TC & MSC & ANSI)

#include <math.h>

double ceil(x);

double x;

Описание

Выполняется округление для значения типа *double*. Функция **ceil** возвращает наименьшее целое, большее или равное значению числа с плавающей точкой.

Результат представляется в формате *double*.

Возвращаемое значение

Число с плавающей точкой.

Смотри также

floor, fmod

CGETS (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

*char *cgets(str);*

*char *str;*

Описание

Функция читает строку символов непосредственно с консоли и помещает строку и ее длину по указателю *str*. Аргумент *str* должен быть указателем на массив символов.

Первый элемент массива *str*[0] должен содержать максимальную допустимую длину считываемой строки. Массив должен иметь достаточную длину, чтобы поместить строку, символ конца строки ('\0') и два байта дополнительно.

Cgets читает символы до тех пор, пока не прочитает символы "возврат каретки" и "перевод строки" (комбинацию (CR-LF)), или пока не будет прочитано установленное число символов.

Прочитанная строка начинается со *str*[2]. Если прочитана комбинация символов CR-LF, эти символы располагаются за символом конца строки ('\0').

Действительная длина строки помещается в *str*[1].

Возвращаемое значение

Указатель на начало строки, т.е. адрес элемента *str*[2]. Нет кодов ошибок.

Смотри также

getch, getche

CHDIR (TC & MSC)

#include <dir.h> /*используется в СП TC*/

#include <direct.h> /*используется в СП MSC*/

int chdir(pathname);

*char *pathname;*

Описание

указывать на существующий каталог.

Функция **chdir** не позволяет изменить имя устройства, подразумеваемое по умолчанию (если указано имя устройства, то меняется текущий каталог на этом устройстве).

Возвращаемое значение

0, если текущий каталог успешно изменен или текущим был каталог, который определен в *pathname*.

1, если установленный в *pathname* каталог не может быть найден. При этом переменной *errno* присваивается значение ENOENT.

Смотри также

mkdir, rmdir, system, getcurdir, getcwd

CHMOD (TC & MSC), _CHMOD (TC)

Использование (MSC)

#include <sys\stat.h> /*Содержит описания констант*/

#include <sys\types.h> /*Содержит описания констант*/

#include <io.h> /*Содержит прототип функции*/

int chmod (pathname, pmode);

*char *pathname;*

int pmode;

Использование (TC)

#include <sys\stat.h> /*Содержит описания констант*/

#include <io.h> /*Содержит прототипы функций*/

int chmod (pathname, pmode);

*char *pathname;*

int pmode;

#include <dos.h> /*Содержит описания констант*/

 # include <io.h> /*Содержит прототип функции*/

int _chmod(pathname, func[, attrib]);

*char *pathname;*

int func;

int attrib;

Описание

Функция **chmod** изменяет права доступа, установленные для файла, который определен в *pathname*. Различаются права доступа для файла по чтению и записи.

Параметр *pmode* — константное выражение, содержащее одну или обе константы S_IWRITE и S_IREAD, определенные в файле *sys\stat.h*. Другие значения для параметра *pmode* игнорируются. Когда задаются обе константы, они разделяются операцией OR (|). Трактовка значений следующая:

S_IWRITE доступ для записи

S_IREAD доступ для чтения

S_IREAD|S-IWRITE доступ для чтения и записи.

В MS-DOS все файлы доступны для чтения, таким образом значения S_IWRITE и (S_IREAD | S_IWRITE) являются эквивалентными.

Функция **_chmod** (доступная только в системе программирования TC) позволяет узнать или установить атрибуты файла в ОС MS-DOS.

Значение параметра *func* определяет действие: 0 — вернуть текущие атрибуты файла, 1 — установить для файла новые атрибуты по значению параметра *attrib*.

Параметр *attrib* — константное выражение, составленное при помощи побитовой операции OR (|) из следующих констант, определенных в файле *dos.h*:

FA_RDONLY — атрибут доступа только по чтению

FA_HIDDEN — "невидимый" файл

FA_SYSTEM — системный файл

Возвращаемое значение

Функция **pmode** возвращает значение:

0 — права доступа успешно изменены;

1 — не может быть найден указанный файл, при этом переменной *errno* присваивается значение ENOENT.

Функция **_pmode** возвращает:

при успешном завершении операции — текущие атрибуты файла;

значение -1, если произошла ошибка, при этом переменной *errno* присваивается одно из значений:

ENOENT — не найден указанный файл,

EACCESS — нарушение прав.

Смотри также

access, creat, fstat, open, stat, unlink

CHSIZE (MSC & TC 2.0)

#include <io.h> /*используется только для описания функции*/

int chsize(handle, size);

int handle;

long size;

Описание

Функция **chsize** расширяет или сужает файл, имеющий дескриптор *handle*, до длины, задаваемой параметром *size*. Файл должен быть доступен для записи.

Если происходит расширение файла, добавляются нулевые символы ('\0').

Если файл сужается, все записи с конца файла обрезаются.

Возвращаемое значение

Значение 0 — если длина успешно изменена.

Значение — 1 обозначает ошибку, при этом переменной *errno* одно из следующих значений:

EACCES — указанный файл доступен только для чтения.

В MS-DOS 3.0 и в поздних версиях EACCES может обозначать, что файл закрыт для доступа.

EBADF — недействительный номер *handle*.

ENOSPC — недостаточно места на устройстве.

Смотри также

close, creat, open

_CLEAR87 (TC & MCS)

#include <float.h>

unsigned int _clear87(void);

Описание

Функция очищает слово состояния сопроцессора с плавающей точкой. Слово состояния является комбинацией слова состояния сопроцессора 8087/80287 и других условий, обнаруженных обработчиком исключительных состояний сопроцессора 8087/80287, такие как переполнение или исчерпание стека чисел с плавающей точкой.

Возвращаемое значение

Биты в возвращаемом значении соответствуют старому статусу сопроцессора с плавающей точкой.

В файле *float.h* описаны значения различных битов в значении, возвращаемом функцией **_clear87**.

Смотри также

_control87, _status87, _fpreset

CLEARERR (TC & MSC & ANSI)

#include <stdio.h>

void clearerr (stream);

*FILE *stream;*

Описание

Функция **clearerr** очищает признак ошибки и признак конца файла для указанного потока.

При работе с файлом признак ошибки автоматически не очищается, т.е. если установлен признак ошибки для указанного потока, то при последующих операциях с потоком будет возвращаться признак ошибки до тех пор, пока не будет вызвана функция **clearerr** или функция **rewind** (эти функции сбрасывают признак ошибки).

Возвращаемое значение

Функция не возвращает значения.

Смотри также

eof, feof, ferror, perror, rewind

CLOSE (TC & MSC), _CLOSE (TC)

#include <io.h> /*используется только для описания функции*/

int close(handle);

int handle;

int _close(handle); /*функция _close доступна только в системе программирования TC*/

int handle;

Описание

Функции **close** и **_close** закрывают файл, связанный с дескиптором *handle*.

Этот файл мог быть открыт с помощью одой из функций: **_creat**, **creat**, **creatnew**, **creattemp**, **dup**, **dup2**, **_open**, **open**.

Вызов функции **close** для файла, открытого в текстовом режиме, вызывает дозапись в конец файла символа <CNTRL/Z>. Чтобы избежать дозаписи этого символа в файл, необходимо воспользоваться для закрытия файла функцией **_close**.

Возвращаемое значение

Обе функции возвращают:

значение 0, если файл успешно закрыт.

значение -1, если указано неверное значение параметра *handle*; переменной *errno* при этом присваивается значение EBADF.

Смотри также

chsize, creat, dup, dup2, open, unlink, read, write

_CONTROL87 (MSC & TC 2.0)

#include <float.h>

unsigned int _control87 (new, mask);

unsigned int new;

unsigned int mask;

Описание

Функция получает и устанавливает слово управления сопроцессора с плавающей точкой. Слово управления позволяет программе изменить точность, способ округления, способ работы с предельно большими числами в пакете арифметики с плавающей точкой.

Также могут быть замаскированы или размаскированы исключительные ситуации.

Если значение *mask* равно 0, то функция **_control87** используется для получения текущего значения слова управления. Если значение *mask* ненулевое, то новое значение слова устанавливается следующим образом: для любого бита, который включен (равен 1) в *mask*, соответствующий бит в new используется, чтобы изменить слово управления. Другими словами, действие этой функции:

fpcntrl=((fpcntrl&~mask)|(new&mask)),

где *fpcntrl* является словом управления плавающей точки.

Возвращаемое значение

Биты в возвращаемом значении указывают состояние слова управления для операций с плавающей точкой.

Смотри включаемый файл *float.h* для получения описания бит в значении, возвращаемом функцией **_control87**.

Смотри также

_clear87, _status87

CORELEFT(TC)

Использование (для моделей памяти *tiny*, *small*, *medium*)

#include <alloc.h>

#include <stdlib.h>

unsigned int coreleft(void);

Использование (для моделей памяти *compact*, *large*, *huge*)

#include <alloc.h>

#include <stdlib.h>

unsigned long coreleft(void);

Описание

Функция возвращает приблизительный размер в байтах памяти, доступной для динамического распределения (смотри описание функции **malloc**).

Возвращаемое значение

Для моделей памяти *tiny*, *small*, *medium* функция возвращает значение, определяющее объем неиспользуемой оперативной памяти между областью статических данных программы и стеком, возвращаемое значение имеет тип *unsigned int*.

Для моделей памяти *compact*, *large*, *huge* функция возвращает значение, определяющее объем неиспользуемой оперативной памяти от области динамического распределения памяти (верхней границы программы) до стека, возвращаемое значение имеет тип *unsigned long*.

Смотри также

calloc, mailoc, realloc, _memavl

COS, COSH (TC & MSC & ANSI)

#include <math.h>

double cos(x);

double cosh(x);

double x;

Описание

Функции **cos** и **cosh** возвращают косинус *x* и гиперболический косинус *x* соответственно.

Возвращаемое значение

Функция **cos** возвращает косинус от *x*.

Если *x* большое и происходит потеря значащих разрядов, то функция **cos** генерирует ошибку PLOSS, но сообщение не печатает.

Если *x* такое большое, что происходит полная утрата результата, то **cos** печатает ошибочное сообщение TLOSS в поток *sdterr* и возвращает 0. В обоих случаях переменной *errno* присваивается значение ERANGE.

Функция **cosh** возвращает гиперболический косинус от *x*.

Если результат является очень большим, то функция **cosh** возвращает значение HUGE_VAL и присваивает переменной *errno* значение ERANGE.

Обработку ошибок для функций **cos** и **cosh** можно изменить, используя функцию **matherr**.

Смотри также

acos, asin, atan, atan2, matherr, sin, sinh, tan, tanh

COUNTRY (TC)

#include <dos.h>

*struct country *country(countrycode, countryp);*

int countrycode;

*struct country *countryp;*

Описание

Функция возвращает информацию о способе представления времени (дата, время суток и т. д.) и записей о денежных суммах в данной стране.

Работа функции зависит от конкретной версии ОС MS-DOS.

Если параметр *countryp* имеет значение -1, текущая страна определяется параметром *countrycode*, значение которого не должно быть равным 0. В противном случае, страна определяется параметром *countryp*, который указывает на структуру типа *country*.

Эта структура заполнена информацией, зависящей от страны: либо это текущая страна (если значение параметра *countrycode* равно 0), либо это страна, заданная параметром *countrycode*.

Структура *country* определяется в файле *dos.h* следующим образом:

```
struct country
{
int co_date /*Формат даты*/
char co_curr[5]; /*Символ обозначения валютной единицы*/
char co_thsep[2]; /*Символ-разделитель между тысячами*/
char co_desep[2]; /*Десятичный разделитель*/
char co_dtsep[2]; /*Разделитель в дате*/
char co_tmset[2]; /*Разделитель в записи о времени*/
char co_currstyle; /*Стиль записи денежных сумм*/
char co_digits; /*Количество значимых цифр записи денежных сумм*/
int(far *co_case)(); /*Функция, устанавливающая соответствие символов разных регистров*/
char co_dasep; /*Разделитель между данными*/
char co_fill[10]; /*Символ-заполнитель*/
};
```

Формат даты задается значением параметра *co_date*:

0 — для стиля даты, месяца и времени США;

1 — европейский стиль;

2 — японский стиль.

Текущий стиль изображения, задается значением параметра *co_currstyle*:

0 — перед текущим символом численное значение ставится без пробела;

1 — численное значение, следующее за символом, не отделяется от него пробелом;

2 — предшествующее текущему символу численное значение отделено от него пробелом;

3 — численное значение, следующее за символом, отделяется от него пробелом.

Возвращаемое значение

Возвращается значение указателя *countryp*.

Переносимость

Только для MS-DOS.

CPRINTF (TC & MSC)

#include <conto.h> /*используется только для описания функции*/

int cprintf(format[, argument…]);

*char *format;*

Описание

Функция **cprintf** форматирует и выводит последовательности символов и значений непосредственно на консольный терминал, используя для вывода символов функцию **putch**.

Каждый аргумент преобразуется и выводится в том формате, который определен в строке *format*. *Format*-строка имеет те же синтаксис и семантику, что и для функции **printf** (смотри описание функции **printf**).

В отличие от функций **fprintf**, **printf**, **sprintf**, функция **cprintf** не переводит символы *<LF>* в последовательности *<CR><LF>*.

Возвращаемое значение

Возвращается количество напечатанных символов.

Смотри также

fprintf, printf, sprintf, vprintf

CPUTS (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

int cputs(str);

*char *str;*

Описание

Функция **cputs** записывает строку (адрес которой задает параметр *str*) вместе с символом конца строки на консольный терминал.

Заметим, что не выводится комбинация символов "возврат каретки"/"перевод строки" (*<CR>*/*<LF>*).

Возвращаемое значение

Для системы программирования TC версии 1.5 и для системы программирования MSC версии 4.0 функция не возвращает значения.

Для системы программирования TC версии 2.0 возвращается последний напечатанный символ.

Для системы программирования MSC версии 5.1 возвращается значение:

0 — все нормально

1 — ошибка вывода

Смотри также

putch, fputs, puts

CREAT (TC & MSC), _CREAT (TC)

Использование (MSC)

#include <sys\stat.h> /*файл содержит описания констант*/

#include <sys\types.h> /*файл содержит описания констант*/

#include <io.h> /*файл содержит прототип функции*/

int creat(pathname, pmode);

*char *pathname;*

int pmode;

Использование (TC)

#include <sys\stat.h> /*файл содержит описания констант*/

#include <io.h> /*файл содержит прототипы функций*/

int creat(pathname, pmode);

*char *pathname;*

int pmode;

#include <dos.h> /*файл содержит описания констант*/

int _creat(pathname, attrib);

*char *pathname;*

int attrib;

Описание

Функция **creat** либо создает новый файл, либо открывает и сужает существующий файл.

Если файл, указанный в *pathname*, не существует, новый файл создается с заданным доступом и открывается для записи. Если файл уже существует с доступом для записи, **creat** сужает файл до длины 0, уничтожая предыдущие данные и открывая его для записи.

Если файл существует и доступен только по записи, то функция **creat** возвращает признак ошибки и файл остается неизмененным.

Права доступа к файлу, установленные в *pmode*, применимы только к вновь создаваемым файлам. Новый файл получает указанные права доступа после того, как он будет закрыт.

Параметр *pmode* — константное выражение, содержащее одну или обе константы S_IWRITE и S_IREAD, определенные в файле *sys\stat.h*. Другие значения для параметра pmode игнорируются. Когда задаются обе константы, они разделяются операцией OR (|). Трактовка значений следующая:

S_IWRITE доступ для записи

S_IREAD доступ для чтения

S_IREAD|S_IWRITE доступ для чтения и записи.

В MS-DOS все файлы доступны для чтения, таким образом значения S_IWRITE и (S_IREAD|S_IWRITE) являются эквивалентными.

Режим открытия файла (текстовый или двоичный) при использовании функции **creat** определяется по значению глобальной переменной *_fmode* (ее значением может быть одна из констант, O_TEXT или O_BINARY).

Чтобы открыть файл в другом режиме, можно либо установить соответствующее значение переменной *_fmode*, либо открыть файл через вызов функции **open** с установленными атрибутами O_CREAT и O_TRUNC и соответствующим атрибутом O_TEXT или O_BYNARY, например, вызов **open** ("TMP", O_CREAT|O_TRUNC|O_BINARY, S_IREAD) может быть использован для создания файла в двоичном режиме, доступного только по чтению, с усечением длины файла до 0 байт, если файл уже существует.

Функция **_creat** выполняет те же действия, что и функция **creat**, но при этом:

1) файл всегда открывается в двоичном режиме;

2) файл открывается одновременно на чтение и на запись;

3) параметр *attrib* — константное выражение, составленное при помощи побитовой операции OR (|) из следующих констант, определенных в файле *dos.h*:

FA_RDONLY — атрибут доступа только по чтению

FA_HIDDEN — "невидимый" файл

FA_SYSTEM — системный файл

Возвращаемое значение

Функции **creat** и **_creat** возвращают значение дескриптора (*handle*) для созданного файла.

Значение -1 сигнализирует об ошибке, при этом переменной *errno* присваивается одно из значений:

EACCES *pathname* указывает на существующий файл с доступом только для чтения или указывает на каталог вместо файла.

EMFILE — нет больше доступных файлов (т.е. слишком много открытых файлов).

ENOENT — файл не найден.

Смотри также

creatnew, creattemp, chmod, chsize, close, dup, dup2, open, sopen, umask

CREATNEW (TC), CREATTEMP (TC)

#include <dos.h> /*файл содержит описания констант*/

#include <io.h> /*файл содержит прототипы функций*/

int cratnew(pathname, attrib);

*char *pathname;*

int attrib;

int creattemp(pathname, attrib);

*char *pathname;*

int attrib;

Описание

Функция **creatnew** используется для создания новых файлов (без затирания существующих), функция **creattemp** используется для создания новых файлов.

Функция **creatnew** эквивалентна функции **_creat** с тем отличием, что если файл существует, то функция возвращает признак ошибки и файл остается неизмененным.

Функция **creattemp** похожа на функцию **_creat** с тем отличием, что имя файла (имя пути), задаваемое параметром pathname, заканчивается символом обратной косой черты ('\\'). Для указанного каталога выбирается уникальное имя файла. Имя файла дописывается в конец строки, адрес которой передается через параметр *pathname* (должен быть соответствующий резерв в конце строки). После нормального завершения программы созданный файл автоматически уничтожается.

Соглашения об открытии файла для функций **creatnew** и **creattemp**:

1) файл всегда открывается в двоичном режиме;

2) файл открывается на чтение и на запись;

3) параметр *attrib* — константное выражение, составленное при помощи побитовой операции OR(|) из следующих констант, определенных в файле *dos.h*:

FA_RDONLY — атрибут доступа только по чтению

FA_HIDDEN — "невидимый" файл

FA_SYSTEM — системный файл

Возвращаемое значение

Функции **creatnew** и **creattemp** возвращают значение дескриптора (*handle*) для созданного файла.

Значение -1 сигнализирует об ошибке, при этом переменной *errno* присваивается одно из значений:

EACCES *pathname* указывает на существующий файл с доступом только для чтения или указывает на каталог вместо файла.

EMFILE — нет больше доступных файлов (т.е. слишком много открытых файлов).

ENOENT — файл не найден.

Смотри также

creat, chmod, chsize, close, dup, dup2, open, sopen, umask, tempnam, tmpnam

CSCANF (TC & MSC)

#include <conio.h>

int cscanf(format[, argument…]);

*char *format;*

Описание

Функция **cscanf** читает данные с консоли, выполняет форматные преобразования и помещает значения в переменные, адреса которых определяются аргументами функции.

Функция **cscanf** использует функцию **getche** для ввода символов.

Каждый аргумент должен указывать на переменную того типа, который соответствует ее описанию в строке *format*.

Смотри описание функции **scanf**.

Возвращаемое значение

Количество элементов ввода, которые были успешно преобразованы и присвоены.

Значение EOF, если встретился символ-признак конец файла.

Значение 0, если нет присвоенных областей.

Смотри также

fscanf, scanf, sscanf, vscanf, vfscanf

CTIME (TC & MSC & ANSI)

#include <time.h> /*используется только для описания функции*/

*char *ctime(time);*

*const time_t *time;*

Описание

Функция **ctime** преобразует время из длинного целого в строку символов.

Значение **time** обычно получают, вызывая функцию **time**, которая возвращает число секунд, прошедших с 1 января 1970 года.

Строка-результат после выполнения **ctime** содержит 26 символов и имеет вид, показанный на следующем примере:

"Ven Jul 07 05:30:05 1989\n\0".

Каждое поле имеет символьный формат. Символ новой строки ('\n') и нулевой символ ('\0') занимают две последние позиции в строке.

Возвращаемое значение

Указатель на строку-результат.

Нет кодов ошибок.

Смотри также

acstime, ftime, gmtime, localtime, time

CTRLBRK (TC)

#include <dos.h>

void ctrlbrk(fptr);

*int(*fptr)(void);*

Описание

Функция **ctrlbrk** устанавливает новую функцию (определяемую параметром *fptr*) обработки ситуации, когда пользователь нажимает на клавиатуре комбинацию клавиш *control-break*.

Используется системный вызов MS-DOS 0x23.

Устанавливаемая функция обработки не вызывается непосредственно. Функция **ctrlbrk** делает так, что обработчик прерываний DOS вызовет данную функцию.

Обрабатывающая функция может выполнять любое количество операций и системных вызовов.

Обработчик не может возвратить управление в программу через оператор *return*; он может использовать функцию **longjmp** для возврата в произвольную точку программы (через эту функцию он может возвращать значение).

Возвращаемое значение

Функция **ctrlbrk** не возвращает значения.

Функция-обработчик возвращает 0 для прерывания выполнения текущей программы; любое другое значение вызовет продолжение выполнения программы.

Переносимость

Только для MS-DOS.

Смотри также

longjmp, setjmp

DIEEETOMSBIN, DMSBINTOIEEE (MSC)

#include <math.h>

int dieeetomsbin(src8, dst8);

int dmsbintoieee(src8, dst8);

*double *src8, *dst8;*

Описание

Функция **dieeetomsbin** преобразует число двойной точности в IEEE-формате в Microsoft-двоичный формат.

Команда **dmsbintoieee** преобразует число двойной точности в Microsoft-двоичном формате в IEEE-формат.

Аргумент *src8* является указателем на значение типа *double*, которое будет преобразовываться.

Результат преобразования заносится по адресу, определяемому параметром *dst8*.

Эти функции позволяют программам на Си (которые хранят вещественные числа в в формате

IEEE) использовать числовые данные из файлов, подготовленных программами, созданными с помощью тех версий системы программирования Microsoft BASIC, которые используют для представления вещественных чисел двоичный формат Microsoft.

Эти функции не обрабатывают "специальные" числа IEEE-формата, такие как NAN и бесконечность, такие числа интерпретируются при преобразованиях как значение 0.0.

Возвращаемое значение

0, если преобразование выполнено успешно.

1, если произошло переполнение.

Смотри также

fieeetomsbin, fmsbintoieee

DIFFTIME (TC & MSC & ANSI)

#include <time.h>

double difftime(time2, time1);

time_t time2;

time_t time1;

Описание

Функция вычисляет разницу в секундах между временем, задаваемым параметром *time2*, и временем, задаваемым параметром *time1*.

Имя типа *time_t* определяется в файле *time.h* следующим образом:

typedef long time_t;

Возвращаемое значение

Число двойной точности, которое является разницей в секундах от *time1* до *time2*.

Смотри также

time

DISABLE (TC), _DISABLE (MSC 5.1)

Использование (MSC)

#include <dos.h>

void disable(void);

Использование (MSC 5.1)

#include <dos.h>

void _disable(void);

Описание

Функция отменяет прерывания.

Допускается только немаскируемое прерывание NMI от любого внешнего устройства.

Вызывается машинная инструкция CLI.

Возвращаемое значение

Функция не возвращает значения.

Переносимость

Только для архитектуры с микропроцессорами семейства Intel 8086/80286.

Смотри также

enable, getvect

DIV (TC 2.0 & ANSI)

#include <stdlib.h>

div_t div(number, denom);

int number;

int denom;

Описание

Функция **div** предназначена для выполнения деления двух чисел типа *int*, с выделением частного и остатка от деления.

Значения параметров *number* и *denom* задают делимое и делитель соответственно.

Тип *div_t* описан в файле *stdlib.h* следующим образом:

typedef struct

{

int quot; /*частное*/

int rem; /*остаток от деления*/

} div_t;

Возвращаемое значение

Возвращается структура из двух элементов (возвращаемое значение — сама структура, а не ее адрес), которые содержат частное и остаток от деления.

Смотри также

ldiv

DOSEXTERR (TC & MSC)

#include <dos.h>

int dosexterr(buffer);

*struct DOSERROR *buffer;*

Описание

Функция **dosexterr** использует прерывание DOS 0x59. Эта функция позволяет получить подробную информацию об ошибке, произошедшей при вызове предыдущей функции DOS.

Функция записывает значения регистров, возвращаемые системным вызовом MS-DOS, в структуру, на которую указывает *buffer*.

Структура типа DOSERROR определена в *dos.h* и имеет следующий вид:

```
struct DOSERROR

{

int exterror; /*значение регистра AX, признак ошибки*/

char class; /*значение регистра BH, тип ошибки*/

char action; /*значение регистра BL, действие*/

char locus; /*значение регистра CH*/

}
```

Для системы программирования MSC, если в качестве аргумента задано значение NULL, функция **dosexterr** возвращает значение (регистра AX) без заполнения полей структуры.

Функция работает для версий ОС MS-DOS 3.0 и выше.

Возвращаемое значение

Значение в регистре AX (идентичное значению поля *exterror*). Если возвращается значение 0, значит, во время выполнения предыдущего системного вызова DOS не было ошибки.

Смотри также

perror

DOSTOUNIX (TC)

```
#include <dos.h>

long dostounix(dateptr, timeptr);

struct date *dateptr;

struct time *timeptr;
```

Описание

Функция **dostounix** преобразует дату и время, возвращаемые функциями **getdate** и **gettime**, в формат ОС UNIX.

Параметр *dateptr* — указатель на структуру date, параметр *timeptr* — указатель на структуру *time*, эти структуры содержат информацию о времени и дате, в формате, соответствующем данной версии ОС MS-DOS.

Возвращаемое значение

Функция **dostounix** возвращает текущее время в формате ОС UNIX: число секунд со времени

00:00:00 1 января 1970 г. (GMT).

Только для MS-DOS.

Смотри также

ctime, getdate, gettime, unixtodos

DUP, DUP2 (TC & MSC)

#include <io.h> /*используется только для описания функции*/

int dup(handle);

int handle;

int dup2(handle1, handle2);

int handle1;

int handle2;

Описание

Функции **dup** и **dup2** позволяют связать второй дескриптор (*handle*) с уже открытым файлом.

В операциях с файлом можно использовать любой *handle*, так как все *handle*, связанные с заданным файлом, используют один и тот же (внутренний) указатель файла.

Способ доступа, разрешенный для файла, не меняется при создании нового дескриптора.

Функция **dup** возвращает еще один дескриптор (*handle*) для заданного файла.

Функция **dup2** принудительно заставляет дескриптор *handle2* ссылаться на тот же файл, что и дескриптор *handle1*. Если *handle2* связан с открытым файлом во время вызова функции **dup2**, то этот файл закрывается.

Возвращаемое значение

Функция **dup** возвращает новый дескриптор файла (*handle*).

Функция **dup2** возвращает значение 0, если операция выполнена успешно.

Обе функции возвращают значение -1, если произошла ошибка; и переменной *errno* присваивается одно из следующих значений:

EBADE недействительный дескриптор *handle*;

EMFILE нет больше доступных *handle* (слишком много открытых файлов).

Смотри также

close, creat, open

ECVT (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

*char *ecvt(value, ndigits, decptr, signptr);*

double value;

int ndigits;

*int *decptr;*

*int *signptr;*

Описание

Функция **ecvt** преобразует число с плавающей точкой в строку символов.

Функция **ecvt** записывает *ndigits* цифр числа *value* в строку и добавляет в конец строки нулевой байт ('\0').

Если число цифр в *value* больше, чем *ndigits*, то отсекаются младшие разряды.

Если число цифр меньше, чем *ndigits*, то строка дополняется нулями.

В возвращаемой строке помещаются только цифры. Позиция десятичной точки и знак числа записываются в переменные, адреса которых определяются параметрами *decptr* и *signptr*.

Параметр *decptr* указывает на целое, значение которого является позицией десятичной точки от начала строки. Значение 0 или отрицательное число говорит о том, что позиция десятичной точки находится слева от первой цифры.

Параметр *signptr* является указателем на целое, обозначающее знак числа. Если значение 0, то число положительно. В других случаях число отрицательно.

Возвращаемое значение

Указатель на строку цифр.

Замечание. Функции **ecvt** и **fcvt** используют один и тот же буфер для преобразований. При каждом вызове одной из этих функций результат предыдущего вызова теряется.

Смотри также

atof, atoi, atol, fcvt, gcvt, printf

ENABLE (TC), _ENABLE (MSC 5.1)

Использование (MSC)

#include <dos.h>

void enable(void);

Использование (MSC 5.1)

#include <dos.h>

void _enable(void);

Описание

Функция разрешает прерывания. Допускаются прерывания от всех устройств.

Вызывается машинная инструкция STI.

Возвращаемое значение

Функция не возвращает значения.

Переносимость

Только для архитектуры с микропроцессорами семейства Intel 8086/80286.

Смотри также

disable, getvect

EOF (TC & MSC)

#include <io.h> /*требуется только для описания функции*/

int eof(handle);

int handle;

Описание

Функция **eof** определяет — был ли достигнут конец файла, связанного с дескриптором *handle*.

Возвращаемое значение

1, если текущая позиция есть конец файла

0, если не конец файла.

-1, неверный дескриптор *handle*, при этом переменной *errno* присваивается значение EBADF.

Смотри также

clearerr, feof, ferror, perror, open, read, write

EXEC... (TC & MSC)

#include <process.h> /*требуется только для описания функции*/

int execl(pathname, arg0, arg1, ..., argn, NULL);

int execle(pathname, arg0, arg1, ..., argn, NULL, envp);

int execlp(pathname, arg0, arg1, ..., argn, NULL);

int execlpe(pathname, arg0, arg1, ..., argn, NULL, envp);

int execv(pathname, argv);

int execve(pathname, argv, envp);

int execvp(pathname, argv);

int execvpe(pathname, argv, envp);

*char *pathname;*

*char *arg0, *arg1, ... , argn;*

char argv[];

char envp[];

Описание

Эти функции загружают и выполняют новые порождаемые процессы.

Если вызов прошел успешно, порожденный процесс располагается в памяти на том месте, где раньше располагался вызвавший процесс. Должно быть достаточно оперативной памяти для загрузки и выполнения порожденного процесса.

Аргумент *pathname* определяет файл, который должен быть выполнен как порожденный процесс. *Pathname* может определить весь путь (начиная с корня), частичный путь (с текущего каталога) или просто имя файла.

Если *pathname* не имеет расширения и не заканчивается точкой, то **exec**-функция сначала добавляет справа расширение "COM" и ищет файл; если файл найти не удается, то добавляется расширение "EXE". Если в строке, задаваемой аргументом *pathname*, указано расширение имени файла, то используется только оно. Если *pathname* заканчивается с точкой, **exec**-функция ищет файл без расширения.

Функции **execlp** и **execvp** ведут поиск имени файла в каталогах, определенных в PATH-переменных оболочки ОС MS-DOS.

Аргументы передаются новому процессу заданием одного или более указателей на строки символов, как аргументы при вызове **exec**-функций.

Эти символьные строки составляют список аргументов для порождаемого процесса.

Суммарная длина строк, формирующая список аргументов для порождаемого процесса, не должна превышать 128 байтов. Нулевые символы ('\0') в конце каждой строки не учитываются, но пробелы (разделяющие аргументы) учитываются.

Аргументы могут быть переданы по отдельности (при вызове функций **execl**, **execle**, **execlp**) или как массив указателей (при вызове функций **execv**, **execve**, **execvp**).

Минимум один аргумент, *arg0* или *argv[0]*, должен быть передан в порождаемый процесс. Обычно этот аргумент является копией значения аргумента pathname.

Вызов функций **execl**, **execle**, **execlp** используется обычно в тех случаях, когда число аргументов известно заранее. Аргумент *arg0* является обычно указателем на *pathname*. Аргументы от *arg0* до *argn* являются указателями на символьные строки, составляющие список аргументов. Следующий за *argn* параметр должен быть указателем NULL, что обозначает конец списка аргументов.

Функции **execv**, **execve**, **execvp** используются обычно в тех случаях, когда число аргументов для нового процесса переменно.

Указатели на аргументы передаются как массив *argv*, состоящий из указателей. Указатель *argv[0]* является обычно указателем на *pathname*. Указатели от *argv[1]* до *argv[n]* являются указателями на символьные строки, составляющие список аргументов.

Указатель *argv[n+1]* должен иметь значение NULL, что обозначает конец списка аргументов.

Файлы, которые открыты на момент вызова какой-либо **exec**-функции, остаются открытыми для нового процесса.

При вызове функций **execl**, **execlp**, **execv** и **execvp** порождаемый процесс унаследует также окружение родителя.

Функции **execle** и **execve** позволяют изменить окружение для порождаемого процесса, передав список параметров окружающей среды через аргумент *envp*. Аргумент envp является массивом символьных указателей, каждый элемент которого является указателем на строку, определяющую переменную из окружения. Такая строка обычно имеет следующую форму:

NAME=value

NAME — имя переменной окружения;

value — строковое значение, в которое переменная устанавливается.

Если значение *envp* равно NULL, то окружающая среда унаследуется у родителя.

Возвращаемое значение

Значение -1 сигнализирует об ошибке, при этом переменной *errno* присваивается одно из следующих значений:

E2BIA — список аргументов превышает 128 байтов или память, требуемая для информации об окружающей среде, превышает 32 байта.

EACCES — нарушение прав доступа к файлу.

EMFILE — слишком много открытых файлов (указанный в команде файл открывается, чтобы проверить, можно ли его выполнить).

ENOENT — файл или путь не найдены.

ENOEXEC — указанный файл не является выполняемым или имеет неправильный формат, чтобы его можно было выполнить.

ENOMEM — нет достаточной памяти, чтобы выполнить порождаемый процесс; или доступная память испорчена: существуют блоки неверной длины, что означает, что процесс-родитель был плохо размещен в памяти.

Замечание 1. При вызове *exec* не передаются режимы '*t*' и '*b*' (текстовый и двоичный) использования открытых файлов. Если порожденный процесс должен использовать файлы, унаследованные от родителя, то должна быть использована команда *setmode*, чтобы установить необходимый режим (если этот режим отличен от режима по умолчанию). Установленная реакция на сигналы также не сохраняется, в порожденном процессе происходит установка реакции по умолчанию.

Замечание 2. В ОС MS-DOS версий 3.0 и выше первый передаваемый порождаемому процессу параметр (*arg0* или *argv*[0]) содержит адрес строки, содержащей имя пути для файла, из которого запускается процесс.

Смотри также

abort, exit, _exit, onexit, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, system

EXIT (TC & MSC & ANSI), _EXIT (TC & MSC)

#include <process.h>

#include <stdlib.h> /*используйте либо process.h, либо stdlib.h*/

void exit(status);

void _exit(status);

int status; /*статус завершения*/

Описание функции _EXIT

Функции **exit** и **_exit** завершают вызванный процесс.

Функция **_exit** завершает процесс без сбрасывания буферов, связанных с потоками файлов, и без вызова функций завершения.

Значение аргумента *status*, равное нулю, сигнализирует (по соглашению в рамках операционной системы) о нормальном завершении процесса, значение, отличное от нуля, сигнализирует об ошибке.

Описание функции EXIT (MSC 4.0 & TC 1.5)

Функция **exit** скидывает все буфера и закрывает все открытые файлы перед завершением процесса.

Значение аргумента *status*, равное нулю, сигнализирует (по соглашению в рамках операционной системы) о нормальном завершении процесса, значение, отличное от нуля, сигнализирует об ошибке.

Описание функции EXIT (TC 2.0)

Скидывает все буфера открытых файлов и закрывает их, вызывает функции, зарегистрированные с помощью функции **atexit()**, затем завершает процесс.

Значение аргумента *status*, равное нулю, сигнализирует (по соглашению в рамках операционной системы) о нормальном завершении процесса, значение, отличное от нуля, сигнализирует об ошибке. Для задания значения параметра *status* могут быть использованы константы:

EXIT_SUCCESS — нормальное завершение.

EXIT_FAILURE — аварийное завершение.

Описание функции EXIT (MSC 5.1)

Сначала вызываются функции, зарегистрированные с помощью *atexit()* и *onexit()*, затем сбрасываются все буфера открытых файлов и закрываются эти файлы, затем завершается текущий процесс.

Значение аргумента *status*, равное нулю, сигнализирует (по соглашению в рамках операционной системы) о нормальном завершении процесса, значение, отличное от нуля, сигнализирует об ошибке. Младший байт значения *status* доступен ожидающему процессу-родителю, если такой есть, после завершения текущего процесса.

Смотри также

abort, execl, execle, execlp, execv, execve, execvp, onexit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, system.

EXP (TC & MSC & ANSI)

#include <math.h>

double exp(x);

double x;

Описание

Функция возвращает экспоненту от аргумента, числа с плавающей точкой.

Возвращаемое значение

Экспонента от аргумента (e^x).

Возвращается значение HUGE_VAL, если произошло переполнение, при этом переменной *errno* присваивается значение ERANGE.

Возвращается значение 0, если происходит потеря точности, *errno* не изменяется.

Смотри также

log, log10, frexp, ldexp, matherr, pow, pow10, sqrt

_EXPAND (MSC)

#include <malloc.h> /*используется только для описания функции*/

*char *_expand(ptr, size);*

*char *ptr;*

unsigned size;

Описание

Функция **_expand** изменяет размер блока памяти, ранее выделенного с помощью функции **malloc** или **calloc**, не меняя при этом место расположения блока.

Аргумент *ptr* указывает на начало блока. Аргумент *size* задает новый размер блока в байтах. Содержимое блока не изменяется.

Аргумент *ptr* может также указывать на блок, который был освобожден (при помощи функции **free**), до тех пор, пока не будет сделан вызов одной из функций **calloc**, **_expand**, **halloc**, **malloc**, или **realloc** после освобождения блока. Необходимо учесть при этом, что некоторые библиотечные функции, например функция **fopen**, сами вызывают функцию **malloc**.

Если *ptr* указывает на освобожденный блок, то блок будет оставаться освобожденным и после вызова функции **_expand**.

Возвращаемое значение

Символьный указатель на переопределенный блок памяти. В отличие от **realloc**, **_expand** не может переместить блок при изменении его размера.

NULL, если нет возможности расширить блок до заданных размеров без перемещения блока. Область памяти, на которую указывает возвращаемое значение, является выравненной для размещения объектов любого типа. Новый размер памяти может быть проверен функцией **_msize**.

Чтобы получить указатель типа, отличного от *char*, необходимо использовать явное преобразование типа для возвращаемого значения.

Смотри также

calloc, free, halloc, malloc, _msize, realloc

FABS (TC & MSC & ANSI)

#include <math.h>

double fabs(x);

double x;

Описание

Функция возвращает абсолютное значение числа с плавающей точкой.

Возвращаемое значение

Абсолютное значение аргумента. Нет кодов ошибок.

Смотри также

abs, cabs, labs

FARCALLOC, FARMALLOC, FARCORELEFT, FARFREE, FARREALLOC (TC)

#include <alloc.h>

*void far *farcalloc(nunits, unitsz);*

unsigned long nunits;

unsigned long unitsz;

*void far *farmalloc(size);*

unsigned long size;

unsigned long farcoreleft(void);

void farfree(block);

*void far *block;*

*void far *farrealloc(block, newsize);*

*void far *block;*

unsigned long newsize;

Описание

Функции по назначению подобны функциям **malloc**, **calloc**, **coreleft**, **free**, **realloc**, но употребляются для динамического размещения памяти в области размера больше 64 Кбайтов. Смотри также описания перечисленных функций.

Отличия **far**-функций:

— для динамического распределения доступна вся свободная оперативная память;

— могут быть выделены блоки размером более 64 Кбайтов;

— должны быть использованы дальние (*far*) указатели для доступа к выделенным блокам.

Для моделей памяти *compact*, *large*, *huge* эти функции подобны, хотя и не идентичны, функциям без *far*-приставок. Отличие в том, что аргументы имеют тип *unsigned long*, а не *unsigned*.

В *tiny*-модели памяти нельзя использовать эти функции, если программа должна быть записана в файл в .COM-формате.

Для моделей памяти *small* и *medium* блоки, размещенные с помощью функции **farmalloc** или **farcalloc**, могут быть освобождены только с помощью функции **farfree**, размещенные с помощью

моделей памяти создаются две независимые области для динамического размещения памяти.

Замечание. Система программирования MSC предоставляет альтернативный набор функций для динамического распределения памяти из области более 64 Кбайтов:

_fmalloc — получить блок памяти вне данного сегмента;

_ffree — освободить блок, полученный посредством **_fmalloc**;

_freect — возвращает примерное число областей заданного размера, которые можно получить;

_fmsize — возвращает размер блока памяти, на который указывает далекий (*far*) указатель;

halloc — получить память для большого массива;

hfree — освободить блок памяти, полученный посредством **halloc**.

Смотри также

Функции, общие для систем программирования MSC и TC:

calloc, coreleft, free, malloc

Функции, доступные только в системе программирования MSC:

alloca, _expand, _ffree, _fmalloc, _freect, _fmsize, halloc, hfree, _memavl, _msize, _nfree, _nmalloc, _nmsize, stackavail

FCLOSE (TC & MSC & ANSI), FCLOSEALL (TC & MSC)

#include <stdio.h>

int fclose(stream);

*FILE *stream;*

int fcloseall();

Описание

Функции **fclose** и **fcloseall** закрывают поток или потоки, связанные с открытыми при помощи функции **fopen** файлами.

Все буфера, связанные с потоками, сбрасываются перед закрытием. Память из-под системных буферов автоматически освобождается при закрытии потока.

Память из-под буферов, размещенных по команде *setbuf* или *setvbuf*, автоматически не освобождается.

Функция **fclose** закрывает поток, определяемый аргументом *stream*.

Функция **fcloseall** закрывает все потоки, за исключением следующих:

(TC 1.5) — **stdin**, **stdout**.

(MSC 4.0) — **stdin**, **stdout**, **stdoux**, **stdprn**.

(TC 2.0 & MSC 5.1) — **stdin**, **stdout**, **stdoux**, **stdprn**, **stdaux**.

Возвращаемое значение

Функция **fclose** возвращает значение 0, если поток успешно закрыт.

Функция **fcloseall** возвращает количество закрытых потоков.

Обе функции возвращают значение **EOF**, если произошла ошибка.

Смотри также

close, fdopen, fflush, fopen, freopen

FCVT (TC & MSC)

#include <stdlib.h> /*требуется только для описания функции*/

char fcvt(value, ndec, decptr, signptr);

double value;

int ndec;

*int *decptr;*

*int *signptr;*

Описание

Функция преобразует число с плавающей точкой в строку символов.

Значение параметра *value* — число с плавающей точкой, которое должно быть преобразовано.

Функция **fcvt** помещает цифры числа в строку с завершающим нулевым байтом ('\0'). Значение параметра *ndec* определяет число цифр после десятичной точки. Если число цифр после десятичной точки в значении *value* больше *ndec*, то число округляется, согласно преобразованиям в F-формате вывода оператора WRITE языка программирования FORTRAN-IV. Если меньше, то строка дополняется нулями.

В строке хранятся только цифры. Позиция десятичной точки и знак числа присваиваются переменным, адреса которых задаются параметрами *decptr* и *stgntr*, откуда могут быть извлечены после вызова функции **fcvt**.

Параметр *decptr* указывает на целое значение, которое является позицией десятичной точки от начала строки. Ноль или отрицательное значение означают, что десятичная точка располагается левее первой цифры в строке. Значение **signtr* обозначает знак числа. Нулевое значение означает, что число положительно; отличное от нуля значение означает, что число отрицательно.

Возвращаемое значение

Указатель на строку цифр.

Нет ошибочных кодов возврата.

Замечание. Функции **ecvt** и **fcvt** используют один и тот же буфер для преобразований. При каждом вызове одной из этих команд результаты предыдущего вызова теряются.

Смотри также

atof, atoi, atol, ecvt, gcvt

FDOPEN (TC & MSC)

#include <stdlib.h>

*FILE *fdopen(handle, type);*

int handle;

*char *type;*

Описание

Функция **fdopen** связывает поток ввода/вывода (ввод/вывод верхнего уровня) с файлом, связанным с дескриптором *handle* (нижний уровень ввода/вывода).

Таким образом, есть возможность для файлов, открытых с помощью библиотечных функций низкоуровневого ввода/вывода (например, при помощи функции **open**), начать буферизацию и работать с ними через функции форматированного ввода/вывода.

Аргумент *type* указывает на строку символов, определяющих способ доступа к файлу, и может принимать следующие значения:

"*r*" Открыть для чтения (файл должен существовать);

"*w*" Открыть пустой файл для записи; если файл существует, то его содержимое теряется;

"*a*" Открыть для записи в конец файла (добавления); файл создается, если он не существует;

"*r+*" Открыть для чтения и записи (файл должен существовать);

"*w+*" Открыть пустой файл для чтения и записи; если файл существует, ее содержание теряется;

"*a+*" Открыть для чтения и добавления; файл создается, если он не существует.

Замечание. Будьте осторожны при использовании "*w*" и "*w+*", т.к. они могут запортить существующие файлы.

Определяемый тип должен быть совместим с правами доступа к файлу.

Когда файл открыт в режиме "*a*" или "*a+*", данные записываются в конец файла. Хотя указатель файла может быть перемещен, используя функции **fseek** или **rewind**, он всегда будет перемещаться обратно на конец файла при записи данных. Таким образом, существующие данные не могут быть затерты.

Когда определен один из режимов "*r+*","*w+*" или "*a+*", разрешено чтение и запись (т.е. файл открыт для изменения ("*update*")). Однако при переключении с чтения на запись и наоборот должно осуществляться "ручное" позиционирование указателя по файлу с помощью функций **fseek**, **rewind**, **fsetpos**.

Для того чтобы указать, что файл должен быть открыт в текстовом режиме, необходимо добавить символ '*t*' в строку *type*, если в двоичном — '*b*'.

В текстовом режиме последовательности символов *<CR><LF>* преобразуются в символ *<LF>* при вводе. Символы *<LF>* преобразуются в последовательности символов *<CR><LF>* при выводе. Кроме того, *<CTRL/Z>* интерпретируется как символ конца файла при вводе. Для файлов, открываемых на чтение или чтение/запись, по возможности происходит проверка и удаление символов *<CTRL/Z>* (это необходимо для корректной работы *fseek*).

В двоичном режиме перечисленные выше преобразования не производятся.

Если не указан ни символ '*b*', ни символ '*t*', принимается режим, задаваемый по умолчанию

глобальной переменной *_fmode*.

Возвращаемое значение

Указатель на открытый поток.

Значение NULL, если обнаружена ошибка.

Смотри также

dup, dup2, fclose, fcloseall, fopen, freopen, open.

FEOF (TC & MSC & ANSI)

#include <stdio.h>

int feof(stream);

*FILE *stream;* /*Указатель на структуру FILE*/

Описание

Функция определяет, достигнут ли конец данного потока.

Если конец потока (т.е. соответствующего потоку файла) достигнут, операция чтения будет возвращать символ конца файла (значение EOF) до тех пор, пока поток не будет закрыт или не будет вызвана функция **rewind**.

Возвращаемое значение

Ненулевое значение, если достигнут конец файла (это будет обнаружено после первого обращения к операции чтения, которой не хватило данных в файле).

Нулевое значение, если не конец файла.

Замечание. **feof** выполняется как макрокоманда.

Смотри также

clearerr, eof, ferror, perror.

FERROR (TC & MSC & ANSI)

#include <stdio.h>

int ferror(stream);

*FILE *stream;*

Описание

Функция осуществляет проверку наличия ошибки при чтении/записи данных для заданного потока.

Если имела место ошибка, то (внутренний) индикатор ошибки для потока остается установленным до тех пор, пока поток не будет закрыт или не будет вызвана функция **rewind** или **clearerr**, соответственно, на протяжении этого периода времени обращение к функции **feof** будет оканчиваться возвратом признака ошибки ввода/вывода.

Возвращаемое значение

0, если нет ошибок.

Ненулевое значение, если встретилась ошибка для заданного потока.

Замечание. **ferror** выполняется как макрокоманда.

Смотри также

clearerr, eof, feof, fopen, perror.

FFLUSH (TC & MSC & ANSI)

#include <stdio.h>

int fflush(stream);

*FILE *stream;*

Описание

Функция сбрасывает содержимое, буфера, связанного с потоком *stream*, в файл, связанный с этим потоком. Поток остается открытым. Если поток не буферизован, то вызов функции **fflush** не влечет никаких действий.

Если файл открыт на чтение, содержимое буфера очищается.

Функция **fflush** отменяет действие функции **ungetc**, вызванной непосредственно перед ней.

Возвращаемое значение

0, если буфер успешно сброшен.

Значение 0 возвращается и в тех случаях, когда поток не буферизован или открыт только для чтения.

Значение EOF сигнализирует об ошибке.

Замечание. Буфер потока автоматически скидывается, когда он заполняется, когда закрывается поток или когда программа завершилась нормально без закрытия потока.

Смотри также

fclose, flushall, setbuf

_FFREE (MSC)

#include <malloc.h> /*используется только для описания функции*/

void _ffree(ptr);

*char far *ptr;*

Описание

Функция освобождает блок памяти, который располагается вне данного сегмента.

Аргумент *ptr* указывает на блок памяти, динамически выделенный ранее через обращение к функции **_fmalloc**. Число освобождаемых байтов равняется тому числу, которое было определено при выделении блока. После вызова функции **_ffree** освобожденный блок снова доступен для распределения.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

fmalloc, free, malloc, farmalloc

Замечание. Попытка освободить неверный блок (для которого адрес (*ptr*-указатель) не был получен посредством вызова функции **_fmalloc**) может нарушить последующее выделение памяти и привести к ошибке.

FGETC (TC & MSC & ANSI), FGETCHAR (TC & MSC)

#include <stdio.h>

int fgetc(stream);

*FILE *stream;*

int fgetchar(void);

Описание

Функция **fgetc** читает один символ из вводного потока *stream* с текущей позиции и перемещает (внутренний) указатель файла на следующий символ.

Вызов функции **fgetchar()** эквивалентен вызову **fgetc(stdin)**.

Возвращаемое значение

Введенный символ.

Значение **EOF**, если конец файла или ошибка; чтобы отличить конец файла от ошибки, необходимо воспользоваться функцией **feof** или **ferror**.

Замечание. Вызовы **fgetc** и **fgetchar** являются идентичными вызовам **getc** и **getchar**, но являются вызовами функций, а не макроопределений.

Смотри также

fputc, fputchar, getc, getchar

FGETS (TC & MSC & ANSI)

#include <stdio.h>

*char *fgets(string, n, stream);*

*char *string;*

int n;

*FILE *stream;*

Описание

Функция читает строку из входного потока *stream* и помещает ее в строку, адрес которой задается значением параметра *string*. Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки (*'\n'*), который включается в строку, или пока не наступит конец потока, или

пока не будет прочитано (*n-1*) символов. Результат помещается в *string* и заканчивается нулевым символом (*'\0'*). Если *n* равно *1*, то формируется пустая строка.

Возвращаемое значение

Возвращается адрес строки;

значение NULL, если произошла ошибка или достигнут конец файла, эти ситуации можно различить, используя функции **feof** и **ferror**.

Смотри также

fputs, gets, puts

FIEEETOMSBIN, FMSBINTOIEEE (MSC)

#include <math.h>

int fieeetomsbln(src4, dst4);

int fmsbintoieee(src4, dst4);

*float *src4, *dst4;*

Описание

Функция **fieeetomsbin** преобразует число с плавающей точкой из IEEE-формата в Microsoft-двоичный формат.

Функция **fmsbintoieee** преобразует число с плавающей точкой из Microsoft-двоичного формата в IEEE-формат.

Эти функции позволяют Си-программам (которые хранят число с плавающей точкой в IEEE формате) использовать числовые данные из файлов, созданных с помощью программ, разработанных в системе программирования Microsoft BASIC (которые хранят числа с плавающей точкой в Microsoft-двоичном формате), и наоборот.

Аргумент *src4* указывает на значение типа *float*, которое будет преобразовываться. Результат помещается по адресу, задаваемому *dst4*.

Возвращаемое значение

0, если преобразование выполнено успешно.

1, если произошло переполнение.

Смотри также

dieeetomsbin, dmsbintoieee

FILELENGTH (TC & MSC)

#include <io.h>

long filelength(handle);

int handle;

Описание

Функция возвращает длину в байтах файла, связанного с заданным дескриптором *handle*.

Возвращаемое значение

Длина файла в байтах.

Значение -1L, если обнаружена ошибка, при этом переменной *errno* присваивается значение EBADF, что означает неверное значение *handle*.

Смотри также

chsize, fileno, fstat, stat

FILENO (TC & MSC)

#include <stdio.h>

int fileno(stream);

*FILE *stream;*

Описание

Функция возвращает значение дескриптора *handle* (используемого функциями ввода/вывода нижнего уровня), связанного с заданным указателем потока *stream* (используемым функциями ввода/вывода верхнего уровня).

Если несколько дескрипторов связаны с потоком, то функция возвращает значение того из них, который был открыт первым.

Возвращаемое значение

Дескриптор файла. Нет ошибочных кодов возврата.

Результат не определен, если указано неверное значение параметра *stream* (*stream* не является указателем потока).

Замечание. Вызов **fileno** выполняется как макрос.

Смотри также

fdopen, filelength, fopen, freopen

FINDFIRST (TC), _DOS_FINDFIRST (MSC 5.1)

Использование (TC)

#include <dir.h>

#include <dos.h>

int findfirst(pathname, buffer, attrlb);

*char *pathname;*

*struct ffblk *buffer;*

int attrib;

struct ffblk

{

char ff_reserved[21]; /*зарезервировано DOS*/

char ff_attrib; /*найденный атрибут*/

int ff_ftime; /*время*/

int ff_fdate; /*дата*/

long ff_fsize; /*размер*/

char ff_name[13]; /*найденное имя*/

}

Использование (MSC 5.1)

#include <dos.h> /*обратите внимание на различный порядок следования аргументов*/

unsigned _dos_findfirst(pathname, attrib, buffer);

*char *pathname;*

unsigned attrib;

*struct find_t *buffer;*

struct find_t

{

char reserved[21]; /*зарезервировано DOS*/

char attrib; /*найденный атрибут*/

int wr_time; /*время*/

int wr_date; /*дата*/

long size; /*размер*/

char name[13]; /*найденное имя (без пути)*/

}

Описание

Функция начинает поиск файла или каталога на диске (функция находит первый файл, удовлетворяющий заданным условиям, другие файлы, удовлетворяющие тем же условиям, могут быть найдены при помощи функции **findnext**).

Используется системный вызов MS-DOS 0x4E.

Аргумент *pathname* — адрес строки, содержащей необязательный спецификатор устройства и полное имя искомого файла (включающее имя пути). Имя файла может содержать регулярное выражение (символы '?' или '*' с тем же значением, что и в командной строке ОС MS-DOS). Когда заданный файл найден, происходит заполнение структуры, адрес которой задается параметром buffer, информацией о каталоге.

Параметр *attrib* — байт атрибутов файла MS-DOS, используемый при выборе допустимых файлов в поиске.

Параметр *attrib* может принимать одно из следующих значений, определенных в *dos.h*:

TC	MSC 5.1	
A_NORMAL		Обычный файл (чтение/запись)
FA_RDONLY	_A_RDONLY	Только чтение
FA_HIDDEN	_A_HIDDEN	Скрытый файл
FA_SYSTEM	_A_SYSTEM	Системный
FA_LABEL	_A_VOLID	Метка тома
FA_DIREC	_A_SUBDIR	Каталог
FA_ARCH	_A_ARCH	Архив

Структура, заполняемая при вызове функции, содержит информацию, необходимую для продолжения поиска. На каждый последующий вызов функции **findnext** будет возвращаться новое имя файла до тех пор, пока не возникнет такая ситуация, что не останется более ни одного файла, имя которого подходило бы под заданное регулярное выражение.

Функция устанавливает значение регистра DTA (*disk-transfer address*) для адресации структуры, в которой хранится информация о файле. Если потребуется изменить значение DTA, необходимо его сохранить, а затем восстановить, чтобы обеспечить возможность продолжения поиска.

Сохранение и восстановление производится с помощью функций **getdta** и **setdta**.

Примечание. Поиск производится только в пределах каталога. Для организации обхода дерева каталогов надо самостоятельно написать алгоритм, который перебирал бы по очереди все каталоги (найдя их при помощи функций **findfirst** и findnext) и в каждом инициировал бы независимый поиск новым вызовом функции **findfirst**.

Возвращаемое значение

Возвращается значение 0 при успешном поиске.

Если файл не найден или в имени файла встретилась какая-либо ошибка, возвращается значение -1, и глобальной переменной *errno* присиаивается одно из значений:

ENOENT — Имя не найдено

ENMFILE — Файлов нет (только для системы программирования TC)

Переносимость

Только для MS-DOS.

FINDNEXT (TC), _DOS_FINDNEXT (MSC 5.1)

Использование (TC)

#include <dir.h>

#include <dos.h>

*int findnext(struct ffblk *buffer);*

Использование (MSC 5.1)

#include <dos.h>

*unsigned _dos_findnext(struct find_t *buffer);*

Описание

Изучите сначала описание функции **findfirst** (**_dos_findfirst**).

Функция **findnext** (**_dos_findnext**) используется для поиска следующего файла, имя которого подходит под регулярное выражение, заданное как аргумент для последнего вызова функции **findfirst** (**_dos_findfirst**).

Используется системный вызов 0x4F.

Структура, заполняемая при вызове функции, содержит информацию, необходимую для продолжения поиска. На каждый последующий вызов функции **findnext** будет возвращаться новое имя файла до тех пор, пока не возникнет такая ситуация, что не останется более ни одного файла, имя которого подходило бы под заданное регулярное выражение.

Функция устанавливает значение регистра DTA (*disk-transfer address*) для адресации структуры, в которой хранится информация о файле. Если потребуется изменить значение DTA, необходимо его сохранить, а затем восстановить, чтобы обеспечить возможность продолжения поиска.

Сохранение и восстановление производится с помощью функций **getdta** и **setdta**.

Примечание. Поиск производится только в пределах каталога. Для организации обхода дерева каталогов надо самостоятельно написать алгоритм, который перебирал бы по очереди все каталоги (найдя их при помощи функции **findfirst** и **findnext**) и в каждом инициировал бы независимый поиск новым вызовом функции **findfirst**.

Возвращаемое значение

Возвращается значение 0 при успешном поиске.

Если файл не найден или в имени файла встретилась какая-либо ошибка, возвращается значение -1, и глобальной переменной *errno* присваивается одно из значений:

ENOENT — Имя не найдено

ENMFILE — Файлов нет (только для системы программирования TC)

Переносимость

Только для MS-DOS.

FLOOR (TC & MSC & ANSI)

#include <math.h>

double floor(x);

double x;

Описание

Функция возвращает значение с плавающей точкой, которое представляет наибольшее целое, меньшее или равное *x*.

Возвращаемое значение

Число с плавающей точкой.

Нет ошибочных кодов возврата.

Смотри также

ceil, fmod

FLUSHALL (TC & MSC)

#include <stdio.h> /*используется только для описания функции*/

int flushall();

Описание

Функция записывает содержимое всех буферов, связанных с открытыми выводными потоками, в файлы, связанные с этими потоками. Все потоки остаются открытыми после **flushall**.

Функция обнуляет также все буфера, связанные с файлами, открытыми на чтение. Для буферизованных потоков вызов функции **flushall** отменяет действие функции **ungetc**, вызванной непосредственно перед этим.

Возвращаемое значение

Количество открытых потоков (вводных и выводных).

Нет ошибочных кодов возврата.

Замечание. Буфер потока автоматически скидывается, когда он заполняется, когда закрывается поток или когда программа завершилась нормально без закрытия потока.

Смотри также

fflush

_FMALLOC (MSC)

#include <malloc.h> /*используется только для описания функции*/

*char far *_fmalloc(size);*

unsigned size;

Описание

Функция используется для динамического выделения блока оперативной памяти как минимум *size* байтов вне заданного сегмента. (Размер блока может быть больше *size* байтов в результате выравнивания.)

Возвращаемое значение

Далекий (*far*) указатель на *char*. Область памяти, на которую указывает возвращаемое значение, является выровненной для объектов любого типа. Для получения указателя типа, отличного от *char*, необходимо использовать операцию явного преобразования типа. Если нет достаточной памяти вне текущего сегмента, будет попытка получить память, используя текущий сегмент. Если и в этом случае памяти окажется недостаточно, будет возвращено значение NULL.

Смотри также

_ffree, _fmsize, malloc, realloc, farmalloc

FMOD (TC & MSC & ANSI)

#include <math.h> /*используется только для описания функции*/

double fmod(x, y);

double x;

double y;

Описание

Функция вычисляет остаток от деления *x* на *y*, такой, что *x=iy+f*, где *i* — целое, *f* имеет такой же знак, как *x*, и абсолютное значение *x* меньше абсолютного значения *y*.

Возвращаемое значение

Остаток в виде числа с плавающей точкой.

Значение 0.0, если *y* имеет нулевое значение или деление *x* на *y* приводит к переполнению.

Смотри также

ceil, fabs, floor

_FMSIZE (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned _fmsize(ptr);

*char far *ptr;*

Описание

Функция возвращает размер в байтах блока памяти, выделенного по вызову функции **_fmalloc**.

Возвращаемое значение

Размер в байтах, как беззнаковое целое.

Смотри также

_ffree, _fmalloc, malloc, _msize, _nfree, _nmalloc, _nmsize

FNMERGE (TC)

#include <dir.h>

void fnmergef path, drive, dir, name, ext);

*char *path;*

*char *drive;*

*char *dir;*

*char *name;*

*char *ext;*

Описание

Функция **fnmerge** создает полное имя файла из отдельных компонент: формируется строка "X:\DIR\SUBDIR\NAME.EXT" и записывается по адресу, задаваемому аргументом *path*, при этом

составляющие части определяются следующим образом:

"X:" задается аргументом *drive*;

"\DIR\SUBDIR\" — задается аргументом *dir*;

"NAME.EXT" — задается аргументами *name* и *ext*.

Функция **fnmerge** полагает, что *path* содержит адрес буфера, в котором зарезервировано достаточно места для полного имени файла. Максимальная длина полного имени определяется константой MAXPATH, заданной в файле *dir.h*.

Строка *drive* включает в себя двоеточие ("C:", "A:" и т.д.).

Строка *dir* содержит начальный и заключительный символ '\\' (например, "*turboc**include*\", "*source*\" и т.д.).

Строка *ext* содержит точку, предшествующую расширению (".c", ".exe" и т.д.).

Функции **fnmerge** и **fnsplit** взаимно дополняют друг друга: если вы разбили данное полное имя *path* с помощью **fnsplit**, объединение полученных компонент с помощью **fnmerge** снова даст *path*.

Возвращаемое значение

Функция **fnmerge** не возвращает значения.

Переносимость

Только для MS-DOS.

FNSPLIT (TC)

#include <dir.h>

int fnsplit(path, drive, dir, name, ext);

*char *path;*

*char *drive;*

*char *dir;*

*char *name;*

*char *ext;*

Описание

Функция **fnsplit** разбивает полное имя файла (задаваемое аргументом *path*), представляемое строкой вида "X:\DIR\SUBDIR\NAME.EXT", на 4 компоненты.

Затем они помещаются в строки, адреса которых задаются аргументами *drive*, *dir*, *name*, *ext*. Каждая компонента обязательна, но может быть NULL-ссылкой.

Максимальные размеры этих строк представлены константами MAXDRIVE, MAXDIR, MAXPATH, MAXNAME, MAXEXT, описанными в файле *dir.h*, и каждый размер учитывает запас конечного, нулевого байта ('\0'). Значения констант задаются следующие:

Константа	***Максимальный размер***	***Строка***
MAXPATH	80	path
MAXDRIVE	3	drive, включая двоеточие (:)

```
MAXDIR        66                    dir, включая начальную и последнюю (\)
MAXFILE       9                     name
MAXEXT        5                     ext, включая начальную точку (.)
```

Функция **fnsplit** полагает, что достаточно места для хранения каждой ненулевой компоненты.

При разбитии полного имени *path* на части функция **fnsplit** трактует знаки пунктуации следующим образом:

Строка *drive* включает в себя двоеточие ("C:", "A:" и т.д.).

Строка *dir* содержит начальный и заключительный символ '\\' (например,"*turboc**include*\", "*source*\" и т.д.).

Строка *ext* содержит точку, предшествующую расширению (".c", ".exe" и т. д.).

Функции **fnmerge** и **fnsplit** взаимно дополняют друг друга: если вы разбили данное полное имя *path* с помощью **fnsplit**, объединение полученных компонент с помощью **fnmerge** снова даст *path*.

Возвращаемое значение

Функция **fnsplit** возвращает целое значение, составленное из 5 флагов, определенных в *dir.h*, показывающее, какие компоненты полного имени были представлены в *path*. Флаги и представляемые ими компоненты:

EXTENSION расширение

FILENAME имя файла

DIRECTORY каталог (и, возможно, подкаталоги)

DRIVE спецификация устройства (см. *dir.h*)

WILDCARD регулярное выражение (символы ? и *)

Переносимость

Только для MS-DOS.

FOPEN (TC & MSC & ANSI)

#include <stdio.h>

*FILE *fopen(pathname, type);*

*char *pathname;*

*char *type;*

Описание

Функция открывает файл, имя которого задается аргументом *pathname*, и связывает с ним поток (для выполнения высокоуровневого ввода/вывода).

Аргумент *type* указывает на строку символов, определяющих способ доступа к файлу, и может принимать следующие значения:

"*r*" Открыть для чтения (файл должен существовать);

"*w*" Открыть пустой файл для записи; если файл существует, то его содержимое теряется;

"*a*" Открыть для записи в конец файла (добавления); файл создается, если он не существует;

"*r+*" Открыть для чтения и записи (файл должен существовать) ;

"*w+*" Открыть пустой файл для чтения и записи; если файл существует, его содержание теряется;

"*a+*" Открыть для чтения и добавления; файл создается, если он не существует.

Замечание. Будьте осторожны при использовании "*w*" и "*w+*", т.к. они могут запортить существующие файлы.

Определяемый тип должен быть совместим с правами доступа к файлу.

Когда файл открыт в режиме "*a*" или "*a+*", данные записываются в конец файла. Хотя указатель файла может быть перемещен, используя функции **fseek** или **rewind**, он всегда будет перемещаться обратно на конец файла при записи данных. Таким образом, существующие данные не могут быть затерты.

Когда определен один из режимов "*r+*","*w+*" или "*a+*", разрешено чтение и запись (т.е. файл открыт для изменения ("*update*")). Однако при переключении с чтения на запись и наоборот должно осуществляться "ручное" позиционирование указателя по файлу с помощью функций **fseek**, **rewind**, **fsetpos**.

Для того чтобы указать, что файл должен быть открыт в текстовом режиме, необходимо добавить символ *'t'* в строку type, если в двоичном — *'b'*.

В текстовом режиме последовательности символов *<CR><LF>* преобразуются при чтении в символ *<LF>*. Символы *<LF>* преобразуются в последовательности символов *<CR><LF>* при выводе. Кроме того, *<CTRL/Z>* интерпретируется как символ конца файла при вводе. Для файлов, открываемых на чтение или чтение/запись, по возможности происходит проверка и удаление символов *<CTRL/Z>* (это необходимо для корректной работы функции **fseek**).

В двоичном режиме перечисленные выше преобразования не производятся.

Если не указан ни символ *'b'*, ни символ *'t'*, принимается режим, задаваемый по умолчанию глобальной переменной *_fmode*.

Возвращаемое значение

Указатель на открытый поток.

Значение NULL, если обнаружена ошибка.

Смотри также

fclose, fcloseall, fdopen, ferror, fileno, freopen, open, setmode

_FPRESET (TC & MSC)

#include <float.h>

void _fpreset();

Описание

Функция инициализирует пакет функций арифметики с плавающей точкой.

Эта функция обычно используется вместе с функциями **signal**, **system** или **exec...** и **spawn...**

операций с плавающей точкой (SIGFPE) с помощью функции **signal**, то эти сигналы можно обработать с помощью **_fpreset** и вернуться из подпрограммы обработки посредством функции **longjmp**.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, signal, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe

FPRINTF (TC & MSC & ANSI)

#include <stdio.h>

int fprintf(stream, format_string[, argument...]);

*FILE *stream;*

char format_string;

Описание

Функция выполняет форматные преобразования для данных, предназначенных для вывода, и печатает последовательности символов и значений в выводной поток *stream*.

Каждый аргумент *argument* (если их несколько) преобразуется и выводится согласно формату преобразования, определяемому в строке, задаваемой аргументом *format_string*.

Строка *format_string* имеет определенный вид, и существуют соглашения об обозначении формата различных элементов вывода. Эти соглашения те же, что и для функции **printf**.

Смотрите описание функции **printf**.

Возвращаемое значение

Количество выведенных символов (завершающий нулевой символ не учитывается).

(Только для системы программирования TC) — при ошибке возвращается значение EOF.

Смотри также

cprintf, fscanf, printf, sprintf, fwrite

FPUTC (TC & MSC & ANSI), FPUTCHAR (TC & MSC)

#include <stdio.h>

int fputc(c, stream);

int c;

*FILE *stream;*

int fputchar(c);

int c;

Описание

Функция **fputc** записывает одиночный символ **c** в поток *stream* в текущую позицию. Вызов **fputchar(c)** эквивалентен вызову **fputc(c, stdout)**.

Возвращаемое значение

Записанный символ.

Значение EOF, если встретился конец файла или ошибка, эти ситуации можно различить, используя функцию **feof** или **ferror**.

Смотри также

fgetc, fgetchar, putc, putchar

FPUTS (TC & MSC & ANSI)

#include <stdio.h>

int fputs(string, stream);

*char *string;*

*FILE *stream;*

Описание

Функция копирует строку **string** в поток *stream*, с текущей позиции. Завершающий нулевой символ ('\0') не копируется.

Возвращаемое значение (TC & MSC 4.0)

Последний записанный символ.

Значение 0, если строка *string* пустая;

значение EOF, если произошла ошибка.

Возвращаемое значение (MSC 5.1)

Возвращается значение 0, если все в порядке;

при неудаче возвращает ненулевое значение.

Смотри также

fgets, gets, puts

FP_OFF, FP_SEG (TC & MSC)

#include <dos.h>

unsigned FP_OFF(longptr);

unsigned FP_SEG(longptr);

*char far *longptr;*

Описание

FP_OFF и FP_SEG — макросы — возвращают смещение и адрес сегмента указателя *longptr* соответственно.

В моделях памяти *small* и *medium* эти макросы работают только в том случае, если указатель *far* указывает по сегменту данных, используемому по умолчанию.

Возвращаемое значение

Функция FP_OFF возвращает беззнаковое целое, представляющее собой смещение внутри сегмента.

Функция FP_SEG возвращает беззнаковое целое, представляющее собой адрес сегмента.

Смотри также

segread

FREAD (TC & MSC & ANSI)

#include <stdio.h>

int fread(buffer, size, count, stream);

*void *buffer;*

size_t size;

size_t count;

*FILE *stream;*

Описание

Функция читает *count* элементов длины *size* из входного потока *stream* и помещает их в заданный массив *buffer*. Указатель файла, связанный с потоком *stream*, увеличивается на число действительно прочитанных байтов.

Форматных преобразований данных (как для функции **fscanf**) не производится. Символы перевода строки ('\n') специально (как для функции **fgets**) не обрабатываются.

Для работы с файлом, содержащим двоичную информацию, Файл необходимо открыть в двоичном режиме (смотри описание функции **fopen**).

Возвращаемое значение

Количество действительно прочитанных элементов, которое может быть меньше, чем *count*, если встретилась ошибка или конец файла раньше, чем было прочитано *count* элементов.

Смотри также

fwrite, read

FREE (TC & MSC & ANSI)

#include <stdlib.h> /*используется в СП TC, MSC*/

#include <malloc.h> /*используется в СП MSC*/

#include <alloc.h> /*используется в СП TC*/

void free (ptr);

*void *ptr;*

Описание

Функция освобождает блок памяти, аргумент *ptr* указывает на блох памяти, который был ранее получен по вызову функции **calloc**, **malloc** или **realloc**.

Число освобождающихся байтов — это число байтов, которые были выделены при получении блока памяти. После вызова освободившаяся область памяти снова может быть выделена по запросу.

(Для системы программирования MSC 5.1) — NULL-аргумент игнорируется.

Возвращаемое значение

Нет кодов возврата.

Замечание. Если для **free** задано неправильное значение *ptr* (значение указателя, которое не было получено посредством вызова функции **calloc**, **malloc**, **realloc**), это может привести к труднолокализуемым ошибкам.

Смотри также

calloc, malloc, realloc, hfree, _ffree, _nfree, farfree

_FREECT (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned int _freect(stze);

size_t size;

Описание

Функция позволяет определить, сколько памяти доступно для динамического выделения, возвращается приблизительное число — сколько раз можно вызвать **malloc** для выделения области памяти заданного размера в сегменте данных, к которому происходит обращение по умолчанию.

Возвращаемое значение

Число возможных обращений как беззнаковое целое.

Смотри также

calloc, _expand, malloc, _memavl, _msize, realloc, coreleft

FREEMEM (TC), _DOS_FREEMEM (MSC 5.1)

Использование (TC)

#include <dos.h>

int freemem(unsigned seg);

Использование (MSC 5.1)

#include <dos.h>

unsigned _dos_freemem(unsigned seg);

Описание

Функция **freemem** освобождает блок памяти, выделенный предыдущим вызовом *allocmem*. Аргумент *seg* — это адрес сегмента данного блока. Освобожденная память больше недоступна прикладным процессам.

Используется системный вызов MS-DOS 0x49.

Возвращаемое значение

Функция **freemem** в случае нормального завершения возвращает значение 0.

В случае ошибки возвращается значение -1, а переменной *errno* присваивается значение ENOMEM.

Причиной ошибки может быть обращение с адресом сегмента, не полученным через *allocmem* или *setblock*, а также разрушение заголовка одной из областей.

Переносимость

Только для MS-DOS.

Смотри также

coreleft, allocmem, malloc, setblock, biosmem

FREOPEN (TC & MSC & ANSI)

#include <stdio.h>

*FILE *freopen(pathname, type, stream);*

*char *pathname;*

*char *type;*

*FILE *stream;*

Описание

Функция закрывает файл, связанный в текущий момент с потоком *stream*, и переназначает поток *stream* на файл, определенный в *pathname*.

Функция **freopen** обычно используется, чтобы связать потоки *stdin*, *stdout*, *stderr*, *stdaux* и *stdprn* с файлами, задаваемыми пользователем.

Новый файл, связанный с потоком *stream*, открывается с типом доступа, определяемым *type*, см. описание функции **fopen**.

Возвращаемое значение

Указатель потока для нового открытого файла.

Значение NULL, если встретилась ошибка; при этом первоначально открытый (связанный с потоком *stream*) файл закрывается.

Смотри также

fclose, fcoseall, fdopen, fileno, fopen, open, setmode

FREXP (TC & MSC & ANSI)

#include <math.h>

double frexp(x, expptr);

double x;

*int *expptr;*

Описание

Функция разбивает число с плавающей точкой, задаваемое параметром *x*, на мантиссу *m* и экспоненту *n* таким образом, что абсолютное значение *m* больше или равно 0.5 и *n* меньше 1.0 и *x=m*2*. Целочисленное значение экспоненты *n* присваивается переменной, адрес которой задается параметром *expptr*.

Возвращаемое значение

Значение мантиссы *m*.

Значение 0.0 для мантиссы (при этом экспонента равна 0), если значение параметра *x* равно 0.

Нет ошибочных кодов возврата.

Смотри также

ldexp, modf

FSCANF (TC & MSC & ANSI)

#include <stdio.h>

int fscanf (stream, format_strtng[, argument...]);

*FILE *stream;*

*char *format_string;*

Описание

Функция читает данные из указанного входного потока *stream*, выполняет форматные преобразования и полученные значения записывает в переменные, адреса которых задаются параметрами *argument...*

Каждый из параметров *argument* должен быть указателем на переменную типа, соответствующего типу, заданному очередным элементом описания поля формата (такие элементы начинаются с символа '*%*') в строке, адрес которой задается аргументом *format_string*.

Format_string определяет тип вводимых значений и имеет тот же вид, что и для функции **scanf**; смотри описание *format_string* в описании функции **scanf**.

Возвращаемое значение

Количество успешно введенных, преобразованных и присвоенных элементов ввода. При этом не учитываются элементы, которые были успешно введены, но не присвоены.

Возвращается значение FOF, если уже достигнут конец файла.

Возвращается значение 0, если нет элементов, которые были бы присвоены.

Смотри также

cscanf, fprintf, scanf, sscanf

FSEEK (TC & MCS & ANSI)

#include <stdio.h>

int fseek(stream, offset, origin);

*FILE *stream;*

long offset;

int origin;

Описание

Функция перемещает (внутренний) указатель файла, связанного с потоком *stream*, на новое место в файле, которое вычисляется по смещению *offset* и указанию направления отсчета *origin*.

Следующая операция ввода/вывода с указанным потоком *stream* будет выполнена, начиная с той позиции, на которую произведено перемещение. В потоке, открытом для изменения, следующей операцией может быть чтение или запись.

Аргумент *origin* должен быть одной из следующих констант, определенных в *stdio.h*:

SEEK_SET (значение 0) Начало файла

SEEK_CUR (значение 1) Текущая позиция указателя файла

SEEK_END (значение 2) Конец файла

Для текстового режима работы с файлом значение аргумента *offset* должно быть получено с помощью функции *ftell()* или равняться нулю.

Функция может быть использована для перемещения указателя в любое место файла. Указатель может быть перемещен за конец файла. Однако попытка переместить указатель за начало файла приведет к ошибке.

При перемещении указателя по файлу признак наличия для файла символа, возвращенного в поток через вызов функции *ungetc()*, обнуляется, т.е. будет забыто, что был вызов функции *ungetc()*.

Возвращаемое значение

Значение 0, если указатель успешно перемещен.

Ненулевое значение, если произошла ошибка.

Неопределенное значение для таких устройств, как терминал и принтер.

Смотри также

ftell, lseek, rewind

FSTAT (TC & MSC)

#include <sys\types.h>

#include <sys\stat.h>

int fstat(handle, buffer);

int handle;

*struct stat *buffer;*

Описание

Функция позволяет получить информацию об открытых файлах, связанных с заданным дескриптором *handle* (понятие ввода/вывода нижнего уровня).

Информация помещается в структуру, адрес которой содержит указатель *buffer*.

Структура с типом *stat* определена в файле *sys\stat.h*:

```
struct stat
{
short st_dev;
short st_ino;
short st_mode;
short st_nlink;
int st_uid; /*идентификатор пользователя (не используется)*/
int st_gid; /*идентификатор группы (не используется)*/
short st_rdev;
long st_size;
long st_atime;
long st_mtime;
long st_ctime;
};
```

Значение полей структуры:

st_mode битовая строка, содержащая информацию о режимах работы с открытыми файлами. Следующие константы, определенные в файле *sys\stat.h*, могут быть использованы для выделения битов:

```
#define S_IFMT 0xF000 /*маска выделения признаков типа файла*/
#define S_IFDIR 0x4000 /*каталог*/
#define S_IFIFO 0x1000 /*устройство типа очередь (FIFO)*/
#define S_IFCHR 0x2000 /*символьное устройство*/
#define S_IFBLK 0x3000 /*блочное устройство*/
#define S_IFREG 0x8000 /*то же 0x0000, обыкновенный файл*/
```

#define S_IREAD 0x0100 /*доступен по чтению*/

#define S_IWRITE 0x0080 /*доступен по записи*/

#define S_IEXEC 0x0040 /*доступен для выполнения*/

Бит S_IFCHR устанавливается, если файл, с которым связан дескриптор *handle*, открыт не на устройстве произвольного доступа.

Бит S_IFREG устанавливается, если с дескриптором *handle* связан обыкновенный файл.

Биты S_IREAD/S_IWRITE чтения/записи устанавливаются согласно типу доступа к файлу:

st_dev — номер дисковода, где расположен файл, или *handle* в случае устройства;

st_rdev — то же, что и *st_dev*;

st_nlink — этому элементу всегда присваивается значение 1;

st_size — размер открытого файла в байтах;

st_atime — время последней модификации файла;

st_mtime — время последней модификации файла (копия *st_atime*);

st_ctime — время последней модификации файла (копия *st_atime*).

Возвращаемое значение

Значение 0, если информация об открытом файле получена.

Значение -1, если задан неправильный дескриптор *handle*; при этом переменной *errno* присваивается значение EBADF.

Замечание. Если заданный *handle* относится к устройству, то поля размера и времени в структуре *stat* не имеют смысла.

Смотри также

access, chmod, filelength, stat

FTELL (TC & MSC & ANSI)

#include <stdio.h>

long ftell(stream);

*FILE *stream;*

Описание

Функция позволяет получить текущую позицию (внутреннего) указателя файла, связанного с потоком *stream*. Позиция задается как смещение (значение типа *long int*) относительно начала файла.

Возвращаемое значение

Возвращается смещение по файлу.

Значение -1L сигнализирует об ошибке.

Для устройств терминал и принтер, а также когда файл не открыт, возвращаемое значение не определено.

Смотри также

fseek, lseek, tell

FTIME (TC 2.0 & MSC)

#include <sys\types.h>

#include <sys\timeb.h>

void ftime(timeptr);

*struct timeb *timeptr;*

Описание

Функция получает текущее время и записывает его в структуру, адрес которой задается значением параметра *timeptr*.

Структура *timeb* определяется в файле *sys\timeb.h*.

struct timeb

{

long time;

short millitm;

short timezone;

short dstflag;

};

В поле структуры *time* задается время в секундах, начиная с момента 00:00:00 по Гринвичу (Greenwich Mean Time) 1 января 1970 г.

В поле структуры *millitm* задается дробная часть этого же времени в тысячных долях секунды (в миллисекундах).

В поле *timezone* задается различие в минутах, при движении в западном направлении, между временем по Гринвичу (Greenwich Mean Time) и местным временем.

Значение поля *timezone* устанавливается по значению глобальной переменной *timezone* (смотри описание функции **tzset**).

Если значение поля *dstflag* ненулевое, то время двенадцатичасовой способ представления времени (Daylight Saving Time) используется для данного региона, значение определяется из значения глобальной переменной *daylight* (смотри описание функции **tzset**).

Возвращаемое значение

Функция не возвращает значения.

Смотри также

asctime, ctime, gmtime, localtime, time, tzset

FWRITE (TC & MSC & ANSI)

#include <stdio.h>

int fwrite(buffer, size, count, stream);

*char *buffer;*

int size;

int count;

*FILE *stream;*

Описание

Функция дописывает *count* записей по *size* байтов из области *buffer* в выходной поток *stream* (высокоуровневый ввод/вывод).

Указатель файла (имеется в виду внутренний указатель), связанный с потоком *stream*, увеличивается на число записанных байтов.

Форматных преобразований данных (как для функции **fprintf**) не производится. Символы перевода строки ('*n*') специально (как для функции **fputs**) не обрабатываются.

Для работы с файлом, содержащим двоичную информацию, файл необходимо открыть в двоичном режиме (смотри описание функции **fopen**).

Возвращаемое значение

Количество реально помещенных в файл записей; это число может быть меньше, чем значение *count*, если имела место ошибка.

Смотри также

fread, write

GCVT (TC & MSC)

#include <stdlib.h>

char gcvt(value, ndec, buffer);

double value;

int ndec;

*char *buffer;*

Описание

Функция преобразует число с плавающей точкой (значение параметра *value*) в строку символов и записывает эту строку по адресу, задаваемому параметром *buffer*.

Буфер должен быть достаточно большим, чтобы принять преобразованное число с символом конца строки ('\0'), который добавляется автоматически.

Функция попытается перевести *ndec* значимых цифр в F-формате вывода языка программирования

FORTRAN-IV. Если это не удалось, переводится *ndec* значимых цифр в E-формат вывода языка программирования FORTRAN-IV. Незначащие нули подавляются при преобразовании.

Возвращаемое значение

Указатель на строку символов, возвращаемое значение равно значению параметра *buffer*.

Нет ошибочных кодов возврата.

Смотри также

atof, atoi, atol, ecvt, fcvt

GENINTERRUPT (TC)

#include <dos.h>

void geninterrupt(int intr_num);

Описание

Функция **geninterrupt** инициирует внутреннее прерывание программы, номер прерывания задается параметром *intr_num*.

Функция реализуется как встроенная (в текст программы вставляется машинная команда INT).

Возвращаемое значение

Возвращает значение, зависящее от вызванного прерывания.

Смотри также

getvect, bdos, bdosptr, int86, int86x, iritdos, intdosx intr

GETC, GETCHAR (TC & MSC & ANSI)

#include <stdio.h>

int getc(stream);

*FILE *stream;*

int getchar();

Описание

Функция **getc** читает символ из входного потока *stream* и передвигает связанный с потоком (внутренний) указатель файла на следующий символ.

Функция **getchar** реализуется как макроопределение *getc(stdin)*.

Вызовы **getc** и **getchar** реализуются через макроопределения.

Возвращаемое значение

Оба макроса возвращают прочитанный символ.

Возвращается значение EOF, если встретилась ошибка или достигнут конец файла, которые можно распознать, используя функции **feof** или **ferror**.

Замечание. **getc** и **getchar** идентичны **fgetc** и **fgetchar**, но являются макросами, а не функциями.

Смотри также

fgetc, fgetchar, getch, getche, putc, putchar, ungetc

GETCBRK (TC)

#include <dos.h>

int getcbrk(void);

Описание

Функция возвращает текущую установленную реакцию на ввод с клавиатуры комбинации символов (*control-break*).

Функция использует системный вызов 0x33 MS-DOS.

Возвращаемое значение

Если функция возвращает значение 0, то проверка выключена и программа по *control-break* не прерывается, если 1 — включена.

Замечание. Программа будет прервана в любом случае, если комбинация клавиш *control-break* будет введена во время ожидания программой ввода с терминала.

Смотри также

setcbrk

GETCH (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

int getch();

Описание

Функция читает без эхо-печати (без отражения на экране) одиночный символ с консольного (управляющего) терминала.

Функция **getch** использует входной поток *stdin*.

Возвращаемое значение

Прочитанный символ.

Смотри также

cgets, getche, getchar

GETCHE (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

int getche();

Описание

Функция читает одиночный символ с консольного терминала, с эхо-отображением введенного символа на экране с текущей позиции непосредственно через видеоадаптер или через BIOS.

Если набрана комбинация <*control/break*> CONTROLC, система выполняет прерывание INT23H (CONTROL-C *exit*).

Возвращаемое значение

Прочитанный символ.

Нет ошибочных кодов возврата.

Смотри также

cgetc, getch, getchar

GETCURDIR (TC)

#include <dir.h>

*int getcurdir(int drive, char *direc);*

Описание

Функция читает имя текущего рабочего каталога на указанном параметром *drive* устройстве (дисководе).

Нумерация устройств с 0 (0 — устройство по умолчанию, 1="A", 2="B", 3="C" и т.д.).

Имя каталога записывается в символьный массив, адрес которого задается значением параметра *direc*.

Максимальная длина имени каталога определяется константой MAXDIR, заданной в файле *dir.h*.

Возвращаемое значение

Функция возвращает -1 при ошибке и 0 в противном случае.

GETCWD (TC & MSC)

#include <dir.h> /*используется в СП TC*/

#include <direct.h> /*используется в СП MSC*/

*char *getcwd(buf, n);*

*char *buf;*

int n;

Описание

Функция читает полное имя текущего рабочего каталога (включая имя устройства) длины не более *n* символов, сохраняя его в символьном массиве, адрес которого задается значением параметра *buf*.

Если для записи полного имени пути требуется строка длиннее *n* байтов, фиксируется ошибка.

Если первоначально значение *buf* равно NULL, буфер длины *n* байтов будет выделен с помощью *malloc*. Этот буфер может быть позже освобожден с помощью функции *free*, передав ей как параметр возвращаемое значение функции **getcwd** (указатель на полученный буфер).

Возвращаемое значение

Функция возвращает адрес буфера, в который записано имя массива.

В случае ошибки возвращается значение NULL, при этом переменной *errno* присваивается одно из следующих значений:

ENOMEM Недостаточно памяти, чтобы получить *n* байтов (когда значение NULL задано в качестве *buf*).

ERANGE Имя пути превышает *n* символов.

Смотри также

chdir, mkdir, rmdir

GETDATE (TC)

#include <dos.h>

*void getdate(struct date *dateblk);*

Описание

Функция **getdate** получает текущую дату и помещает ее в структуру, адрес которой задается значением параметра *dateblk*.

Структура *date* описана в файле *dos.h* и имеет следующий вид:

struct date

{

int da_year; /*текущий год*/

char da_day; /*месяц года*/

char da_mon; /*месяц (январь = 1)*/

}

Возвращаемое значение

Функция не возвращает значения.

Смотри также

setdate, gettime, gettime

GETDFREE (TC)

#include <dos.h>

*void getdfree(int drive, struct dfree *dfreep);*

Описание

Функция получает информацию о наличии свободного пространства на диске, задаваемом значением параметра *drive* (0=по умолчанию, 1="A", 2="B" и т.д.). Информация записывается в структуру, адрес которой задается значением параметра *dfreep*.

Структура описана в файле *dos.h* и имеет следующий вид:

struct dfree

{

unsigned df_avail; /*число незанятых кластеров*/

unsigned df_total; /*общее число кластеров*/

unsigned df_bsec /*число байт в секторе*/

unsigned df_sclus /*число секторов в кластере*/

}

Возвращаемое значение

Функция не возвращает значения.

Если произошла ошибка, элементу *df_sclus* присваивается значение -1.

Смотри также

getfat, getdisk

GETDISK (TC)

#include <dir.h>

int getdisk(void);

Описание

Функция **getdisk** возвращает номер текущего диска (0="A", 1="B", 2="C" и т.д.). Используется системный вызов 0x19 MS-DOS.

Возвращаемое значение

Номер диска.

Смотри также

setdisk, getcwd, getcurdir

GETDTA (TC)

#include <dos.h>

*char far *getdta(void);*

Описание

Функция **getdta** возвращает текущее значение адреса области передачи данных диска (DTA) (подробности смотри в руководстве "Technical Reference Manual").

Используется системный вызов MS-DOS 0x2F.

Примечание. Корректно работает с моделями памяти: *compact*, *large* и *huge*. В малых моделях памяти работает корректно только если не вызываются подпрограммы, написанные на ассемблере (полагается, что DTA-адрес лежит в пределах текущего сегмента данных, подпрограммы на других языках могут нарушить это соглашение).

Возвращаемое значение

Адрес передачи для последней выполненной дисковой операции.

Смотри также

setdta, findfirst

GETENV (TC & MSC & ANSI)

#include <stdlib.h> /*используется только для описания функции*/

*char *getenv(varname);*

*char *varname;*

Описание

Функция организует поиск имени *varname* в списке переменных окружения.

Переменные окружения определяют среду, в которой выполняется процесс (например, переменная окружения PATH определяет, в каких каталогах по умолчанию ведется поиск файлов для выполнения).

Возвращаемое значение

Указатель по таблице параметров окружения на строку, содержащую значение переменной *varname*.

Для системы программирования TC версии 1.5 возвращается значение NULL, если заданная переменная не определена.

Для системы программирования TC версии 2.0 возвращается указатель на пустую строку, если заданная переменная не определена.

Замечание 1. Таблицу переменных окружения нельзя изменить непосредственно. Чтобы сделать это, используйте функцию **putenv**. Чтобы модифицировать строку, указатель на которую возвращает функция **getenv**, без изменений в таблице окружения, используйте функцию **strdup** или **strcpy**, чтобы предварительно скопировать строку на новое место.

Замечание 2. Функции **getenv** и **putenv** используют глобальную переменную *environ*, чтобы получить доступ к таблице окружений.

Замечание 3. Имя переменной, которое подается как аргумент функции **getenv**, может быть задано как в верхнем регистре (прописными буквами), так и в нижнем (строчными).

Смотри также

putenv

GETFAT, GETFATD (TC)

#include <dos.h>

void getfat(drive, fatblkp);

int drive;

*struct fatinfo *fatblkp;*

void getfatd(fatblkp);

*struct fatinfo *fatblkp;*

Описание

Функции **getfat** и **getfatd** позволяют получить информацию из таблицы размещения файлов (File Allocation Table — FAT).

Параметр *drive* для функции **getfat** задает номер устройства: (0=устройство по умолчанию, 1="A", 2="B", 3="C" и т.д.).

Функция **getfatd** получает информацию об устройстве по умолчанию.

Обе функции записывают информацию о диске в структуру, адрес которой задается параметром *fatblkp*.

Структура описывается в файле *dos.h* и имеет следующий вид:

struct fatinfo

{

char fi_sclus; /*число секторов в кластере*/

char fi_fatid; /*FAT-идентификатор*/

char fi_ndus; /*число кластеров*/

int fi_bysec; /*число байт в секторе*/

}

Смотри также

getdfree

GETFTIME (TC)

#include <dos.h>

*int getfttme (int handle, struct ftime *fttmep);*

Описание

Функция **getftime** позволяет получить время и дату создания файла, открытого с помощью функции **open**, файл идентифицируется по значению связанного с ним дескриптора *handle*.

Результат записывается в структуру, адрес которой определяется значением параметра *ftimep*.

Структура *ftime* описывается в файле *dos.h* и имеет следующий вид:

struct ftime

{

unsigned ft_sec:5; /*деленное на два число секунд*/

unsigned ft_min:6; /*минуты*/

unsigned ft_hour:5; /*часы*/

unsigned ft_day:5; /*число*/

unsigned ft_month:4; /*месяц*/

unsigned ft_year:7; /*год—1980*/

};

Возвращаемое значение

Функция возвращает значение 0 при успешном завершении операции и число меньше 0 при ошибке.

Смотри также

setftime

GETPASS (TC)

#include <conio.h>

*char *getpass(char *prompt);*

Описание

Функция **getpass** используется для ввода с консольного терминала пароля, который пользователь должен набрать в ответ на выводимое на экран приглашение, задаваемое строкой *prompt* с завершающим нулем ('\0').

При вводе пароля отключается эхо-печать вводимых с терминала символов.

Возвращаемое значение

Возвращается указатель на строку с завершающим нулем, содержащую не более восьми первых символов вводимого пароля.

Пароль хранится в статическом буфере (статическом символьном массиве) функции и переписывается при каждом обращении к ней.

Смотри также

getch

GETPID (MSC)

#include <process.h> /*используется только для описания функции*/

int getpid();

Описание

Функция возвращает целое число — номер текущего (выполняемого) процесса, по этому номеру однозначно определяется вызванный процесс.

Возвращаемое значение

Возвращается номер процесса.

Нет ошибочных кодов возврата.

Смотри также

mktemp

GETPSP (TC)

#include <dos.h>

unsigned getpsp(void);

Описание

Функция используется для получения сегментного префикса текущего программного адреса для текущего выполняемого процесса, через системное прерывание 0x62 операционной системы MS-DOS. За подробностями обращайтесь к справочнику "Technical Reference".

Работает только в ОС MS-DOS версий 3.0 и более поздних.

Возвращаемое значение

Возвращается сегментный адрес.

Смотри также

getenv

GETS (TC & MSC & ANSI)

#include <stdio.h>

*char *gets(buffer);*

*char *buffer;*

Описание

Функция **gets** читает строку из стандартного потока ввода *stdin* (высокоуровневый ввод/вывод) и помещает считанную строку по адресу, задаваемому параметром *buffer*.

В строку включаются все символы до первого встретившегося символа новой строки ('*n*'), не включая его. Строка-результат заканчивается нулевым символом ('*0*').

Возвращаемое значение

Значение параметра (адрес буфера), если строка считана удачно.

Значение NULL, если произошла ошибка или достигнут конец файла, эти ситуации можно различить, используя функции **feof** и **ferror**.

Смотри также

fgets, fputs, puts

GETTIME (TC)

#include <dos.h>

*void gettime (struct time *timep);*

Описание

Функция **gettime** получает текущее время и записывает его в структуру, адрес которой задается значением аргумента *timep*.

struct time

{

unsigned char ti_min /*минуты*/

unsigned char ti_hour /*часы*/

unsigned char ti_hund /*сотые доли секунд*/

unsigned char ti_sec /*секунды*/

}

Возвращаемое значение

Функция не возвращает значения.

GETVECT (TC)

#include <dos.h>

*void interrupt(*getvect(intr_num))();*

int intr_num;

Описание

ОС MS-DOS поддерживает набор распознаваемых аппаратурой векторов прерываний с номерами от 0 до 255. 4-байтовое значение, которое хранится в каждом векторе, является адресом функции обработки прерывания.

Функция **getvect** читает значение вектора прерывания по номеру прерывания и интерпретирует считанное значение как *far*-указатель на функцию обработки прерывания.

Используется системный вызов операционной системы MS-DOS 0x35.

Возвращаемое значение

Адрес функции, используемой как обработчик прерываний.

Смотри также

disable, setvect

GETVERIFY (TC)

#include <dos.h>

int getverify(void);

Описание

Функция позволяет узнать, установлен ли внутренний флажок ОС MS-DOS, управляющий выполнением контрольных проверок при каждой записи на диск. Если этот флаг включен, каждая операция записи на диск проверяется.

Используется системный вызов ОС MS-DOS 0x54.

Возвращаемое значение

Функция **getverify** возвращает значение 0, если проверка отключена. Возвращается значение 1, если проверка включена.

Смотри также

setverify

GETW (TC & MSC)

#include <stdio.h>

int getw(stream);

*FILE *stream;*

Описание

Функция **getw** читает очередное двоичное значение типа *int* из входного потока *stream* (считывая для этого количество байтов, равное *sizeof(int)*) и продвигает внутренний указатель файла, связанный с этим потоком, на следующий еще не прочитанный символ.

Функция **getw** не предполагает специального выравнивания символов в потоке.

Возвращаемое значение

Прочитанное значение целого.

Значение EOF, если обнаружена ошибка или встретился конец файла; однако значение EOF может быть возвращено и при правильном значении целого, так что необходимо использовать функции **feof** и **ferror**, чтобы распознать ошибочные условия.

Замечание. Функция может быть использована для совместимости с другими библиотеками. Проблема переноса может заключаться в том, что размеры переменной типа *int* и старшинство байт в переменной типа *int* могут быть различными в разных архитектурах.

Смотри также

putw

GMTIME (TC & MSC & ANSI)

#include <time.h>

*struct tm *gmtime(time);*

*long *time;*

Описание

Функция **gmtime** преобразует время из длинного целого к нотации по Гринвичу.

Длинное значение времени представляет собой секунды, прошедшие с момента 00:00:00 1 января 1970 г., по Гринвичу (Greenwich Mean Time); это значение обычно получают при вызове функции *time*.

Функция **gmtime** разбивает значение *time* и помещает компоненты в структуру типа *tm*, определенного в файле *time.h*.

Результат отражает время по Гринвичу (Greenwich Mean Time), но не время региона.

Структура типа *tm* содержит следующие поля:

tm_sec Секунды

tm_min Минуты

tm_hour Часы (0-24)

tm_mday День месяца (1-31)

tm_mon Месяц (0-11; январь=0)

tm_year Год (текущий год минус 1900)

tm_wday День недели (0-6; воскресенье=0)

tm_yday День года (0-365; первое января=0)

tm_isdst Ненулевое, если установлено Daylight Saving Time, иначе нулевое.

Под ОС MS-DOS даты до 1980 года не воспринимаются. Если параметр *time* задает дату до 1 января 1980 г., то *gmtime* возвращает указатель на структуру, представляющую время 00:00:00 1 января 1980 г.

Возвращаемое значение

Указатель на строку-результат.

Нет ошибочных кодов возврата.

Смотри также

asctime, ctime, ftime, localtime, time

GSIGNAL (TC 1.5)

#include <signal.h>

int gsignal(sig);

int sig;

Описание

Функция **gsignal** возбуждает программный сигнал (пользовательское прерывание выполняемого процесса) и вызывает функцию обработки, связанную с этим сигналом.

Замечание. В системе программирования TC версии 2.0 используется для тех же целей функция *raise*.

Смотри также

ssignal, signal, abort, exit, _exit, _fpreset, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

HALLOC (MSC)

#include <malloc.h> /*используется только для описания функции*/

*char huge *halloc(n, size);*

long n; /*обратите внимание на тип параметра n*/

unsigned size;

Описание

Функция **halloc** позволяет выделять динамически участок памяти размером более 64 Кбайтов.

Функция выделяет память для массива из *n* элементов, каждый элемент которого имеет размер в *size* байтов.

Каждый элемент инициализируется 0.

Если размер массива превышает 128 Кбайтов, то размер элемента массива должен быть степенью 2.

Замечание. Система программирования TC не предоставляет аналогичных возможностей.

Возвращаемое значение

Huge-указатель на выделенную память.

Выделенная память, на которую указывает возвращаемое значение, является выровненной для любого типа объектов. Чтобы получить указатель типа, отличного от *char huge*, необходимо провести соответствующие преобразования.

Возвращается значение NULL, если нет достаточной доступной памяти.

Смотри также

calloc, free, hfree, malloc, realloc, faralloc

HARDERR (TC)

#include <dos.h>

void harderr(fptr);

*int(*fptr)();*

Описание

Функция **harderr** предназначена для обработки аппаратных ошибок.

Функция **harderr** регистрирует функцию, которая будет использоваться для обработки аппаратных ошибок вместо стандартных обработчиков.

Используется системное прерывание MS-DOS 0x24.

Функция, на которую указывает параметр *fptr*, должна иметь заголовок вида:

handler(int errval, int ax, int bp, int si)

errval — это код ошибки, установленный в регистре DI; *ax*, *bp* и *si* — это соответствующие регистры; *ax* указывает, на каком диске или устройстве произошла ошибка. Если *ax*<0, то не на диске. Если *ax*>0, то, используя маску 0x00FF, можно получить номер диска (1=A, 2=B и т.д.).

bp и *si* вместе указывают на заголовок драйвера устройства, соответствующего отказавшему дисководу. *bp* содержит адрес сегмента, *si* — смещение.

Функция, на которую указывает *fptr*, не вызывается непосредственно.

Функция **harderr** настраивает обработчик прерываний MS-DOS, который в свою очередь будет вызывать задаваемую функцию при возникновении аппаратных ошибок.

Примечание. Функция, на которую указывает параметр *fptr*, должна возвращать следующие

значения:

0 — игнорировать ошибку,

1 — повторить операцию,

2 — прекратить выполнение операции.

Возвращаемое значение

Функция **harderr** не возвращает значения.

Смотри также

hardresume, hardretn, peek, poke

HARDRESUME (TC)

#include <dos.h>

void hardresume(int rescode);

Описание

Функция **hardresume** предназначена для обработки аппаратных ошибок.

Обработчик ошибок (задаваемый при помощи функции **harderr**) может передать управление обратно операционной системе либо через оператор *return*, либо через вызов функции **hardresume**.

Значение параметра *rescode* передается в ОС и имеет следующий смысл: 0 — игнорировать ошибку, 1 — повторить операцию, 2 — прервать операцию.

Прерывание осуществляется с помощью вызова прерывания MS-DOS 0x23, (*control-break* прерывание).

Возвращаемое значение

Функция не возвращает ни значения, ни управления.

Смотри также

harderr, hardretn

HARDRETN (TC)

#include <dos.h>

void hardretn(int errcode);

Описание

Функция **hardretn** предназначена для выполнения прямого возврата из обработчика прерываний (указанного с помощью функции *harderr*) в прикладную программу.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

harderr, hardresume

HFREE (MSC)

#include <malloc.h> /*используется только для описания функции*/

void hfree(ptr);

*char huge *ptr;*

Описание

Функция **hfree** освобождает блок памяти, выделенный ранее через обращение к функции **halloc**. Число освобождаемых байтов совпадает с указанным в запросе на получение блока. После вызова освобожденный блок вновь доступен для получения.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

halloc, free, malloc

HYPOT (TC & MSC)

#include <math.h> /*используется только для описания функции*/

double hypot(x, y);

double x, y;

Описание

Функция **hypot** вычисляет длину гипотенузы по двум заданным катетам *x* и *y*. Вызов функции **hypot** эквивалентен вычислению значения выражения *sqrt(x*x+y*y)*.

Возвращаемое значение

Длина гипотенузы.

Значение HUGE, если произошло переполнение; при этом переменной *errno* присваивается значение ERANGE.

Смотри также

cabs

INP (MSC), INPORT (TC), INPORTB (TC)

Использование (MSC)

#include <conio.h> /*используется только для описания функции*/

int inp(port);

unsigned port;

Использование (TC)

#include <dos.h>

int inport(int port);

int inportb(int port);

Описание (MSC)

Функция **inp** читает один байт с вводного порта *port*. Аргумент *port* — это беззнаковое целое от 0 до 65535.

Описание (TC)

Функция **inport** читает слово из порта ввода *port*.

Функция **inportb** читает байт из порта ввода *port*.

Примечание (TC). **Inport** и **inportb** — функции, если не включен файл *dos.h*. **Inport** и **inportb** — макросы, вызывающие вставку в текст программы соответствующей машинной инструкции, если включен файл *dos.h*.

Примечание (TC 2.0). Для переносимости программ в файл *dos.h* вставлено макроопределение:

#define inp(a, b) inportb((a), (b))

Возвращаемое значение

Прочитанный байт для функции **inp** (MSC).

Прочитанный байт или прочитанное слово соответственно для функций **inportb** и **inport** (TC).

Нет ошибочных кодов возврата.

Смотри также

outp

INT86 (TC & MSC)

#include <dos.h>

int int86(intno, inregs, outregs);

int intno;

*union REGS *inregs;*

*union REGS *outregs;*

Описание

Функция **int86** выполняет системное прерывание для микропроцессоров семейства Intel 8086/8088, прерывание определяется номером, задаваемым значением параметра *intno*.

До возбуждения прерывания функция **int86** копирует содержимое структуры, адрес которой задается значением параметра *inregs*, в соответствующие регистры.

После завершения обработки прерывания функция копирует содержимое регистров в структуру, адрес которой задается значением параметра *outregs*.

Также копируется значение флага статуса системы (системный флаг) в поле *cflag* в структуре *outregs*.

Функция **int86** используется для непосредственного обращения к системным вызовам операционной системы MS-DOS через прерывания.

Структуры, адреса которых передаются как параметры, имеют тип REGS, описанный в файле *dos.h*.

Возвращаемое значение

Содержимое регистра AX после прерывания.

Если поле *cflag* в *outregs* ненулевое, значит, имела место ошибка и переменной *doserrno* присвоено значение, обозначающее код ошибки.

Смотри также

bdos, intdos, intdosx, int86x

INT86X (MSC & TC)

#include <dos.h>

int int86x(intno, inregs, outregs, segregs);

int intno;

*union REGS *inregs;*

*union REGS *outregs;*

*struct SREGS *segregs;*

Описание

Функция **int86x**, как и функция **int86**, вызывает системное прерывание для процессора семейства Intel 8086/80286, номер прерывания определяется значением параметра *intno*.

До начала прерывания функция **int86x** выполняет копирование содержимого структур, адреса которых задаются значениями параметров *inregs* и *segregs*, в соответствующие регистры.

В структуре, адрес которой задается параметром *segregs*, используются только значения регистров DS и ES.

После прерывания функция **int86x** копирует текущие значения регистров в структуру, адрес которой задается значением параметра *outregs*, и восстанавливает значение регистра DS. Также копируется статус системы (системный флаг) в поле *cflag* в структуре, адрес которой задается значением параметра *outregs*.

Структуры REGS и SREGS описываются в файле *dos.h* (смотри также описание функции **int86**).

struct SREGS

{

unsigned int es; /*значение регистра ES*/

unsigned int cs; /*значение регистра CS*/

unsigned int ss; /*значение регистра SS*/

unsigned int ds; /*значение регистра DS*/

}

Функция **int86x** предназначена для непосредственного обращения к прерываниям DOS, которые используют значение сегмента DS, отличное от значения этого сегмента по умолчанию, и/или принимают аргумент через регистр ES.

Возвращаемое значение

Значение регистра AX после прерывания.

Если поле *cflag* в *outregs* ненулевое, значит, встретилась ошибка и переменной *doserrno* присвоено значение, обозначающее код ошибки.

Замечание (MSC). Значения сегмента, аргументы *segregs*, могут быть получены с использованием либо функции **segread**, либо макроса **FP_SEG**.

Смотри также

bdos, intdos, intdosx, int86, segread, FP_SEG

INTDOS (TC & MSC)

#include <dos.h>

int intdos(inregs, outregs);

*union REGS *inregs;*

*union REGS *outregs;*

Описание

Функция **intdos** выполняет вызов к ОС MS-DOS через прерывание 0x21 (0x21 — основное прерывание, через которое происходит обращение к системным функциям операционной системы MS-DOS).

Выполняются те же действия, что и для функции **int86**.

Структура REGS определяется в файле *dos.h*.

До выполнения команды функция **intdos** копирует содержимое структуры, адрес которой задается значением параметра *inregs*, в соответствующие регистры. После выполнения команды функция **intdos** копирует текущие значения регистров в структуру, адрес которой задается значением параметра *outregs*. Также копируется статус системы (системный флаг) в поле *cflag* в структуре *outregs*. Если это поле ненулевое, значит, флаг был установлен при выполнении системного вызова и сигнализирует об ошибке.

Возвращаемое значение

Значение регистра AX после системного вызова.

Если значение поля *cflag* в структуре *outregs* ненулевое, значит, встретилась ошибка и переменной *doserrno* присвоено значение, соответствующее коду ошибки.

Смотри также

bdos, intdosx

INTDOSX (TC & MSC)

#include <dos.h>

int intdosx(inregs, outregs, segregs);

*union REGS *inregs;*

*union REGS *outregs;*

*struct SREGS *segregs;*

Описание

Функция **intdosx** выполняет вызов к ОС MS-DOS через прерывание 0x21 (0x21 — основное прерывание, через которое происходит обращение к системным функциям операционной системы MS-DOS).

Выполняются те же действия, что и для функции **int86x**.

Структуры REGS и SRREGS определяются в файле *dos.h* (смотри описание функций **int86** и **int86x**).

До прерывания функция **intdosx** копирует содержимое структур, адреса которых задаются значениями параметров *inregs* и *segregs*, в соответствующие регистры. В структуре *segregs* используются только значения регистров DS и ES.

После прерывания функция **intdosx** копирует текущие значения регистров в структуру, адрес которой задается значением параметра *outregs*, и восстанавливает значение регистра DS. Также копируется статус системы (системный флаг) в поле *cflag* в структуре *outregs*.

Функция **intdosx** предназначена для непосредственного обращения к DOS с использованием значения сегмента DS, отличного от значения этого сегмента по умолчанию, и/или передачей аргумента через регистр ES.

Функция **intdosx** предоставляет возможность программам, которые используют сегменты данных *large*-модели памяти или *far*-указатели, определяющие эти сегменты, использовать эти указатели во время вызова системы.

Возвращаемое значение

Значение регистра AX после системного вызова.

Если поле *cflag* в *outregs* ненулевое, значит, встретилась ошибка и переменной *doserrno* присвоено значение, соответствующее коду ошибки.

Замечание (MSC). Значения сегмента, аргументы *segregs*, могут быть получены с использованием либо функции **segread**, либо макроса FP_SEG.

Смотри также

bdos, intdos, int86, segread, FP_SEG

INTR (TC)

#include <dos.h>

*void intr(int intr_num, struct REGPACK *preg);*

Описание

Предоставляет альтернативный интерфейс по сравнению с функцией **int86**.

Функция выполняет прерывание номер *intr_num*, копируя значение регистров из структуры, адрес которой задается значением параметра *preg*.

После завершения прерывания значения регистров записываются в структуру, адрес которой задается значением параметра *preg*.

Структура REGPACK описана в <*dos.h*>:

```
struct REGPACK
{
unsigned r_ax, r_bx, r_cx, r_dx;
unsigned r_ bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

Возвращаемое значение

Функция не возвращает значения.

Смотри также

int86, int86x, intdos, intdosx, geninterrupt

IOCTL (TC)

#include <io.h>

*int ioctl(int handle, int cmd[, int *argdx, int argcx]);*

Описание

Функция **ioctl** предназначена для контроля устройств ввода/вывода.

Использует системный вызов MS-DOS 0x44.

Значение параметра *cmd* указывает вид операции для устройства.

Таблица 10.1.

cmd	Операция
0	Получить информацию об устройстве
1	Задать информацию об устройстве
2	Прочитать байт argcx по адресу, указанному в argdx
3	Записать байт argcx по адресу, указанному в argdx
4	Аналогично 2, только значение параметра handle интерпретируется как номер устройства (0=текущий, 1=A и т.д.)
5	Аналогично 3, только значение параметра handle интерпретируется как номер устройства (0=текущий, 1=A и т.д.)
6	Получить статус ввода
7	Получить статус вывода
8	Проверяет возможность отключения; для ОС MS-DOS версии 3.0 и выше
11	Установить счетчик повторных попыток удовлетворения конфликтов; только для ОС MS-DOS версии 3.0 и выше

Смотрите руководство "MS-DOS Programmer's Reference Manual" на предмет более детальной информации.

Замечание. Функция **ioctl** в системе программирования TC не совместима с функцией **ioctl** в системах программирования языка Си для операционной системы UNIX.

Возвращаемое значение

Если *cmd*=0 или 1 — функция возвращает информацию об устройстве (копию регистра DX).

Если *cmd*=2-5 — функция возвращает число переданных байтов.

Если *cmd*=6-7 — функция возвращает слово состояния устройства.

Функция **ioctl** возвращает -1 при ошибке, при этом переменной *errno* присваивается одно из следующих значений:

EINVAL — неверный аргумент

EBADF — плохой номер файла

EINVDAT — неверные данные

ISALNUM, ISALPHA (TC & MSC & ANSI), ISASCII (TC & MSC)

#include <ctype.h>

int isalnum(c); /*тест для символов ('A'-'Z', 'a'-'z', '0'-'9')*/

int isalpha(c); /*тест для букв ('A'-'Z', 'a'-'z')*/

int isascii(c); /*тест для символов из кодировки ASCII*/

int c;

Описание

Эти функции проверяют условие для аргумента, возвращая ненулевое значение, если целое удовлетворяет условию теста, и нулевое значение, если нет. Имеется в виду набор символов ASCII (Американский стандарт на кодировку информации для обмена).

Функция **isascii** возвращает корректный результат для всех значений целого. Однако остальные функции возвращают корректный результат только для значений целых, соответствующих набору ASCII-символов (т.е. только для тех, для которых значение *isascii(c)* истинно), для не ASCII-символов возвращают значение EOF (константа, определенная в файле *stdio.h*).

Замечание. Функции выполняются как макросы.

Замечание (TC). Проверяется младший байт аргумента.

Смотри также

iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, tolower, toupper

ISATTY (TC & MSC)

#include <io.h> /*используется только для описания функции*/

int isatty(handle);

int handle;

Описание

Функция **isatty** возвращает ненулевое значение, если устройство, связанное с дескриптором файла *handle* (ввод/вывод нижнего уровня), является символьным устройством, и значение 0 — в противном случае.

Символьными устройствами считаются: терминал, консоль, принтер, последовательный порт.

Возвращаемое значение

Ненулевое значение, если устройство символьное.

Значение 0 иначе.

ISCNTRL, ISDIGIT, ISGRAPF, ISLOWER, ISPRINT, ISPUNCT, ISSPACE, ISXDIGIT, ISUPPER (TC & MSC & ANSI)

#include <ctype.h>

int iscntrl(c); /*тест для управляющих символов (0x00-0x1f или 0x7f)*/

int isdigit(c); /*тест для цифр ('0'—'9')*/

int isgrapf(c); /*тест для печатных символов, не включая пробел (0x21-0x7e)*/

int islower(c); /*тест для прописных букв ('a'-'z')*/

int isprint(c); /*тест для печатных символов (0x20-0x7e)*/

int ispunct(c); /*тест для символов — знаков пунктуации*/

int isspace(c); /*тест для пробела <0x09-0x0d или 0x20)*/

int isxdigit(c); /*тест для 16-чных цифр ('A'-'F','a'-'f' или '0'-'9')*/

int isupper(c); /*тест для заглавных букв ('A'-'Z')*/

int c;

Описание

Эти функции проверяют условие для аргумента, возвращая ненулевое значение, если значение аргумента удовлетворяет условию теста, и нулевое значение, если нет. Имеется в виду набор символов ASCII.

Функции возвращают корректный результат только для целых значений, соответствующих набору ASCII-символов (только те, для которых *isascii(c)* истинно), для не ASCII-символов возвращают значение EOF (константа, определенная в файле *stdio.h*).

Замечание. Функции выполняются как макросы.

Замечание (TC). Проверяется младший байт аргумента.

Смотри также

isalnum, isalpha, isascii, toascii, tolower, toupper

ITOA (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

*char *itoa(value, string, radix);*

int value;

*char *string;*

int radix;

Описание

Функция **itoa** преобразует значение, заданное параметром value, в строку символов, завершающуюся нулевым символом ('\0'), и помещает результат по адресу, определяемому значением параметра *string*.

Значение аргумента *radix* определяет систему исчисления для представления результата, значение должно быть в пределах от 2 до 36. Если значение *radix* равно 10 и значение *value* отрицательно, первым символом в строке будет знак '-'.

Возвращаемое значение

Указатель на строку-результат (копия значения параметра *string*).

Нет ошибочных кодов возврата.

Замечание. Область памяти для *string* должна быть достаточной, чтобы поместить результат. Функция может возвращать до 17 байтов.

Смотри также

ltoa, ultoa

KBHIT (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

int kbhit();

Описание

Функция **kbhit** определяет нажатие клавиши (ввод символа) на клавиатуре консольного терминала.

Возвращаемое значение

Ненулевое значение, если какая-либо клавиша была нажата; значение 0 в противном случае.

Смотри также

getch, getche, bioskey

KEEP (TC)

#include <dos.h>

void keep(int status, intsize);

Описание

Возвращает управление MS-DOS со статусом завершения *status*. Текущая программа при этом остается резидентной в оперативной памяти.

Программа ограничивается в длину в *size* параграфов, остальная оперативная память становится

свободной (может быть занята другими программами).

Используется системный вызов MS-DOS 0x31.

Возвращаемое значение

Функция не возвращает значения, как и управления (управление передается не вызвавшей функции, а операционной системе).

Смотри также

abort, exit

LABS (TC & MSC & ANSI)

#include <stdlib.h> /*используется только для описания функции*/

long labs(n);

long n;

Описание

Функция **labs** возвращает абсолютное значение длинного целого *n*.

Возвращаемое значение

Абсолютное значение (типа *long*). Нет ошибочных кодов возврата.

Смотри также

abs, cabs, fabs

LDEXP (TC & MSC & ANSI)

#include <math.h>

double ldexp(x, exp);

double x;

int exp;

Описание

Функция вычисляет значение $x*2^{exp}$, значение *x* умножить на значение 2 в степени *exp*.

Возвращаемое значение

Возвращается значение $x*2^{exp}$.

Замечание (MSC). Возвращается значение HUGE, если произошло переполнение, при этом знак HUGE зависит от знака *x*. Возвращается значение 0, если происходит потеря значимости. В обоих случаях переменной *errno* присваивается значение ERANGE.

Смотри также

frexp, modf

LDIV (TC 2.0 & ANSI)

#include <stdlib.h>

ldiv_t ldiv(number, denom);

long number;

long denom;

Описание

Функция **ldiv** предназначена для выполнения деления двух чисел типа *long*, с выделением частного и остатка от деления.

Значения параметров *number* и *denom* задают делимое и делитель соответственно.

Тип *ldiv_t* описан в файле *stdlib.h* следующим образом:

typedef struct

{

long int quot; /*частное*/

long int rem; /*остаток от деления*/

ldiv_t;

}

Возвращаемое значение

Возвращается структура из двух элементов (возвращаемое значение — сама структура, а не ее адрес), которые содержат частное и остаток от деления.

Смотри также

div

LFIND, LSEARCH (TC & MSC)

#include <search.h> /*используется в системе программирования MSC*/

#include <stdlib.h> /*используется в системе программирования TC*/

*char *lfind(key, base, num, width, compare);*

*char *lsearch(key, num, width, compare);*

*char *key;*

*char *base;*

*unsigned *num, width;*

*int(*compare)(char *pl, char *p2);*

Описание

Функции **lfind** и **lsearch** выполняют последовательный поиск значения *key* (размер *key* считается равным *width* байтов) в массиве из *num* элементов, каждый из которых имеет размер *width* байтов.

В отличие от функции **bsearch**, функции **lfind** и **lsearch** не требуют, чтобы массив был отсортирован.

Аргумент *base* является указателем на начало массива, в котором будет вестись поиск. Если элемент *key* не найден, функция **lsearch** добавляет его в конец, функция **lfind** этого не делает.

Аргумент *compare* является указателем на предполагаемую пользователем функцию, с помощью которой будут сравниваться два элемента, функция, адрес которой содержит параметр *compare*, должна возвращать значение, определяющее их равенство/неравенство.

Эта функция должна сравнивать элементы и возвращать следующие значения:

Ненулевое элемент1 и элемент2 различны

0 элемент1 и элемент2 идентичны

Обе функции, **lfind** и **lsearch**, вызывают функцию, адрес которой задается значением параметра compare, один или более раз во время поиска, передавая ей как параметры указатели на сравниваемые элементы.

Возвращаемое значение

Указатель на первое месторасположение *key* в массиве. Значение NULL, если элемент *key* не найден в массиве.

Смотри также

bsearch

LOCALTIME (TC & MSC)

#include <time.h>

*struct tm *localtime(time);*

*long *time;*

Описание

Функция **localtime** преобразует время из длинного целого к формату структуры *tm*.

Значение *time* — секунды, прошедшие с момента 00:00:00 первого января 1970 г. по Гринвичу (Greenwitch Mean Time); это значение обычно получают с помощью функции *time*.

Функция **localtime** разбивает значение *time*, корректирует его для времени региона (временной зоны) и учитывает приведение к 12-часовому представлению (Daylight Saving Time), и помещает скорректированное время в структуру типа *tm* (смотри описание функции **gmtime** для получения описания структуры *tm*).

Под операционной системой MS-DOS даты до 1980 г. не воспринимаются. Если *time* представляет дату до 1980 г., то **localtime** возвращает структуру, представляющую дату 00:00:00 первого января 1980 г.

Функция **localtime** осуществляет коррекцию для времени региона, если пользователь установил переменную окружения (операционной системы) TZ. Значение переменной TZ должно быть трехбуквенным именем зоны (такой как PST), следующим за, возможно, знаковым числом, задающим различие между временем по Гринвичу (Greenwitch Mean Time) и временем региона. Число может следовать за трехбуквенным именем, задающим способ представления времени (Daylight Saving Time) региона (такой как PDT). Функция **localtime** использует различие между временем по Гринвичу (Greenwitch Mean Time) и временем региона, чтобы скорректировать значение времени.

Если Daylight Saving Time в настоящем установлено в TZ, **localtime** также корректирует время для этого региона. Если переменной TZ не присвоено значение, по умолчанию используется значение PSTPDT.

Когда переменной TZ присвоено значение, три другие переменные окружения, TIMEZONE, DAYLIGHT и TZNAME, устанавливаются автоматически. Смотри описание функции **tzset** для описания этих переменных.

Возвращаемое значение

Указатель на строку-результат.

Нет ошибочных кодов возврата.

Замечание. Функции **gmtime** и **localtime** используют один и тот же буфер для преобразований. Поэтому при каждом вызове одной из этих функций результаты предыдущего вызова теряются.

Смотри также

asctime, ctime, ftime, gmtime, time, tzset

LOCKING (MSC), LOCK(TC), UNLOCK(TC)

Использование (MSC)

#include <sys\locking.h>

#include <io.h>

int locking(handle, mode, nbyte);

int handle;

int mode;

long nbytes;

Использование (TC)

#include <io.h>

int lock(int handle, long offset, long length);

int unlock(int handle, long offset, long length);

Описание (MSC)

Функция **locking** запирает или отпирает *nbytes* байтов файла, определяемого дескриптором *handle* (ввод/вывод нижнего уровня). Запертые байты в файле недоступны последующему чтению или записи другими процессами. Отпирание файла разрешает другим процессам читать и писать в ранее запертые файлы. Запирание и отпирание начинается с текущей позиции указателя файла и действует на последующие *nbytes* байтов или до конца файла.

Mode определяет запирающее действие и должно иметь одно из следующих значений:

LK_LOCK Запереть определяемые байты. Если байты не могут быть заперты, попытаться снова через 1 сек.

Если после 10 попыток байты не удалось запереть, возвращается ошибка.

LK_RLCK Так же, как *lk_lock*.

LK_NBLK Запереть определяемые байты. Если байты не удалось запереть, возвратить ошибку.

LK_NBRLCK Так же, как *lk_nblck*.

LK_UNLCK Отпереть определяемые байты. Байты должны быть заперты ранее. Перечисленные константы определяются в файле *sys\locking.h*:

#define LK_UNLCK 0

#define LK_LOCK 1

#define LK_NBLCK 2

#define LK_RLCK 3 /*блокирование записи в участок файла*/

#define LK_NBRLCK 4 /*разблокирование записи в участок файла*/

В файле может быть несколько запертых участков, но они не должны пересекаться. Только один участок может отпираться за одно обращение. При отпирании файла участок файла, который отпирается, должен соответствовать участку файла, который был ранее заперт.

Функция **locking** не работает со смежными участками, поэтому, если два запертых региона являются смежными, каждый регион должен отпираться отдельно.

Не должно оставаться запертых участков перед закрытием файла или завершением программы.

Возвращаемое значение (MSC)

Значение 0, если запирание прошло успешно.

Значение -1, при неудачной попытке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES Нарушение запирания (участок уже заперт и отперт).

EBADF Неверный дескриптор файла (*handle*).

EDEADLOCK Нарушение запирания: когда определены флаги *lk_lock* или *lk_rlck* и файл не удалось запереть после 10 попыток.

EINVAL режим разделения SHARE_COM не установлен.

Замечание. Запирание должно использоваться только под операционной системой MS-DOS версии 3.0 и выше, под ранними версиями результат не достигается.

Описание (TC)

Функции **lock** и **unlock** предназначены для блокировки и разблокировки файла, заданного дескриптором *handle*. Работают только в версиях ОС MS-DOS 3.0 и выше.

Участок файла определяется по смещению в файле (аргумент *offset*) и размеру (аргумент *length*).

Если файл заблокирован, то все остальные программы могут только читать область, заданную *offset* и *length*.

Возвращаемое значение (TC)

Функции возвращают 0 при успешном выполнении операции и -1 при ошибке.

Смотри также

creat, open

LOG, LOG10 (TC & MSC & ANSI)

#include <math.h>

double log(x);

double log10(x);

double x;

Описание

Функции **log** и **log10** вычисляют натуральный логарифм и логарифм по основанию 10 от параметра *x* соответственно.

Возвращаемое значение

Результат логарифма.

Если значение *x* отрицательно, обе функции печатают сообщение об ошибке DOMAIN в стандартный поток вывода сообщений об ошибках *stderr* и возвращают отрицательное значение HUGE.

Если значение *x* нулевое, обе функции печатают сообщение об ошибке SING в поток *stderr* и возвращают отрицательное значение HUGE. В обоих случаях переменной *errno* присваивается значение EDOM.

Обработку ошибок можно изменить, используя функцию **matherr**.

Смотри также

exp, matherr, pow

LONGJMP (TC & MSC & ANSI)

#include <setjmp.h>

void longjmp(env, value);

jmp_buf env;

int value;

Описание

Функция **longjmp** восстанавливает окружение, ранее сохраненное в переменной *env* посредством вызова функции **setjmp**.

Окружение включает в себя (для системы программирования TC):

— сегментные регистры CS, DS, ES, SS

— регистры-переменные SI, DI

— указатель стека SP

— указатель на рамку функции в стеке BP

— флажки

Для системы программирования TC тип *jmp_buf* определяется в файле *setjmp.h* следующим образом:

```
typedef struct
{
unsigned j_sp;
unsigned j_ss;
unsigned j_flag;
unsigned j_cs;
unsigned j_ip;
unsigned j_bp;
unsigned j_di;
unsigned j_es;
unsigned j_si;
unsigned j_ds;
}
jmp_buf[l]; /*всего 20 байтов*/
```

Для системы программирования MSC тип *jmp_buf* определяется в файле *setjmp.h* следующим образом:

```
#define _JBLEN 9 /*bp, di, si, sp, ret addr, ds*/
typedef int jmp_buf[_JBLEN ]; /*всего 18 байтов*/
```

Функции **setjmp** и **longjmp** дают возможность выполнить нелокальный переход и обычно используются при обработке ошибок и исключительных ситуаций.

Вызов **setjmp** сохраняет текущее окружение в *env*. Последующий вызов **longjmp** восстанавливает сохраненное окружение и возвращает управление в точку после соответствующего вызова **setjmp**. Выполнение продолжается таким же образом, как если бы заданное value было возвращено при вызове **setjmp**.

Вызов функции **longjmp** должен быть выполнен после вызова функции **setjmp**. Управление не должно покидать функцию (в том смысле, что не должен выполняться возврат из этой функции), из которой был произведен вызов функции **setjmp**, в промежуток времени между вызовами функций **setjmp** и **longjmp**.

Замечание. При вызове **setjmp** не сохраняются значения регистровых переменных.

Возвращаемое значение

Функция не возвращает значения в точку вызова и не возвращает управления в точку вызова.

Смотри также

setjmp

_LROTL (TC 2.0)

 # include <stdlib.h>

unsigned long _lrotl(val, count);

unsigned long val;

int count;

Описание

Функция **_lrotl** выполняет циклический сдвиг данного значения *val* на *count* битов влево.

Возвращаемое значение

Возвращается значение — результат сдвига.

Смотри также

_lrotr

_LROTR (TC 2.0)

#include <stdlib.h>

unsigned long _lrotr(val, count);

unsigned long val;

int count;

Описание

Функция **_lrotr** выполняет циклический сдвиг данного значения *val* на *count* битов вправо.

Возвращаемое значение

Возвращается значение — результат сдвига.

Смотри также

_lrotl

LSEEK (TC & MSC)

#include <io.h> /*используется только для описания функции*/

long lseek(handle, offset, origin);

int handle;

long offset;

int origin;

Описание

Функция **lseek** перемещает (внутренний) указатель файла, связанного с дескриптором *handle*

origin.

Следующая операция с файлом начинается с нового места. Аргумент *origin* должен иметь одно из следующих значений:

SEEK_SET=0 Начало файла

SEEK_CUR=1 Текущая позиция указателя файла

SEEK_END=2 Конец файла

Функция **lseek** используется, чтобы переместить указатель в файле.

Указатель может быть перемещен за конец файла. Однако попытка переместить указатель на позицию до начала файла приведет к ошибке.

Возвращаемое значение

Смещение в байтах новой позиции указателя от начала файла.

Значение -1L сигнализирует об ошибке, при этом переменной *errno* присваивается одно из следующих значений:

EBADF — недействительный дескриптор файла *handle*

EINVAL — недействительное значение для *origin*, или позиция, определяемая смещением *offset*, указывает до начала файла

Для устройств типа терминала и принтера возвращаемое значение не определено.

Смотри также

fseek, tell

LTOA (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

char ltoa(value, string, radix);

long value;

*char *string;*

int radix;

Описание

Функция преобразует заданное значение *value* типа *long int* в символьную строку, завершающуюся нулевым символом, и помещает результат в символьный массив, адрес которого задается значением параметра *string*.

Значение аргумента *radix* определяет систему исчисления для представления *value*; это значение должно быть в пределах от 2 до 36. Если значение *radix* равно 10 и *value* отрицательно, первым символом в строку помещается знак '-'.

Возвращаемое значение

Указатель на строку.

Нет ошибочных кодов возврата.

Замечание. Область памяти, отводимой для *string*, должна быть достаточной, чтобы поместить возвращаемую строку. Функция может возвратить до 33 байтов.

Смотри также

itoa, ultoa

MALLOC (TC & MSC & ANSI)

#include <stdlib.h> /*используется в СП TC, MSC*/

#include <malloc.h> /*используется в СП MSC*/

#include <alloc.h> /*используется в СП TC*/

*void *malloc(n);*

size_t n;

Описание

Библиотека языка Си предоставляет механизм распределения динамической памяти (*heap*). Этот механизм позволяет динамически (по мере возникновения необходимости) запрашивать из программы дополнительные области оперативной памяти.

В малых моделях памяти (*tiny*, *small*, *medium*) доступно для использования все пространство между концом сегмента статических данных программы и вершиной программного стека, за исключением 256-байтной буферной зоны непосредственно около вершины стека.

В больших моделях памяти (*compact*, *large*, *huge*) все пространство между стеком программы и верхней границей физической памяти доступно для динамического размещения памяти.

Функция malloc динамически распределяет блок памяти размером, как минимум, *size* байтов. (Блок может быть больше, чем *size* байтов, в результате выравнивания). Блок является выравненным для размещения объектов любого типа, т. е. можно не заботиться о выравнивании. Инициализация буфера не производится.

Имя типа *size_t* определяется следующим образом:

typedef unsigned size_t;

Возвращаемое значение

Указатель на выделенную область памяти.

Значение NULL, если нет достаточного количества доступной для распределения памяти.

Смотри также

free, halloc, hfree, calloc, realloc, farmalloc

MATHERR (TC & MSC)

#include <math.h>

int matherr(x);

*struct exeption *x;*

Описание

Функция **matherr** обрабатывает ошибки, сгенерированные функциями из математической библиотеки. Такие функции вызывают функцию **matherr**, когда обнаружена ошибка. Пользователь может задавать свою функцию **matherr** (для этого достаточно просто описать функцию с таким именем в своей программе), чтобы позаботиться о специальной обработке ошибок.

Когда ошибка встретилась в функции из математической библиотеки, функция matherr вызывается с аргументом-указателем на структуру типа *exeption* (определенную в *math.h*),

```
struct exception

{

int type;

char *name;

double arg1, arg2, retval;

};
```

Элемент *type* определяет тип ошибки и принимает одно из следующих значений, описанных в *math.h*:

DOMAIN — значение аргумента вне области определения

SING — особенность аргумента

OVERFLOW — переполнение

UNDERFLOW — потеря значимости

TLOSS — полная потеря значимости

PLOSS — частичная потеря значимости

Элемент *name* указывает на строку, содержащую имя функции, которая привела к ошибке. Элементы *arg1* и *arg2* устанавливаются в значения аргументов, которые привели к ошибке. (Если задан только один аргумент, он помещается в *arg1*.)

Элемент *retval* — возвращаемое значение по умолчанию для этой ошибки; пользователь может изменить возвращаемое значение.

Возвращаемое функцией **matherr** значение определяет, встретилась ли действительно ошибка. Если **matherr** возвратила ноль, печатается сообщение об ошибке и переменной *errno* присваивается соответствующее значение — код ошибки.

Если функция **matherr** возвращает ненулевое значение, сообщение об ошибке не печатается и значение переменной *errno* остается неизменным.

Возвращаемое значение

Значение 0, если встретилась ошибка.

Ненулевое значение, если все в порядке.

Смотри также

acos, asm, atan, atan2, bessel, cabs, cos, cosh, exp, hypot, log, pow, sin, sinh, sqrt, tan

_MEMAVL (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned int _memavl();

Описание

Функция **_memavl** возвращает приблизительный размер в байтах памяти, доступной для динамического распределения в данном сегменте.

Эта функция может быть использована вместе с функциями **calloc**, **malloc**, или **realloc** в малой и средней моделях памяти и с функцией **_nmalloc** во всех моделях памяти.

Возвращаемое значение

Размер в байтах, как беззнаковое целое.

Смотри также

calloc, malloc, freect, realloc, stackavail

MEMCCPY (TC & MSC)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*(TC*/

#include <string.h> /*(TC & MSC)*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

*char *memccpy(dest, src, C, cnt);*

*char *dest;*

*char *src;*

int C;

unsigned cnt;

Описание

Функция **memccpy** копирует нуль или более байтов из символьного массива, адрес которого задается значением параметра *src*, в символьный массив, адрес которого задается значением параметра *dest*, до тех пор, пока не будет скопирован символ *C* или пока не будет скопировано *cnt* байтов.

Возвращаемое значение

Если символ *C* был скопирован, функция возвращает указатель на следующий байт за скопированным символом.

Значение NULL, если символ *C* не был скопирован.

Смотри также

memchr, memcmp, memcpy, memset

MEMCHR (TC & MSC & ANSI)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*<TC)*/

#include <string.h> /*(TC & MSC)*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

*char *memchr(buf, C, cnt);*

*char *buf;*

int C;

unsigned cnt;

Описание

Функция **memchr** ведет поиск символа *C* в первых *ent* байтах символьного массива, адрес которого определяется значением параметра *buf*.

Поиск продолжается до тех пор, пока не будет найден символ *C* или пока не будет проверено *cnt* байтов.

Возвращаемое значение

Указатель на расположение *C* в *buf*.

Значение NULL, если символ *C* не найден в пределах первых *cnt* байтов массива.

Смотри также

memccpy, memcmp, memcpy, memset

MEMCMP (TC & MSC & ANSI)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*(TC)*/

#include <string.h> /*(TC & MSC)*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

int memcmp(buf1, buf2,cnt);

*char *buf1;*

*char *buf2;*

unsigned cnt;

Описание

Функция **memcmp** лексикографически сравнивает первые *cnt* байтов областей памяти, адреса которых определяются значениями параметров *buf1* и *buf2*, и возвращает значение, определяющее соотношение содержимого областей:

Меньше 0, если содержимое *buf1* меньше, чем *buf2*.

0, если содержимое *buf1* идентично *buf2*.

Больше 0, если содержимое *buf1* больше, чем *buf2*.

Смотри также

memccpy, memchr, memcpy, memset

MEMCPY (TC & MSC & ANSI)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*(TC)*/

#include <string.h> /*(TC & MSC)*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

char memcpy(dest, src, cnt);

*char *dest;*

*char *src;*

unsigned cnt;

Описание

Функция **memcpy** копирует *cnt* байтов из области, адрес которой определяется значением параметра *src*, в область, адрес которой определяется значением параметра *dest*. Если некоторые участки *src* и *dest* перекрываются, функция **memcpy** гарантирует, что байты в перекрывающемся участке скопируются раньше, чем будут перекрыты.

Возвращаемое значение

Значение параметра *dest*.

Смотри также

memccpy, memchr, memcmp, memset

MEMICMP (TC & MSC)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*(TC*/

#include <string.h> /*(TC & MSO*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

int memicmp(buf1, buf2, cnt);

*char *buf1;*

*char *buf2;*

unsigned cnt;

Описание

Функция **memicmp** лексикографически сравнивает первые *cnt* байтов областей памяти, адреса которых определяются значениями параметров *buf1* и *buf2*, считая заглавные и прописные буквы

эквивалентными. Функция возвращает значение, обозначающее соотношение содержимого *buf1* и *buf2*:

Меньше 0 содержимое *buf1* меньше *buf2*

0 содержимое *buf1* идентично *buf2*

Больше 0 содержимое *buf1* больше *buf2*

Смотри также

memccpy, memchr, memcmp, memcpy, memset

MEMMOVE (TC & ANSI)

#include <mem.h>

#include <string.h>

/*для использования функции достаточно указать один включаемый файл (любой)*/

char memmove(dest, src, cnt);

*char *dest;*

*char *src;*

unsigned cnt;

Описание

Функция **memmove** копирует *cnt* байтов из символьного массива, адрес которого задается значением параметра *src*, в символьный массив, адрес которого задается значением параметра *dest*.

Если области *src* и *dest* перекрываются, функция **memmove** гарантирует, что байты в перекрывающемся участке скопируются раньше, чем будут перекрыты (копирование произойдет корректно).

Функция **memmove** полностью идентична функции **memcpy**.

Возвращаемое значение

Значение параметра *dest*.

Смотри также

memccpy, memchr, memcmp, memset

MEMSET (TC & MSC & ANSI)

#include <memory.h> /*(MSC)*/

#include <mem.h> /*(TC)*/

#include <string.h> /*(TC & MSG)*/

/*для использования функции достаточно указать один включаемый файл (любой)*/

*char *memset(dest, val, cnt);*

*char *dest;*

int val;

unsigned cnt;

Описание

Функция **memset** присваивает значение *val* первым *cnt* байтам области памяти, адрес которой задается значением параметра *dest*.

Возвращаемое значение

Значение параметра *dest*.

Смотри также

memccpy, memchr, memcmp, memcpy

MKDIR (TC & MSC)

#include <direct.h> /*используется в системе программирования MSC*/

#include <dir.h> /*используется в системе программирования TC*/

int mkdir(pathname);

*char *pathname;*

Описание

Функция **mkdir** создает новый каталог с именем, задаваемым символьной строкой, адрес которой определяется значением параметра *pathname*.

За один раз можно создать только один каталог, поэтому создается только каталог с последним именем (в последовательности пути по каталогам) в *pathname*.

Возвращаемое значение

Значение 0, если новый каталог был создан.

Значение -1, сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — каталог не создан; заданное имя является именем существующего файла, каталога или устройства.

ENOENT — имя *pathname* не найдено.

Смотри также

chdir, rmdir

MKTEMP (TC & MSC)

#include <io.h> /*используется только для описания функции*/

*char *mktemp(template);*

*char *template;*

Описание

Функция создает уникальное имя файла посредством модификации заданного имени *template*. Аргумент *template* имеет форму

baseXXXXXX

base — это часть нового имени файла, задаваемая пользователем;

XXXXXX (шесть символов '*X*') — часть имени, определяемая функцией **mktemp**.

Функция **mktemp** сохраняет префикс *base* и заменяет шесть *X* алфавитно-цифровым символом, за которым следует значение из 5 цифр. Пятицифровое значение есть уникальное число, определяющее вызванный процесс.

Алфавитно-цифровой символ есть цифра ноль ('0'), когда функция **mktemp** вызывается в первый раз с заданным именем *template*.

При последующих вызовах из того же процесса с тем же *template* **mktemp** проверяет, чтобы выяснить, было ли использовано предыдущее имя для создания файлов. Если файл не существует для заданного имени, **mktemp** возвращает это имя. Если файлы существуют для всех ранее возвращенных имен, функция **mktemp** создает новое имя, заменив алфавитно-цифровой символ в имени на следующую букву. Например, если первым возвращенным именем было "*t012345*" и это имя использовалось для создания файла, следующим возвращаемым именем будет "*ta12345*". При создании новых имен функция **mktemp** использует, как правило, символ '*0*' и буквы от '*a*' до '*z*'.

Возвращаемое значение

Указатель на модифицированное имя *template*.

Значение NULL, если строка *template* имеет плохой формат.

Замечание. Функция **mktemp** генерирует уникальные имена файлов, но не создает и не открывает файлы.

Смотри также

fopen, getpid, open

MK_FP (TC)

#include <dos.h>

*void far *MK_FP(seg, ofs);*

unsigned seg;

unsigned ofs;

Описание

Макрос **MK_FP** используется для составления четырехбайтового (*far*) указателя из его составляющих — адреса сегмента (значение параметра *seg*) и смещения в сегменте (значение параметра *ofs*).

Возвращаемое значение

Возвращается указатель типа *far*.

Смотри также

FP_OFF, FP_SEG

MODF (TC & MSC & ANSI)

#include <math.h>

double modf(x, intptr);

double x;

*double *intptr;*

void movmem(dest, src, cnt);

*char *dest;*

*char *src;*

unsigned cnt;

Описание

Функция **movmem** копирует *cnt* байтов из области памяти, адрес которой определяется значением параметра *src*, в область памяти, адрес которой определяется значением параметра *dest*.

Если некоторые участки областей, на которые указывают значения параметров *src* и *dest*, перекрываются, функция **movmem** гарантирует, что байты в перекрывающемся участке скопируются раньше, чем будут перекрыты.

Функция **movmem** идентична функции **memcpy**, за исключением возвращаемого значения.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

memccpy, memchr, memcmp, memset

_MSIZE (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned _msize(ptr);

*void *ptr;*

Описание

Функция возвращает размер в байтах блока памяти, полученного по вызову **calloc**, **malloc** или **realloc**.

Возвращаемое значение

Размер в байтах как беззнаковое целое.

Смотри также

malloc

_NFREE (MSC)

#include <malloc.h> /*используется только для описания функции*/

void _nfree(ptr);

*char near *ptr;*

Описание

Функция **_nfree** освобождает блок памяти. Значение аргумента *ptr* указывает на блок памяти, ранее выделенный по вызову функции **_nmalloc**.

Число освобождаемых байтов равняется числу байтов, заданных в вызове функции **_nmalloc**, по которому было возвращено значение *ptr*.

После вызова функции **_nfree** освобожденный блок снова доступен для распределения.

Замечание. Попытка освободить область, не полученную посредством **_nmalloc**, может влиять на последующее распределение памяти и привести к ошибкам.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

_nmalloc, free, malloc

_NMALLOC (MSC)

#include <malloc.h> /*используется только для описания функции*/

*char near *_nmalloc (size);*

unsigned size;

Описание

Функция **_nmalloc** получает блок памяти минимум *size* байтов в пределах текущего сегмента данных. (Блок может быть более чем *nsize* байтов в результате выравнивания.)

Изучите также описание функции **malloc**.

Возвращаемое значение

Указатель на полученный блок. Область памяти, на которую указывает возвращаемое значение, является выравненной для любого типа объектов. Чтобы получить указатель типа, отличного от *char*, необходимо использовать операцию явного преобразования типа.

Значение NULL, если нет достаточной доступной для распределения памяти.

Смотри также

_nfree, _nmsize, malloc, realloc

_NMSIZE (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned _nmsize(ptr);

char near ptr;

Описание

Функция **_nmsize** возвращает размер в байтах блока памяти, полученного ранее при вызове **_nmalloc**.

Возвращаемое значение

Размер в байтах, как беззнаковое целое.

Смотри также

_ffree, _fmalloc, _fmsize, malloc, _msize, _nfree, _nmalloc

NOSOUND (TC 2.0)

#include <dos.h>

void nosound(void);

Описание

Функция **nosound** отключает устройство подачи звукового сигнала.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

delay, sound

ONEXIT (MSC)

#include <stdlib.h>

onexit_t onexit(func);

onexit_t func;

Описание

Функция **onexit** принимает как аргумент адрес функции *func*, которая будет вызвана при нормальном завершении программы.

Тип *onexit_t* определяется в файле *stdlib.h* следующим образом:

typedef int(*onexit_t)();

Не более чем 32 функции могут быть зарегистрированы с помощью функции **onexit**; функция **onexit** возвращает значение NULL, если число функций превышает 32.

Функции, передающиеся **onexit**, не могут использовать параметры.

Возвращаемое значение

Указатель на функцию (значение параметра).

Возвращается значение NULL, если функция не может быть зарегистрирована как вызываемая при завершении программы.

Смотри также

exit, atexit

OPEN (TC & MSC), _OPEN (TC)

#include <fcntl.h> /*файл содержит описания констант*/

#include <sys\stat.h> /*файл содержит описания констант*/

#include <sys\types.h> /*(MSC) файл содержит описания констант*/

#include <io.h> /*файл содержит прототип функции*/

int open(pathname, oflag[, pmode]);

*char *pathname;*

int oflag;

int pmode;

int _open(pathname, oflag);

*char *pathname;*

int oflag;

Описание

Функция **open** открывает файл (ввод/вывод нижнего уровня), имя которого задается строкой, адрес которой определяется значением аргумента *pathname*, и подготавливает файл для чтения или записи, как определено значением аргумента *oflag*.

Значение аргумента *oflag* является целым значением, сформированным комбинацией одной или более следующих констант, определенных в файле *fcntl.h*; когда задается более одной константы, они разделяются операциями ИЛИ (|).

Значение констант следующее:

O_APPEND Переместить указатель файла на конец файла перед каждой операцией записи.

O_CREAT Создать и открыть новый файл для записи; не дает результата, если файл уже существует.

O_EXCL Возвращает ошибочный код, если файл, определенный в *pathname*, уже существует. Применяется только вместе с O_CREAT.

O_RDONLY Открыть файл только для чтения; если этот флаг задан, ни O_RDWR, ни O_WRONLY не могут быть заданы.

O_RDWR Открыть файл для чтения и записи; если этот флаг задан, ни O_RDONLY, ни O_WRONLY не могут быть заданы.

O_TRUNC Открыть и сузить существующий файл до нулевой длины; файл должен иметь доступ для записи. Содержимое файла теряется.

O_WRONLY Открыть файл только для записи; если этот флаг задан, ни O_RDONLY, ни O_RDWR не могут быть заданы.

O_BINARY Открыть файл в двоичном режиме (смотри FOPEN).

O_TEXT Открыть файл в текстовом режиме (смотри FOPEN).

Замечание. O_TRUNC полностью затирает содержимое существующего файла. Будьте с ним осторожны.

Режим открытия файла (текстовый или двоичный) при использовании функции **open** определяется по значению глобальной переменной *_fmode* (ее значением может быть одна из констант O_TEXT или O_BINARY).

Чтобы открыть файл в другом режиме с помощью функции **open**, можно либо установить соответствующее значение переменной *_fmode*, либо открыть файл через вызов функции **open** с установленным атрибутом O_TEXT или 0_BYNARY.

Аргумент *pmode* для функции **open** требуется только тогда, когда задано значение O_CREAT. Если файл существует, значение параметра *pmode* игнорируется. Иначе, значение параметра *pmode* определяет способ доступа к файлу, который устанавливается, когда новый файл закрывается впервые. *Pmode* есть целое значение, содержащее одну или обе константы S_IWRITE и S_IREAD, определенных в файле *sys\stat.h*. Когда определяются обе константы, они разделяются оператором ИЛИ(|).

Действие *pmode* следующее:

S_IWRITE Доступ для записи

S_IREAD Доступ для чтения

S_IREAD|S_IWRITE Доступ для чтения и записи.

Если доступ для записи не задан, файл открыт только для чтения. Под MS-DOS все файлы доступны для чтения; поэтому нет необходимости задавать только доступ для записи. Таким образом, режимы S_IWRITE и S_IREAD|S_IWRITE эквивалентны. Функция **_open** доступна только системе программирования TC.

Режим открытия файла (текстовый или двоичный) при использовании функции **_open** всегда определяется по значению глобальной переменной *_fmode* (ее значением может быть одна из констант O_TEXT или O_BINARY).

Для функции **_open** в рамках операционной системы MS-DOS версий более ранних, чем 3.0, допустимое значение параметра *oflag* может быть только O_RDONLY, O_WRONLY или O_RDWR.

Для версий ОС 3.0 и более поздних допустимы дополнительно следующие значения:

O_NOINHERIT — файл не должен наследоваться процессами-потомками

O_DENYALL — доступ к файлу только через текущий дескриптор

O_DENYWRITE — для других операций открытия этого файла разрешить только режим чтения

O_DENYREAD — для других операций открытия этого файла разрешить только режим записи

O_DENYNONE — сделать файл полностью разделяемым

Для системы программирования TC в файле *fcntl.h* определены следующие константы:

#define O_RDONLY 1

#define O_WRONLY 2

#define O_RBWR 4

#define O_CREAT 0x0100

#define O_TRUNC 0x0200

#define O_EXCL 0x0400

#define _O_RUNFLAGS 0x0700

#define _O_EOF 0x0200 /*устанавливается, когда текстовый файл оканчивается символом <ctrl/Z>*/

#define O_APPEND 0x0800 /*Специальные признаки ОС MS-DOS*/

#define O_CHANGED 0x1000 /*пользователь не должен изменять эти биты*/

#define O_DEVICE 0x2000

#deftne O_TEXT 0x4000 /*преобразование CR-LF*/

#define O_BINARY 0x8000 /*нет преобразования CR-LF*/

/*параметры ОС MS-DOS версии 3.0 и более поздних*/

#define O_NOINHERIT 0x80

#define O_DENYALL 0x10

#define O_DENYWRITE 0x20

#define O_DENYREAD 0x30

#define O_DENYNONE 0x40

Для системы программирования MSC в файле *fcntl.h* определены следующие константы:

#define O_RDONLY 0x0000

#define O_WRONLY 0x0001

#define O_RDWR 0x0002

#define O_APPEND 0x0008

#define O_CREAT 0x0100

#define O_TRUNC 0x0200

#define O_EXCL 0x0400

#define O_TEXT 0x4000

#define O_BINARY 0x8000

#define O_RAW O_BINARY

#define O_NOINHERIT 0x0080

Отметим, что константа O_APPEND имеет разные значения в разных системах программирования, поэтому предпочтительнее использовать только мнемонические обозначения констант.

Посмотрите также описание функций **creat** и **_creat**.

Возвращаемое значение

Функции **open** и **_open** возвращают значение дескриптора для открытого файла.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — заданное *pathname* является каталогом, или попытка открыть для записи файл, который открыт только для чтения.

EEXIST - определены флаги O_CREAT и O_EXECL, но файл уже существует.

EMFILE — нет больше доступных дескрипторов *handle* (слишком много открытых файлов).

ENOENT — файл или каталог на пути к нему не найдены.

Смотри также

access, chmod, close, creat, dup, dup2, fopen, sopen, umask

OUTP (MSC), OUTPORT (TC), OUTPORTB (TC)

Использование (MSC)

#include <conio.h> /*используется только для описания функции*/

int outp(port, value);

unsigned port;

int value;

Использование (TC)

#include <dos.h>

void outport(int port, int word);

void outportb(int port, char byte);

Описание (MSC)

Функция **outp** записывает байт (значение параметра *value*) в выводной порт, определяемый значением параметра *port*. Значение параметра *port* может быть беззнаковым целым в пределах от 0 до 65535; значение параметра *value* может быть целым от 0 до 255.

Описание (TC)

Функция **outport** записывает слово *word* в порт вывода *port*.

Функция **outportb** записывает байт *byte* в порт вывода *port*.

Примечание (TC). **outport** и **outportb** — функции, если не включен файл *dos.h*. **outport** и **outportb** — макросы, вызывающие вставку в текст программы соответствующей машинной инструкции, если включен файл *dos.h*.

Примечание (TC 2.0). Для переносимости программ в *dos.h* вставлено макроопределение:

#define outp(a.b) outportb((a),(b))

Возвращаемое значение (MSC)

Значение *value*.

Нет кодов ошибок.

Возвращаемое значение (TC)

Функция не возвращает значения.

Смотри также

inp

PARSFNM (TC)

#include <dos.h>

*char *parsfnm(cmdline, fcbptr, option);*

*char *cmdline;*

*struct fcb *fcbptr;*

int option;

Описание

Функция **parsfnm** разбирает строку, обычно командную строку, на которую указывает *cmdline*, для выделения имени файла. Имя файла будет размещено в структуре *fcbptr*. Функция использует системный вызов MS-DOS 0x29 для разбора имени файла.

Смотрите Ваш MS-DOS Programmer's Reference Manual для детального перечисления возможных значений параметра *option*.

Отметим лишь, что значение *option* соответствует значению регистра AL указанного системного вызова.

Возвращаемое значение

Если вызов прошел удачно, то функция возвращает указатель следующего байта в строке после имени файла, в противном случае функция возвращает значение NULL.

PEEK, PEEKB (TC)

#include <dos.h>

int peek(unsigned segment, unsigned offset);

int peekbf unsigned segment, unsigned offset);

Описание

Функции **peek** и **peekb** используются для доступа к памяти через значения адреса сегмента и смещения.

Если файл *dos.h* включен, то это макросы, если нет, функции.

Возвращаемое значение

(адрес сегмента) и *offset* (смещение).

Функция **peekb** возвращает значение байта памяти по адресу, заданному параметрами *segment* (адрес сегмента) и *offset* (смещение).

Смотри также

poke, pokeb

PERROR (TC & MSC & ANSI)

#include <stdlib.h> /*используется только для описания функции*/

void perror(string);

*char *string;*

int errno;

int sys_nerr;

char sys_errlist[sys_nerr];

Описание

Функция **perror** печатает сообщение об ошибке в стандартный поток для вывода сообщений об ошибках *stderr*. Сначала печатается аргумент-строка (*string*), за которой следует двоеточие, затем системное сообщение об ошибке, соответствующее значению переменной errno, и символ перевода строки ('*n*').

Код ошибки находится в переменной *errno*, которая должна быть описана на верхнем уровне (программист не должен сам ее описывать, она будет взята из стандартной библиотеки языка Си). Системное ошибочное сообщение доступно через переменную *sys_errlist*, которая является массивом сообщений, упорядоченных по соответствующим кодам ошибок.

Функция **perror** печатает сообщение, используя значение переменной *errno* как индекс по массиву *sys_errlist*.

Значение переменной *sys_nerr* устанавливается как максимальное число элементов в *sys_errlist* массиве. Для правильной работы функция **perror** должна быть вызвана сразу после библиотечной функции, которая возвратила код ошибки, иначе значение переменной errno может быть затерто при последующих вызовах.

Замечание. Под ОС MS-DOS некоторые значения *errno* из списка *errno.h* не используются.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

clearerr, ferror, strerror

POKE, POKEB (TC)

#include <dos.h>

int poke(unsigned segment, unsigned offset, int value);

int pokeb(unsigned segment, unsigned offset, char value);

Описание

Функции **poke** и **pokeb** используются для доступа к памяти через значения адреса сегмента и смещения.

Если файл *dos.h* включен, то это макросы, если нет, функции.

Функция **poke** записывает значение типа целое (*value*) по адресу, заданному параметрами *segment* (адрес сегмента) и *offset* (смещение).

Функция **pokeb** записывает байт-значение (*value*) по адресу, заданному параметрами *segment* (адрес сегмента) и *offset* (смещение).

Возвращаемое значение

Функции **poke** и **pokeb** не возвращают значения.

Смотри также

peek, peekb

POLY (TC)

#include <math.h>

double poly(x, n, c);

double x;

int n;

double c[];

Описание

Функция **poly** получает значение в точке *x* для полинома степени *n*, заданного коэффициентами *c[0], c[1], ..., c[n]*.

Возвращаемое значение

Функция возвращает значение полинома.

POW (TC & MSC & ANSI)

#include <math.h>

double pow(x, y);

double x;

double y;

Описание

Функция **pow** вычисляет значение *x* в степени *y*.

Возвращаемое значение

Значение *x* в степени *y*.

Значение 1, если значение *y* нулевое.

Значение HUGE, если значение *x* равно 0 и *y* отрицательно; при этом переменной errno присваивается значение ERANGE.

Значение 0, если *x* отрицательное и *y* не целое; при этом переменной *errno* присваивается EDOM и печатается сообщение об ошибке DOMAIN в поток *stderr*.

Значение HUGE, отрицательное или положительное, если произошло переполнение; при этом переменной *errno* присваивается значение ERANGE.

Смотри также

exp, log, sqrt, pow10

POW10 (TC)

#include <math.h>

double pow(p);

int p;

Описание

Функция **pow10** вычисляет значение *10* в степени *p*.

Возвращаемое значение

Значение *10* в степени *p*. Вычисление результата производится с двойной точностью. Все значения аргумента принимаются как допустимые, хотя возможны ситуации переполнения и потери точности.

Смотри также

pow, exp, log, sqrt

PRINTF (TC & MSC & ANSI)

#include <stdio.h>

int printf(format_string[, argument...]);

*char *format_string;*

Описание

Функция **printf** печатает символы и формирует и печатает задаваемые аргументами значения в стандартный выводной поток *stdout* (ввод/вывод верхнего уровня). Функция **printf** имеет переменное число параметров.

Строка описания формата вывода, адрес которой задается значением единственного обязательного параметра *format_string*, состоит из обычных символов, специальных управляющих последовательностей символов (*escape*-последовательностей) и, если за параметром *format_string* следуют еще дополнительные аргументы, спецификаций полей формата вывода по одному для каждого дополнительного аргумента.

Обычные символы и *escape*-последовательности просто копируются в поток *stdout* в порядке их появления. Например, оператор

printf("Первая строка\n\t\t Вторая строка\n");

выводит:

Первая строка

Вторая строка

Если за параметром *format_string* следуют аргументы *arguments*, то *format_string* должна содержать спецификации форматов, определяющих выводной формат для этих аргументов.

Спецификации форматов начинаются с символа процента ('%') и описываются ниже.

Строка описания формата (адрес которой задается значением параметра *format_string*) прочитывается слева направо. Когда встречается первая спецификация формата, значение первого аргумента после параметра *format_string* преобразуется и выводится согласно спецификации формата. Вторая спецификация формата преобразует и выводит второй аргумент, и, таким образом, обработка продолжается до конца *format_string*. Если задано больше аргументов, чем спецификаций формата, лишние аргументы игнорируются.

Результат будет неопределенным, если нет достаточного количества аргументов для всех спецификаций форматов. Спецификация формата имеет следующую форму:

%[flags][width][.precision][F|N|h|l|L]type

Спецификация формата не содержит внутри себя пробелов. Каждое поле спецификации формата есть одиночный символ или число, означающие необязательный параметр формата. Символ *type*, который появляется после последнего поля формата, определяет, как будет интерпретироваться соответствующий данному описанию формата аргумент: как символ, строка или число (смотри таблицу 10.2).

Самая простая спецификация формата содержит только знак процента и символ *type* (например, "%*s*").

Необязательные поля управляют другими параметрами форматирования:

flags выравнивание выводных символов, управление печатью знаковых символов ('+' и '-'), пробелов, десятичных точек, восьмеричных и шестнадцатеричных префиксов (смотри таблицу 10.3)

width минимальное число выводимых символов

precision максимальное число символов, которые будут напечатаны, для всех или части выводных полей; или минимальное число цифр, которые будут печататься, для значения целого (смотри таблицу 10.4).

F, N префиксы, которые позволяют пользователю не принимать во внимание способы адресации и используемую модель памяти:

F используется в малой модели для печати значений, которые описаны как *far*

N используется в средней, большой и верхней моделях для значений *near*.

префиксы *F* и *N* должны быть использованы только с типами "%*s*" и "%*p*" (в системе программирования TC также с типом "%*n*"), так как они имеют смысл только с аргументами, задаваемыми как указатель.

h, l размер аргумента:

h используется как префикс с целыми типами *d*, *i*, *o*, *u*, *x* и *X*, чтобы определить, что аргумент является коротким целым (*short int*).

l используется как префикс с типами *d*, *i*, *o*, *u*, *x* и *X*, чтобы определить, что аргумент является длинным целым (*long int*); также используется как префикс с типами *c*, *e*, *f*, *g* или *G*, чтобы показать, что аргументы имеют тип *double*, а не *float*.

L (только для системы программирования TC версии 2.0)

Используется как префикс с типами *d*, *i*, *o*, *u*, *x* и *X*, чтобы определить, что аргумент является длинным целым (*long int*); также используется как префикс с типами *c*, *e*, *f*, *g* или *G*, чтобы показать, что аргументы имеют тип *long double*, а не *double*.

Если за символом процента ('%') следует символ, который не является полем формата, этот символ просто копируется в *stdout*.

Таблица 10.2.

Символы type

Символ	Тип аргумента	Выводной формат
d	целое	десятичное целое со знаком
i	целое	десятичное целое со знаком
u	целое	десятичное целое без знака
o	целое	восьмеричное целое без знака
x	целое	шестнадцатеричное целое без знака, для обозначения шестнадцатеричных цифр используются символы"abcdef"
X	целое	шестнадцатеричное целое без знака, для обозначения шестнадцатеричных цифр используются символы "ABCDEF"
f	с плавающей точкой	Значение со знаком, имеет форму [-]dddd.dddd, где dddd — одна и более десятичных цифр. Число цифр до десятичной точки зависит от величины значения, а число цифр после десятичной точки зависит от требуемой точности.
e	с плавающей точкой	Значение со знаком имеет форму [-]d.ddddе[sign]ddd, где d — одна десятичная цифра, dddd — одна или более десятичных цифр, ddd — ровно три десятичные цифры, и sign — это '+' или '-'.
E	с плавающей точкой	Идентично 'e' формату, но знак 'e' заменяется на 'E'.
g	с плавающей точкой	Значение со знаком печатается в формате 'f' или 'e', однако является более компактным для заданного значения и точности precision (смотри ниже). Формат 'e' используется только тогда, когда экспонента значения меньше, чем -4, или больше, чем precision. Незначащие нули отбрасываются, десятичная точка печатается только в том случае, если одна или более цифр следуют за ней.
G	с плавающей точкой	Идентично формату 'g', но символ 'E' используется для обозначения экспоненты вместо 'e'.
c	символ	Одиночный символ.
s	строка	Символы печатаются до первого нулевого символа ('\0') или до тех пор, пока не будет напечатано precision символов.
n	указатель на целое	В ячейку памяти, адрес которой задается как аргумент, соответствующий данной спецификации формата, записывается число символов, выведенных в выводной поток к текущему моменту в данном вызове функции printf.

		Для системы программирования MSC:
p	far-указатель	Печатает адрес, заданный аргументом, в виде xxxx:yyyy, где xxxx — адрес сегмента, yyyy — смещение, цифры x и y — шестнадцатеричные цифры; %Np печатает только смещение адреса, yyyy.
		Для системы программирования TC:
p	указатель	Печатает адрес, заданный аргументом, в формате, соответствующем типу указателя по умолчанию в текущей модели памяти: — far-указатели печатаются в виде xxxx:yyyy, где xxxx — адрес сегмента, yyyy — смещение, цифры x и y — шестнадцатеричные цифры; — для near-указателей печатается только смещение, yyyy. Чтобы напечатать значение указателя, тип которого отличен от типа по умолчанию, необходимо использовать спецификаторы формата "%Fp" и "%Np"

Таблица 10.3

Символы flag

Флаг	Действие флага	По умолчанию
-	Результат, если количество символов для вывода меньше указанного размера поля (width), выравнивается по левой границе (дописываются пробелы справа).	Выравнивание по правой границе
+	Добавляет префикс знака ('+' или '-') для выводного значения, если выводное значение является знаковым типом.	Знак печатается только для отрицательных значений.
пробел	Добавляет префикс пробел для выводного значения, если выводное значение со знаком и положительно; ' + ' преобладает над флагом ' ', если указаны оба.	Пробел не добавляется.
#	Когда используется формат o, x или X, флаг '#' добавляет префикс ненулевому выводному значению: 0, 0x или 0X соответственно. Когда используется формат e, E или f, выводное значение будет содержать десятичную точку во всех случаях. Когда используется формат g или G, выводное значение будет содержать десятичную точку и незначащие нули не подавляются.	Нет префикса.

Поле *width* является неотрицательным десятичным целым, управляющим минимальным числом символов, выводимых по данному описателю формата. Если число символов в выводном значении меньше, чем определено в *width*, пробелы добавляются слева или справа (в зависимости от того, определен ли флаг '-') до тех пор, пока не будет достигнут размер *width*.

Спецификация *width* никогда не означает обрезание значащих символов в выводимом значении; если число символов в выводном значении больше, чем определено в *width*, или *width* не задано, печатаются все символы (возможно, согласно указаниям спецификации *precision*).

Спецификация *width* может быть задана как символ звездочка ('*'); в этом случае сам текущий аргумент из списка аргументов (который должен иметь тип *int*) предполагается как значение, задающее минимальное количество выводимых символов. Этот аргумент должен предшествовать в списке аргументов значению, которое будет форматироваться и выводиться по текущему описателю формата.

Спецификация *precision* является неотрицательным десятичным числом, следующим за точкой ('.'), она определяет точное число символов, которые должны быть напечатаны, или место десятичной точки.

Спецификация *precision* может сузить выводное значение или округлить в случае значения с плавающей точкой.

Спецификация *precision* может быть задана символом звездочка ('*'); в этом случае сам аргумент из списка аргументов предполагается как значение *precision*. Аргумент *precision* должен предшествовать в списке аргументов значению, которое будет форматироваться.

Интерпретация значения *precision* в зависимости от заданного типа *type* показана в таблице 10.4.

Таблица 10.4

Type	Действие	По умолчанию
d i u 0 x X	Precision определяет минимальное число цифр, которые будут напечатаны. Если число цифр в аргументе меньше, чем precision, выводное значение добавляется слева нулями. Значение не сжимается, когда число цифр превышает precision.	Если precision есть 0, или отсутствует, или точка появилась без числа, следующего за ним, precision устанавливается в 1.
e E f	Precision определяет число цифр, которые будут напечатаны после десятичной точки. Последняя напечатанная цифра округляется.	По умолчанию precision равно 6. Если precision равно 0, десятичная точка не печатается.
g G	Precision определяет максимальное число значимых цифр, которые будут напечатаны.	Все десятичные цифры печатаются.
c	Нет действия.	Символ печатается.
s	Precision определяет максимальное число символов, которые будут напечатаны. Лишние символы не печатаются.	Символы печатаются до тех пор, пока не будет достигнут символ конца строки ('\0').

Возвращаемое значение

Число напечатанных символов.

Смотри также

fprintf, scanf, sprintf, vfprintf, vprintf, vsprintf

PUTC, PUTCHAR (TC & MSC & ANSI)

#include <stdio.h>

int putc (sim, stream);

int sim;

*FILE *stream;*

int putchar(sim);

int sim;

Описание

Функция **putc** записывает одиночный символ, код которого задается значением параметра *sim*, в выводной поток, определяемый значением параметра *stream*, в текущую позицию (ввод/вывод верхнего уровня).

Вызов функции **putchar(sim)** идентичен вызову функции **putc(sim, stdout)**.

Возвращаемое значение

Код выведенного символа.

Значение EOF, если произошла ошибка; необходимо использовать функцию **ferror**, чтобы определить, какая произошла ошибка.

Замечание. Функции **putc** и **putchar** сходны с функциями **fputc** и **fputchar**, но являются на самом деле макросами, а не функциями.

Смотри также

fputc, fputchar, getc, getchar

PUTCH (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

void putch(sim);

int sim;

Описание

Функция **putch** записывает символ, код которого задается значением параметра *sim*, на консольный терминал.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

cprintf, getch, getche, putc

PUTENV (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

int putenv(envstring);

*char *envstring;*

Описание

Функция **putenv** добавляет новые переменные окружения или модифицирует существующие переменные окружения.

Переменные окружения (это переменные оболочки операционной системы MS-DOS) определяют окружение, в котором выполняется процесс (например, можно задать последовательность каталогов, в которых надо искать файл с выполняемым кодом при запуске программы на выполнение).

Значение аргумента *envstring* определяет адрес строки (массива символов), следующего вида:

varname=string

varname — имя переменной окружения, которая будет добавлена или модифицирована;

string — новое значение переменной.

Если *varname* уже существует в окружении, ее значение будет *string*; иначе, новое значение *string* будет добавлено в окружение. Переменная может быть установлена в пустое значение, если *string* — пустая строка.

Возвращаемое значение

Значение 0, если операция выполнена успешно. Значение -1 сигнализирует об ошибке.

Замечание. Функции **getenv** и **putenv** используют глобальную переменную *environ*, чтобы получить доступ к таблице окружения. Функция **putenv** может изменить значение переменной *environ* таким образом, что сделает неправильным (устаревшим) значение аргумента *envp* для функции **main**.

Смотри также

getenv

PUTS (TC & MSC & ANSI)

#include <stdio.h>

int puts(string);

*char *string;*

Описание

Функция **puts** записывает строку, адрес которой определяется значением параметра *string*, в стандартный выводной поток *stdout* (ввод/вывод верхнего уровня), добавляя в завершение символ конца строки ('*n*').

Выводятся символы из строки, пока не встретится завершающий строку нулевой символ ('*0*').

Возвращаемое значение

Последний записанный символ, который всегда является символом '*n*'.

Значение EOF сигнализирует об ошибке.

Смотри также

fputs, gets, printf

PUTW (TC & MSC)

#include <stdio.h>

int putw(binint, stream);

int binint;

*FILE *stream;*

Описание

Записывает двоичное значение типа *int* в текущую позицию заданного выводного потока *stream* (ввод/вывод верхнего уровня).

Функция **putw** не производит выравнивания символов в файле.

Замечание. Функция **putw** предназначалась первоначально для работы с ранними библиотеками. Могут возникнуть проблемы с переносом программ, использующих эту функцию, так как размер *int* и старшинство байтов в *int* различаются при переходе от системы к системе.

Возвращаемое значение

Записанное значение.

Значение EOF сигнализирует об ошибке; необходимо использовать функцию **ferror**, чтобы определить, какая встретилась ошибка.

Смотри также

getw

QSORT (TC & MSC & ANSI)

#include <search.h> /*используется только для описания функции*/

void qsort(base, num, width, compare);

*char *base;*

unsigned num, width;

*int(*compare)();*

Описание

Функция **qsort** выполняет алгоритм быстрой сортировки, чтобы отсортировать массив из num элементов, каждый из которых имеет размер *width* байт.

Значение аргумента *base* задает адрес сортируемого массива. Функция **qsort** переупорядочивает этот массив.

Значение аргумента *compare* задает адрес предполагаемой пользователем функции сравнения элементов массива. Функция **qsort** будет вызывать функцию, адрес которой задается значением аргумента *compare*, один или более раз в течение сортировки, передавая ей как параметры указатели на два элемента массива при каждом вызове.

Функция, адрес которой задается значением аргумента *compare*, должна сравнивать элементы, потом возвращать одно из следующих значений:

Меньше 0 элемент 1 меньше элемента 2;

0 элемент 1 эквивалентен элементу 2;

Больше 0 элемент 1 больше элемента 2;

Возвращаемое значение

Функция не возвращает значения.

Смотри также

bsearch, lsearch

RAISE (TC 2.0 & MSC 5.1 & ANSI)

#include <signal.h>

int raise(sig);

int sig;

Описание

Функция **raise** возбуждает программный сигнал (пользовательское прерывание выполняемого процесса) и вызывает функцию обработки, связанную с этим сигналом. Если пользователь не установил для данного сигнала свою функцию обработки с помощью функции **signal**, выполняются действия, принятые по умолчанию для данного сигнала.

Замечание. В системе программирования TC версии 1.5 используется для тех же целей функция **gsignal**.

За подробностями обращайтесь к описанию функции **signal**.

Возвращаемое значение

Значение 0, если операция выполнена успешно. Ненулевое значение, если произошла ошибка.

Смотри также

signal, abort, exit, _exit, _fpreset, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

RAND (TC & MSC & ANSI)

#include <stdlib.h> /*используется только для описания функции*/

int rand();

Описание

Функция **rand** возвращает псевдослучайное целое значение в пределах от 0 до 32767.

Перед первым обращением датчик случайных чисел надо проинициализировать посредством вызова функции **srand**.

Используется конгруэнтный датчик случайных чисел с периодом 232.

Возвращаемое значение

Случайное число.

Нет ошибочных кодов возврата.

Смотри также

srand, random

RANDBRD, RANDBWR (TC)

#include <dos.h>

int randbrd(fcbptr, recent);

*struct fcb *fcbptr;*

int recent;

int randbwr(fcbptr, recent);

*struct fcb *fcbptr;*

int recent;

Описание

Функция **randbrd (random block read)** читает *recent* записей, используя FCB-блок (блок управляющей информации о файле в формате операционной системы MS-DOS), адрес которого определяется значением параметра **fcbptr**.

Считывается заданное количество записей по адресу области передачи данных диска *dta* (смотри описание функции *setdta*).

Функция **randbrd** использует системный вызов 0x27 ОС MS-DOS.

Функция **randbwr** записывает *recent* записей, используя FCB-блок, адрес которого определяется значением параметра **fcbptr**. Функция **randbwr** использует системный вызов 0x28 ОС MS-DOS.

В файле *dos.h* определяется тип структуры *fcb*:

```
struct fcb
{
char fcb_drive; /*устройство 0 = по умолчанию, 1 = 'A', 2 = 'B'*/
char fcb_name [8]; /*имя файла*/
char fcb_ext[3]; /*расширение файла*/
short fcb_curblk; /*текущий номер блока*/
short fcb recsize; /*размер логической записи в байтах*/
long fcb_filsize; /*размер файла в байтах*/
short fcb date; /*дата последней модификации файла*/
char fcb_resv[10]; /*поле, зарезервированное ОС*/
char fcb_currec; /*текущая запись в блоке*/
long fcb_random; /*номер записи для доступа*/
};
```

Возвращаемое значение

Значение 0 — все записи считаны или записаны.

Значение 1 — для записи — нет свободного места (запись не записывается), для чтения — конец файла совпал с концом последней считанной записи.

Значение 2 — считываемая/записываемая запись имеет адрес, превышающий 0xFFFF (считано/записано записей столько, сколько можно).

Значение 3 — для чтения — конец файла встретился до окончания считывания последней записи.

Смотри также

getdta, setdta

RANDOM, RANDOMIZE (TC 2.0)

#include <stdlib.h>

#include <time.h> /*только для функции randomize*/

int random(int num);

void randomize(void);

Описание

Функция **random** возвращает псевдослучайное целое значение в пределах от 0 до значения *num*-1.

Для инициализации датчика случайных чисел необходимо использовать функцию **randomize**.

Замечание 1. Соответствующие функции есть в системе программирования Turbo-Pascal

Замечание 2. Функции **random** и **randomize** реализуются как макроопределения

#define random(num) (rand()%(num))

#define randomize() srand ((unsigned)time(NULL))

Возвращаемое значение

Функция **random** возвращает случайное число в диапазоне от 0 до *num*-1.

Функция **randomize** не возвращает значения.

Смотри также

rand, srand

READ (TC & MSC), _READ (TC)

#include <io.h> /*используется только для описания функций*/

int read(handle, buffer, count);

int handle;

*char *buffer;*

unsigned int count;

int _read(handle, buffer, count); /*функция _read доступна только в системе программирования TC*/

int handle;

*char *buffer;*

unsigned int count;

Описание

Функция **read** пытается прочитать *count* байтов из файла, связанного с дескриптором *handle* (ввод/вывод нижнего уровня), в область памяти, адрес которой определяется значением параметра *buffer*.

Операция чтения начинается с текущей позиции (внутреннего) указателя файла, связанного с заданным файлом. После чтения указатель файла указывает на следующий непрочитанный символ.

Если файл открыт в текстовом режиме, последовательности символов <*CR*><*LF*> преобразуются в символ <*LF*> при вводе. Кроме того, символ <*CTRL/Z*> интерпретируется как символ конца файла при вводе. Для файлов, открываемых на чтение или чтение/запись, по возможности происходит проверка и удаление символов <*CTRL/Z*> (это необходимо для корректной работы функции **fseek**).

В двоичном режиме перечисленные выше преобразования не производятся. Функция **_read** доступна только в системе программирования TC.

В отличие от функции **read**, функция **_read** — это прямое обращение к системному вызову операционной системы, не выполняются преобразования, связанные с работой с файлом в текстовом режиме.

Возвращаемое значение

Функции **read** и **_read** возвращают число действительно прочитанных символов, которое может быть меньше, чем значение *count*, если в файле оставалось меньше байтов, чем значение *count*, или файл был открыт в текстовом виде (смотри ниже).

Функции возвращают значение 0, если была попытка прочитать конец файла.

Функции возвращают значение -1, если произошла ошибка; при этом переменной *errno* присваивается одно из значений:

EVADF Заданное значение *handle* недействительно; или файл не был открыт для чтения; или файл защищен (для ОС MS-DOS версии 3.0 и выше)

Если было прочитано более 32 Кбайтов (максимальный размер для *int*), возвращаемое значение должно быть типа *unsigned int*. Однако максимальное число байтов, которые могут быть прочитаны из файла за одно обращение к функции **read** или **_read**, равно 65534, так как значение 65535 (0xffff) является представлением значения -1 и должно интерпретироваться как ошибка.

Если файл был открыт в текстовом виде, возвращаемое значение для функции **read** может не соответствовать числу действительно прочитанных байтов. Это происходит потому, что каждая комбинация символов <*возврат-каретки*><*новая-строка*> (<*CR*><*LF*>) заменяется символом новой строки <*LF*> и только этот символ засчитывается в возвращаемое значение.

(TC) Также не пропускается и не учитывается в текстовом режиме чтения из файла символ <*CTRL/Z*>.

Смотри также

creat, fread, open, write, _creat, _open, _write

REALLOC (TC & MSC & ANSI)

#include <*malloc.h*> /*используется в системе программирования MSC*/

#include <*stdlib.h*> /*используется в системе программирования TC*/

#include <*alloc.h*> /*используется в системе программирования TC*/

*char *realloc(ptr, size);*

*char *ptr;*

*unsigned *size;*

Описание

Функция **realloc** изменяет размер области памяти, выделенной посредством вызова функции **malloc** или **calloc**.

Значение аргумента *ptr* определяет адрес области. Значение аргумента *size* задает новый размер области в байтах. Содержимое блока не изменяется, за исключением области в диапазоне между новым и старым размерами.

(MSC) Аргумент *ptr* при вызове функции **realloc** может также указывать на блок, который был освобожден только что (с помощью функции **free** или **hfree**), пока не будет вызвана одна из функций **calloc**, **malloc**, **halloc** или **realloc**.

(TC) Если аргумент *ptr* равен нулю, **realloc** работает как **malloc**.

Возвращаемое значение

Указатель на переопределенный блок памяти.

Значение NULL, если нет достаточно доступной памяти для расширения блока до заданного размера *size*, при этом исходный блок освобождается.

(TC) Возвращается значение NULL, если значение аргумента *size* равно 0.

Память, на которую указывает возвращаемое значение, является выравненной для объектов любого типа. Чтобы получить указатель отличного от типа *char**, его необходимо преобразовать.

Смотри также

calloc, free, halloc, malloc

REMOVE (TC 2.0 & MSC & ANSI)

#include <io.h> /*используется в системе программирования MSC*/

#include <stdio.h> /*используется в системах программирования MSC и TC*/

int remove(pathname);

*char *pathname;*

Описание

Функция **remove** удаляет файл, имя которого определяется строкой, адрес которой задается значением параметра *pathname*.

Возвращаемое значение

Значение 0, если удаление прошло успешно.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — имя пути определяет каталог или файл, открытый только для чтения.

ENOENT — файл или путь не найдены.

Смотри также

close, unlink

RENAME (TC & MSC & ANSI)

#include <io.h> /*используется в системе программирования MSC*/

#include <stdio.h> /*используется в системах программирования MSC, TC*/

int rename(oldname, newname);

*char *oldname;*

*char *newname;*

Описание

Функция **rename** переименовывает файл или каталог, имя которого определяется символьной строкой, адрес которой задается значением параметра *oldname*, давая файлу новое имя, определяемое строкой, адрес которой задается значением параметра *newname*.

Строка, адрес которой задается значением параметра *oldname*, должна определять путь существующего файла или каталога.

Строка, адрес которой задается значением параметра *newname*, не должна определять имя существующего файла или каталога.

Функция **rename** может быть использована, чтобы переместить файл из одного каталога в другой, задав другое имя пути в *newname*.

Однако файлы не могут быть перемещены с одного устройства на другое (например, с "A:" на "B:").

Каталоги могут только переименовываться.

Возвращаемое значение

Значение 0, если переименование прошло успешно.

Ненулевое значение сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES - Файл или каталог, определенный в *newname*, уже существует или не может быть создан (неправильный путь); или *oldname* является каталогом, а *newname* определяет другой путь.

ENOENT — файл или путь, определенные в *oldname*, не найдены.

EXDEV — попытка переместить файл на другое устройство.

Смотри также

creat, fopen, open

REWIND (TC & MSC & ANSI)

#include <stdio.h> /*используется только для описания функции*/

void rewind(stream);

*FILE *stream;*

Описание

Функция **rewind** перемещает (внутренний) указатель файла, связанного с потоком *stream* (высокоуровневый ввод/вывод), на начало файла.

Вызов функции **rewind(stream)** эквивалентен вызову функции **fseek(stream, 0L, SEEK_SET)**, за исключением того, что функция **rewind** очищает признак конца файла и признак ошибки, а функция **fseek** этого не делает.

Кроме того, функция **fseek** возвращает значение, которое обозначает, был ли указатель успешно перемещен, а **rewind** не возвращает такого значения.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

fseek, ftell

RMDIR (TC & MSC)

#include <direct.h> /*используется в системе программирования MSC*/

#include <dir.h> /*используется в системе программирования TC*/

int rmdir(pathname);

*char *pathname;*

Описание

Функция **rmdir** удаляет каталог, имя которого задается строкой, адрес которой определяет значение параметра *pathname*.

Каталог должен быть пустым, не может быть текущим рабочим каталогом или корневым каталогом.

Возвращаемое значение

Значение 0, если каталог успешно удален.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — заданное имя пути не является каталогом;

или каталог не пустой; или каталог является текущим рабочим каталогом или корневым каталогом.

ENOENT — имя пути не найдено.

Смотри также

chdir, mkdir

RMTMP (MSC)

#include <stdio.h>

int rmtmp();

Описание

Функция **rmtmp** используется для уничтожения всех временных файлов в текущем каталоге.

Функция **rmtmp** уничтожает только те файлы, которые были созданы посредством функции **tmpfile**.

Функция должна быть использована только в тех каталогах, в которых были созданы временные файлы.

Возвращаемое значение

Число закрытых и удаленных временных файлов.

Смотри также

flushall, tmpfile, tmpnam

_ROTL (TC 2.0)

#include <stdlib.h>

unsigned _rotl(val, count);

unsigned val;

int count;

Описание

Функция **_rotl** выполняет циклический сдвиг данного значения *val* на *count* битов влево.

Возвращаемое значение

Возвращается значение — результат сдвига.

Смотри также

_rotr, _lrotl

_ROTR (TC 2.0)

#include <stdlib.h>

unsigned _rotr(val, count);

unsigned val;

int count;

Описание

Функция **_rotr** выполняет циклический сдвиг данного значения *val* на *count* битов вправо.

Возвращаемое значение

Возвращается значение — результат сдвига.

Смотри также

_rotl, _lrotr

SBRK (TC & MSC)

#include <malloc.h> /*(MSC), используется только для описания функции*/

#include <alloc.h> /*(TC), используется только для описания функции*/

*char *sbrk(incr);*

int incr;

Описание

Функция **sbrk** изменяет значение глобальной переменной *break*, соответствующим образом изменяя размер области памяти, отведенной для данного процесса. Значение переменной *break* определяет адрес первого байта оперативной памяти, расположенного за областью, отведенной программе в оперативной памяти.

Функция **sbrk** добавляет значение параметра *incr* к переменной *break*; размер памяти, получаемой процессом, соответственно изменяется. Заметим, что значение *incr* может быть отрицательным, в этом случае область памяти процесса уменьшается на *incr* байтов.

Возвращаемое значение

Старое значение переменной *break*.

Значение -1 обозначает, что нет достаточной доступной памяти; при этом переменной *errno* присваивается значение ENOMEM.

Замечание (MSC). В программах компактной, большой и высшей модели памяти функция **sbrk** не приводит к результату.

Смотри также

calloc, free, malloc, realloc, keep

SCANF (TC & MSC & ANSI)

#include <stdio.h>

int scanf(format_string[, argument...]);

*char *format_string;*

Описание

Функция **scanf** читает данные из стандартного вводного потока *stdin* (высокоуровневый ввод/вывод) в переменные, адреса которых задаются аргументами *arguments*. Функция имеет переменное число параметров.

Значение первого параметра — *format_string* задает адрес строки, которая управляет интерпретацией элементов ввода (вводных полей). Каждый последующий аргумент должен быть указателем на переменную типа, соответствующего типу, определенному очередным описателем формата элемента ввода в строке, адрес которой задается значением параметра *format_string*.

Строка описания формата ввода может содержать:

— Пробельные символы (пробел (' '), табуляция ('*t*') или символ новой строки ('*n*')). Если

первого символа, не являющегося пробельным, считываются из потока ввода, но не участвуют в присваиваниях значений переменным (т.е. попросту игнорируются).

— Печатные символы — все прочие ASCII-символы, за исключением символа процента ('*%*'). Если печатные символы встречаются в *format_string*, то из потока ввода считываются символы, которые не участвуют в выработке присваиваемых переменным значений, но которые должны соответствовать печатным символам, встретившимся в *format_string*. Если последовательность печатных символов из потока ввода не соответствует последовательности печатных символов в *format_string*, работа функции **scanf** завершается.

— Спецификации формата, которые определяются знаком процента ('*%*'). Спецификация формата заставляет **scanf** прочитать и преобразовать символы на входе в значение определяемого типа. Значение присваивается переменной, адрес которой указан в списке аргументов.

Строка, адрес которой задается значением параметра *format_string*, прочитывается слева направо.

Когда встречается первая спецификация формата, значение первого входного поля преобразуется согласно спецификации формата и записывается по адресу, определяемому значением первого параметра после параметра *format_string*. Согласно второй спецификации формата второе входное поле преобразовывается и помещается по адресу, задаваемому значением второго параметра, и так далее до конца *format_string*.

Входное поле, выделяемое из потока ввода, включает все символы до первого пробельного символа, или до первого символа, который не может быть преобразован согласно спецификации формата, или до длины поля *width*, если она определена. Если параметров в вызове функции **scanf** указано больше, чем задано спецификаций формата, лишние параметры игнорируются. Результат не определен, если задано аргументов меньше, чем указано спецификаций формата.

Спецификация формата имеет следующую форму:

%[*][width](F|N][h|I|L|]type

Каждое поле в спецификации формата есть одиночный символ или число. Символ *type*, который появляется в последнем поле спецификации формата, определяет: входное поле интерпретируется как символ, строка или число.

Самая простая спецификация формата содержит только символ процента и символ типа (например, "*%s*").

Если за символом процента ('*%*') следует символ, который не является символом, управляющим форматом, то этот символ и все последующие символы (до следующего символа процента) рассматриваются как простая последовательность символов на входе. Например, чтобы задать символ процента на входе, используйте "*%%*".

Звездочка ('*'), следующая за символом процента, запрещает присвоение получаемого значения по адресу, задаваемому аргументом. Поле сканируется, но значение в память не записывается.

Спецификация *width* является положительным десятичным целым, определяющим максимальное число символов, которые будут прочитаны из потока *stdin* согласно текущему описанию формата ввода элемента. Не более чем *width* символов считываются и участвуют в получении значения, которое помещается по адресу, задаваемому значением соответствующего параметра.

Может быть прочитано меньше, чем *width* символов, если пробельный символ или символ, который не может быть преобразован согласно заданному формату, встретится раньше.

Префикс *F* или *N* позволяет не принимать во внимание способ адресации используемой модели памяти. *F* должен быть префиксом к аргументу, указывающему на далекий (*far*) объект, *N* должен быть префиксом к аргументу, указывающему на близкий (*near*) объект.

Префикс *l* обозначает, что используется *long*-вариант типа поля *type*, тогда как префикс *h* обозначает, что *short*-вариант *type* будет использован. Соответствующий аргумент должен указывать на *long*- или *double*-объект (для *long*-варианта) или *short*-объект (для *short*-варианта).

Префиксы *l* и *k* могут быть использованы с *d*, *l*, *o*, *x* и *u* символами *type*. Модификация *l* может быть использована с *e* и *f* символами. Префиксы *l* и *h* игнорируются, если определен другой *type*.

Префикс *L* (используемый только в системе программирования TC версии 2.0) сочетается только с символами *type*: *e*, *f*, *g* и означает, что значение должно быть *long double*, а не *double*.

Символы *type* и их значения описаны в таблице 10.5.

Таблица 10.5.

Символы type

Символ	Строка на входе	Тип аргумента
d	Десятичное целое	Указатель на int
D	Десятичное целое	Указатель на long
o	Восьмеричное целое	Указатель на int
O	Восьмеричное целое	Указатель на long
x	Шестнадцатеричное целое	Указатель на int
X	Шестнадцатеричное целое	Указатель на long
i	Десятичное, шестнадцатеричное или восьмеричное целое	Указатель на int
I	Десятичное, шестнадцатеричное или восьмеричное целое	Указатель на long
u	Десятичное целое без знака	Указатель на unsigned int
U	Десятичное целое без знака	Указатель на unsigned long
e	Значение с плавающей точкой, состоящее из необязательного знака (+ или -), набора из одной или более десятичных цифр, возможно, содержащих десятичную точку, и необязательной экспоненты ("e" или "E"), за которой следует целое значение, возможно, со знаком.	Указатель на float
F		
c	Символ. Пробельные символы не пропускаются при чтении; чтобы прочитать следующий непробельный символ, используйте спецификацию "%ls"	Указатель на char
s	Строка	Указатель на массив символов, достаточный для размещения входного поля плюс завершающий символ конца строки ('\0'), который добавляется автоматически
N	Чтение из потока ввода не производится	Указатель на int-перемен-ную, в которую производится запись числа успешно прочитанных к данному моменту символов из потока stream

В системе программирования TC версии 2.0 также являются значимыми следующие символы *type*:

Таблица 10.6.

Символ	Строка на входе	Тип аргумента
G E G	Число с плавающей точкой	Указатель на float
P	Шестнадцатеричное значение	Указатель на указатель в виде yyyy:zzzz или типа far или near zzzz Описатель "%p" предполагает, что тип указателя, адрес которого указывается в списке параметров, является типом указателя по умолчанию для используемой модели памяти

В системе программирования TC версии 2.0 последовательности символов на вводе: "+INF", "-INF", "+NAN", "-NAN" распознаются как вещественные числа.

Чтобы прочитать строки, не разделенные символами пробел (' '), для типа "%*s*" может быть использован параметр — набор символов в квадратных скобках ('[' ']').

Соответствующее входное поле читается до первого символа, который не встречается внутри квадратных скобок. Если первый символ в наборе является знаком вставки ('^'), эффект обратный: входное поле читается до первого символа из указанного набора символов.

(TC) Набор символов можно задавать как диапазон, например "[a-z]".

Чтобы поместить строку без символа конца строки ('*0*'), используйте спецификацию %*nc*, где *n* — десятичное целое. В этом случае *c*-тип обозначает, что аргумент является указателем на массив символов. Следующие *n* символов будут прочитаны из входного потока в определяемое место и символ конца строки ('*0*') не будет добавлен.

Функция **scanf** сканирует (просматривает) каждое входное поле, символ за символом. Чтение может быть остановлено раньше, чем будет достигнут пробельный символ по различным причинам:

— определенное *width* количество символов введено;

— следующий символ не может быть преобразован, как определено;

— следующий символ конфликтует с печатным символом в управляющей строке, в которой подразумевается соответствие;

— следующий символ не должен встретиться (или не встретился) в заданном наборе символов.

Когда это произошло, следующий невведенный символ рассматривается как первый символ следующего входного поля.

Возвращаемое значение

Число полей, которые были успешно преобразованы и присвоены. Возвращаемое значение не включает поля, которые были прочитаны, но не присвоены.

Значение EOF, если была попытка прочитать конец файла.

Значение 0, если не было присвоенных полей.

Смотри также

fscanf, printf, sscanf, vfprintf, vprintf, vsprintf

SEARCHPATH (TC)

#include <dir.h>

*char *searchpath(char *filename);*

Описание

Функция **searchpath** используется для поиска файла по его имени, имя файла задается строкой, адрес которой определяется значением параметра *filename*.

Сначала файл ищется в текущем каталоге, если его там нет, то ищется в каталогах, которые определены в переменной окружения PATH (смотри описание функции **putenv**).

Возвращаемое значение

Возвращается адрес строки, где записано полное имя файла, при удачном поиске и значение NULL при неудачном.

Смотри также

exec..., spawn..., system

SEGREAD (TC & MSC)

#include <dos.h>

void segread(segregs);

*struct SREGS *segregs;*

Описание

Функция **segread** заполняет структуру, адрес которой задается значением параметра *segregs*, текущим содержанием сегментных регистров.

Тип структуры SREGS описан в файле *dos.h* (смотри также описание функции **int86x**).

Функция **segread** предназначена для использования с функциями **intdosx** и **int86x** для восстановления значений сегментных регистров для дальнейшего использования.

Возвращаемое значение

Функция не возвращает значения

Смотри также

intdosx, int86x, FP_SEG

SETBLOCK (TC)

#include <dos.h>

int setblock(int seg, int newsize);

Описание

Функция **setblock** изменяет размер сегмента памяти, выделенного ранее через вызов функции **allocmem**.

функции **allocmem**.

Значение аргумента *newsize* определяет новый требуемый размер в параграфах (один параграф равняется 16 байтам) для сегмента.

Все выделяемые блоки выравниваются на границу параграфа.

Возвращаемое значение

Функция **setblock** в случае нормального завершения возвращает значение -1.

В случае ошибки возвращается размер наибольшего блока, который может быть выделен, при этом переменной *_doserrno* присваивается соответствующее значение.

Смотри также

coreleft, freemem, allocmem

SETBUF (TC & MSC & ANSI)

#include <stdio.h>

void setbuf(streem, buffer);

*FILE *stream;*

*char *buffer;*

Описание

Функция **setbuf** позволяет пользователю управлять буферизацией для заданного потока *stream* (высокоуровневый ввод/вывод).

Значение аргумента *stream* должно соответствовать ранее открытому файлу (смотри описание функции **fopen**).

Если значение аргумента *buffer* равняется NULL, это означает отмену буферизации.

Иначе, значение аргумента *buffer* определяет адрес массива символов длины BUFSIZ, где BUFSIZ — размер буфера, константа, определенная в файле *stdio.h*.

Определенный пользователем буфер используется для буферизованного ввода/вывода указанного потока *stream* вместо выделяемого по умолчанию системного буфера.

Потоки *stderr* и *stdout* по умолчанию не буферизованы, но им могут быть присвоены буфера посредством **setbuf**.

Замечание. Последствия буферизации будут непредсказуемы, если только функция **setbuf** не вызвана сразу вслед за функцией **fopen** или **fseek** для данного потока.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

fflush, fopen, fclose

SETCBRK (TC)

#include <dos.h>

setcbrk (int value);

Описание

Функция **setcbrk** устанавливает реакцию на нажатие на клавиатуре комбинации клавиш <*control/break*>.

Если значение параметра *value* равно 0, то проверка выключается (проверка будет проводиться только при вводе/выводе с консольного терминала, принтера, линий связи).

Если значение *value* равно 1, то проверка включается (программа будет прерываться по нажатию <*ctrl/c*> или <*ctrl/break*> в любой момент).

Возвращаемое значение

Возвращается значение аргумента *value*.

Смотри также

getcbrk

SETDATE (TC)

#include <dos.h>

*void setdate(struct date *dateblk);*

Описание

Функция **setdate** устанавливает текущую дату на основе информации, записанной в структуре, адрес которой задается значением параметра *dateblk*.

Тип структуры *date* описан в файле *dos.h*.

struct date

{

int da_year; /*текущий год*/

char da_day; /*месяц года*/

char da_mon; /*месяц/январь-1*/

}

Возвращаемое значение

Функция не возвращает значения.

Смотри также

getdate, gettime, settime

SETDISK (TC)

#include <dir.h>

int setdisk(drive);

int drive;

Описание

Функция **setdisk** устанавливает текущим (используемым по умолчанию) устройством устройство, определяемое значением параметра *drive* (0='A', 1='B', 3='C' и т.д.), используя системный вызов MS-DOS 0x0E.

Возвращаемое значение

Функция **setdisk** возвращает общее число дисков, имеющихся в наличии.

Смотри также

getdisk, getcwd, getcurdir

SETDTA (TC)

#include <dos.h>

*void setdta(char far *dta);*

Описание

Функция **setdta** устанавливает текущий адрес области передачи данных диска (DTA) (подробности смотри в руководстве "Technical Reference Manual").

Используется системный вызов MS-DOS 0x1A.

Примечание. Корректно работает только с моделями памяти: *compact*, *large* и *huge*.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

getdta, findfirst

SETFTIME (TC)

#include <io.h>

int setftime (handle, ftimep);

int handle;

*struct ftime *ftimep;*

Описание

Функция **setftime** изменяет дату и время для открытого файла, дескриптор которого задается значением параметра *handle*.

Время и дата берутся из структуры, адрес которой задается значением параметра *ftimep*.

Тип структуры *ftime* описывается в файле *io.h*:

struct ftime

{

unsigned ft_sec:5; /*секунды*/

unsigned ft_min:6; /*минуты*/

unsigned ft_hour:5; /*часы*/

unsigned ft_day:5; /*дни*/

unsigned ft_month:4; /*месяц*/

unsigned ft_year:7; /*год, начиная с 1980*/

};

Возвращаемое значение

Значение 0 при успешном завершении операции.

Значение <0 при ошибке.

Смотри также

getftime

SETJMP (TC & MSC & ANSI)

#include <setjmp.h>

int setjmp(env);

jmp_buf env;

Описание

Функция **setjmp** сохраняет окружение в момент вызова, это окружение в дальнейшем может быть восстановлено, с использованием функции **longjmp**. Вызов **setjmp** сохраняет текущее состояние окружения в переменной *env*.

Последующий вызов **longjmp** восстанавливает сохраненное окружение и возвращает управление на точку, следующую непосредственно за соответствующим вызовом функции **setjmp** (во время которого было сохранено окружение в ту переменную env, которая указывается при вызове функции **longjmp**).

Значения всех переменных (для системы программирования MSC — за исключением регистровых переменных), доступных в точке вызова **setjmp**, будут иметь те значения, которые они имели в момент вызова **longjmp**.

Замечание (MSC). Значения регистровых переменных непредсказуемы (что существенно, впрочем, только для тех случаев, когда с помощью функции **longjmp** происходит передача управления в пределах текущей функции).

Более детальное описание можно найти в описании функции **longjmp**.

Возвращаемое значение

Значение 0, после вызова функции **setjmp** для сохранения окружения.

Значение аргумента *value* функции **longjmp**, если функция **setjmp** выполняет возврат и возвращает значение как результат вызова функции **longjmp**.

Смотри также

longjmp

SETMEM (TC)

#include <mem.h>

void memset(dest, cnt, stm);

*char *dest;*

unsigned cnt;

int sim;

Описание

Функция **memset** присваивает первым *cnt* байтам области, адрес которой задается значением параметра *dest*, значение *sim*.

Функция **setmem** идентична функции **memset**, за исключением возвращаемого значения и порядка следования параметров.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

memccpy, memchr, memcmp, memcpy

SETMODE (TC & MSC)

#include <io.h> /*файл содержит прототип функции*/

#include <fcntl.h> /*файл содержит определения констант*/

int setmode(handle, mode);

int handle;

int mode;

Описание

Функция **setmode** устанавливает текстовый или двоичный режим работы с файлом, дескриптор которого определяется значением параметра *handle* (ввод/вывод нижнего уровня).

Значение параметра *mode* должно быть одним из следующих:

O_TEXT Устанавливает текстовый режим.

O_BINARY Устанавливает двоичный режим.

Описание режимов содержится в описании функции **open**.

Функция **setmode** обычно используется для модификикации режимов работы, заданных по умолчанию для потоков *stdin*, *stdout*, *sdterr*, *stdoux* и *stdprn*, но может быть использована для любого файла.

Возвращаемое значение

Ранее установленный режим, если операция выполнена успешно.

-1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EBADF Неправильный аргумент *handle*

EINVAL Неправильный аргумент *mode* (ни O_TEXT, ни O_BINARY).

Смотри также

creat, fopen, open, fileno, read, write

SETTIME (TC)

#include <dos.h>

void settime (tlmep);

*struct time *timep;*

Описание

Функция **settime** устанавливает текущее время в системе, на основе информации, записанной в структуре, адрес которой определяется значением параметра *timep*.

Тип структуры *time* описан в файле *dos.h*:

struct time

{

unsigned char ti_min; /*минуты*/

unsigned char ti_hour; /*часы*/

unsigned char ti_hund; /*сотые доли секунд*/

unsigned char ti_sec; /*секунды*/

};

Возвращаемое значение

Функция не возвращает значения.

Смотри также

getdate, gettime, setdale

SETVBUF (TC & MSC & ANSI)

#include <stdio.h>

int setvbuf(stream, buf, type, size);

*FILE *stream;*

*char *buf;*

int type;

int size;

Описание

Функция **setvbuf** позволяет пользователю управлять буферизацией и размером буфера файла, связанного с потоком *stream*.

Поток *stream* должен относиться к открытому файлу.

Массив, адрес которого задается значением параметра *buf*, используется в качестве буфера, если значение этого параметра не NULL, в последнем случае поток не буферизуется.

Если поток буферизуется, значение параметра *type* определяет тип, который должен быть либо _IONBF, либо _IOFBF, или _IOLBF.

Если тип равен _IOFBF или _IOLBF, то значение параметра *size* используется как размер буфера.

Если тип равен _IONBF, то поток не буферизован, и значения параметров *size* и *buf* игнорируются.

Допустимые значения параметра *size*: больше 0 и меньше, чем максимальный размер целого (*int*).

Значения констант определены в файле *stdio.h*:

#define _IOFBF 0 /*буферизация на полный объем буфера*/

#define _IOLBF 1 /*построчная буферизация появление символа '\n' приводит к скидыванию буфера при записи в файл и к прекращению подкачки в буфер при чтении из файла*/

#define _IONBF 2 /*файл не буферизуется*/

Возвращаемое значение

Значение 0, если операция выполнена успешно.

Ненулевое значение, если тип или размер буфера определены некорректно.

Смотри также

setbuf, fflush, fopen, fclose

SETVECT (TC)

#include <dos.h>

void setvect(intr_num, isr);

int intr_num;

*void interrupt(*isr)();*

Описание

Операционная система MS-DOS поддерживает набор распознаваемых аппаратурой векторов прерываний с номерами от 0 до 255. 4-байтовое значение в каждом векторе является действительным адресом места в памяти, где располагается функция обработки прерывания.

Функция **setvect** устанавливает новую функцию обработки прерываний, адрес этой функции задается значением параметра *isr*, для вектора прерывания, номер которого задается значением параметра *intr_num*.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

disable, getvect

SETVERIFY (TC)

#include <dos.h>

void setverify(int value);

Описание

Функция **setverify** предназначена для установления флага ОС MS-DOS проверки записи на диск. Если этот флаг включен, каждая операция записи на диск проверяется.

Используется системный вызов MS-DOS 0x2E.

Функция **setverify** устанавливает флаг по значению параметра *value*: 0 — выключить, 1 — включить.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

getverify

SIGNAL (TC 2.0 & MSC & ANSI)

#include <signal.h>

*int (*signal(sig, func))();*

int sig;

*int (*func)();*

Описание

Функция позволяет процессу выбрать одну из трех возможностей для управления сигналами прерывания программы, поступающими от операционной системы.

Значение параметра *sig* определяет номер сигнала, на который устанавливается новая реакция.

В системе программирования MSC версии 4.0 аргумент *sig* должен иметь значение одной из констант SIGINT или SIGFPE, определенных в файле *signal.h*.

В системе программирования TC версии 2.0 определены сигналы:

#define SIGABRT 22 /*ненормальное завершение, действия по умолчанию: _exit(3)*/

#define SIGFPE 8 /*прерывание по плавающей точке деление на ноль, неверная операция, действия по умолчанию: _exit(1)*/

#define SIGILL 4 /*неверная инструкция действия по умолчанию: _exit(1)*/

#define SIGINT 2 /*прерывание ctrl-break действия по умолчанию: как для INT 0x23*/

#define SIGSEGV 11 /*нарушение доступа к памяти действия по умолчанию: _exit(1)*/

#define SIGTERM 15 /*"мягкое" завершение процесса действия по умолчанию: _exit(1)*/

Для операционной системы OS/2 в системе программирования TC версии 2.0 зарезервированы сигналы:

#define SIGBREAK 21 /*OS/2 сигнал Ctrl-Brk*/

#define SIGUSR1 16 /*OS/2 A-флаг процесса*/

#define SIGUSR2 17 /*OS/2 B-флаг процесса*/

#define SIGUSR3 20 /*OS/2 C-флаг процесса*/

SIGINT соответствует сигналу прерывания INT23H.

SIGFPE соответствует исключительным ситуациям с плавающей точкой, которые не замаскированы, такие как переполнение, деление на ноль, недействительная операция.

Значение аргумента *func* должно быть равно одной из констант SIG_DFL или SIG_IGN (также определенных в *signal.h*) или же определять адрес новой функции обработки прерывания.

#define SIG_DFL((void(*_Cdecl)(int))0) /*по умолчанию*/

#define SIG_IGN((void(*_Cdecl)(int))1) /*игнорировать*/

Для системы программирования TC версии 2.0 определена также константа

#define SIG_ERR((void(*_Cdecl)(int))-1) /*сгенерировать ошибку*/

Действия при получении сигнала зависят от значения *func* следующим образом:

SIG_IGN — сигнал прерывания игнорируется. Это значение никогда не должно задаваться для сигнала SIGFPE, т.к. состояние процесса с плавающей точкой становится неопределенным.

SIG_DFL — вызванный процесс завершается при появлении сигнала и управление передается в MS-DOS. Все файлы, открытые процессом, закрываются, но буфера не скидываются.

Адрес для сигнала SIGINT: функция, на которую указывает *func*, получает аргумент SIGINT и выполняется. Если функция возвращает управление, вызванный процесс возобновляет выполнение непосредственно за той точкой, в которой был принят сигнал прерывания.

Перед тем как определяемая функция выполнится, значение *func* устанавливается в SIG_DEL; следующий сигнал прерывания трактуется, как описано выше для SIG_DEL, если при обработке реакция на сигнал не переопределена вновь. Это позволяет пользователю устанавливать сигналы в вызванной функции по необходимости.

Для SIGFPE: функция, на которую указывает *func*, получает два аргумента, SIGFPE и код ошибки FPE_*xxx*, и выполняется. (Смотри включаемый файл *float.h* для описания кодов FPE_*xxx*.)

Значение *func* вновь не устанавливается для последующих обработок сигнала. Чтобы вернуться из функции обработки исключительной ситуации при выполнении операции с плавающей точкой, удобно использовать функции **setjmp** и **longjmp**.

(Смотри пример для функции **_fpreset**).

процесса для плавающей точки.

Замечание. Заданная реакция на сигналы не сохраняется в порождаемых процессах, созданных через вызов функций **exec** или **spawn**. В порожденном процессе устанавливается реакция по умолчанию.

Замечание. В системе программирования TC версии 1.5 для этих же целей используется функция **ssignal**.

Возвращаемое значение

Предыдущее значение *func*.

Значение -1, если задано недействительное значение параметра *sig*; переменной *errno* присваивается значение EINVAL.

Смотри также

gsignal, ssignal, abort, exit, _exit, _fpreset, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

SIN, SINH (TC & MSC & ANSI)

#include <math.h>

double sin(x);

double sinh(x);

double x;

Описание

Функции **sin** и **sinh** возвращают синус и гиперболический синус вещественного аргумента *x* соответственно.

Возвращаемое значение

Функция **sin** возвращает значение синуса *x*.

(СП MSC функция **sin**) Если *x* большое и произошла частичная утрата значения в результате, генерируется ошибка PLOSS, но сообщение об ошибке не печатается. Если *x* такое большое, что произошла полная утрата значения в результате, то печатается сообщение об ошибке TLOSS в поток *stderr* и возвращается значение 0. В обоих случаях переменной *errno* присваивается значение ERANGE.

Функция **sinh** возвращает значение гиперболического синуса *x*. Если результат большой, то переменной *errno* присваивается значение ERANGE и возвращается значение HUGE (положительное или отрицательное, в зависимости от значения *x*).

Обработку ошибок можно изменить, используя функцию **matherr**.

Смотри также

acos, asin, atan, atan2, cos, cosh, tan, tanh

SLEEP (TC)

#include <dos.h>

void sleep(unsigned seconds);

Описание

Функция приостанавливает выполнение программы на *seconds* секунд.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

delay

SOPEN (TC 2.0 & MSC)

#include <fcntl.h>

#include <sys\types.h>

#include <sys\stat.h>

#include <share.h>

#include <io.h>

int sopen(pathname, oflag, shflag, pmode);

*char *pathname;*

int oflag;

int shflag;

int pmode;

Описание

Функция **sopen** открывает файл, имя которого задается строкой, адрес которой определяется значением параметра *pathname*, и подготавливает файл для чтения или записи, как определено аргументами *oflag*, *shflag* и *pmode*.

Значение аргумента *oflag* является целым значением, сформированным комбинацией одной или более констант, определенных в файле *fcntl.h*; когда задается более одной константы, они разделяются операциями ИЛИ (|).

O_APPEND — перемещать указатель файла на конец файла перед каждой операцией записи.

O_CREAT — создать и открыть новый файл для записи; не дает результата, если файл уже существует.

O_EXCL — возвращает ошибочный код, если файл, определенный в *pathname*, уже существует.

Применяется только вместе с O_CREAT.

O_RDONLY — открыть файл только для чтения; если этот флаг задан, ни O_RDWR, ни O_WRONLY не могут быть заданы.

O_RDWR — открыть файл для чтения и записи; если этот флаг задан, ни O_RDONLY, ни O_WRONLY не могут быть заданы.

O_TRUNC — открыть и сузить существующий файл до нулевой длины; файл должен иметь

доступ для записи. Содержимое файла теряется.

O_WRONLY — открыть файл только для записи; если этот флаг задан, ни O_RDONLY, ни O_RDWR не могут быть заданы.

O_BINARY — открыть файл в двоичном режиме (см.описание функции **fopen**).

O_TEXT — открыть файл в текстовом режиме (см.описание функции **fopen**).

Замечание. O_TRUNC полностью затирает содержимое существующего файла. Будьте с ним осторожны.

Аргумент *shflag* является константным выражением, состоящим из одной или более констант, определенных в *share.h*. Смотри документацию по MS-DOS для получения подробной информации о значении используемых режимов.

SH_COMPAT Установить возможность разделения файла процессами.

SH_DENYRW Запретить доступ к файлу для чтения и записи.

SH_DENYWR Запретить доступ для записи.

SH_DENYRD Запретить доступ для чтения.

SH_DENYNO Разрешить доступ для чтения и записи.

Аргумент *pmode* требуется только тогда, когда задано O_CREAT. Если файл существует, значение *pmode* игнорируется. Иначе, *pmode* определяет способ доступа к файлу, который устанавливается, когда новый файл закрывается впервые. Значение *pmode* есть целое значение, содержащее одну или обе константы S_IWRITE и S_IREAD, определенных в файле *sys\stat.h*. Когда определяются обе константы, они разделяются оператором ИЛИ(|).

Действие *pmode*:

S_WRITE Доступ для записи.

S_READ Доступ для чтения.

S_READ|S_WRITE Доступ для чтения и записи.

Если доступ для записи не задан, файл открыт только для чтения.

Под MS-DOS все файлы доступны для чтения; поэтому нет необходимости задавать только доступ для записи. Таким образом, режимы S_WRITE и S_READ|S_WRITE эквивалентны.

Замечание. Функция **sopen** должна быть использована только под MS-DOS версий 3.0 и позже. Под ранними версиями аргумент *shflag* игнорируется.

Замечание (TC). **open**(path, access, shflag, mode) реализуется как макрос open(path, ((access)|(shflag)), mode)

Возвращаемое значение

Дескриптор (*handle*) для открытого файла.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES Заданное имя *pathname* является каталогом, или попытка открыть для записи файл, который доступен только для чтения, или встретилось нарушение распределения.

EINVAL SHARE.COM — не задано, (только для СП MSC)

EEXIST — определены флаги O_CREAT и O_EXECL, но файл уже существует. (Только для СП MSC).

EMFILE — нет больше доступных дескрипторов *handle* (открыто максимальное возможное число файлов).

ENOENT — файл или каталог на пути к нему не найдены.

Смотри также

access, chmod, close, creat, dup, dup2, fopen, open, umask

SOUND (TC 2.0)

#include <dos.h>

void sound(unsigned freq);

Описание

Функция **sound** включает устройство подачи звукового сигнала, установив ему частоту *freq* (в Гц),

Замечание. Соответствующая функция существует в системе программирования Turbo-Pascal.

Замечание. Чтобы совсем отключить устройство подачи звукового сигнала, используйте функцию **nosound**.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

delay, nosound

SPAWNL, SPAWNLE, SPAWNLP, SPAWNLPE, SPAWNV, SPAWNVE, SPAWNVP, SPAWNVPE (TC & MSC)

#include <stdio.h>

#include <process.h>

int spawnl (modeflag, pathname, arg0, arg1..., argn, NULL);

int spawnle (modeflag, pathname, arg0, arg1..., argn, NULL, envp);

int spawnlp (modeflag, pathname, arg0, argl., argn, NULL);

int spawnlpe (modeflag, pathname, arg0, arg1..., argn, NULL, envp);

int spawnv (modeflag, pathname, argv);

int spawnve (modeflag, pathname, argv, envp);

int spawnvp (modeflag, pathname, argv);

int spawnvpe (modeflag, pathname, argv, envp);

int modeflag;

*char *pathname;*

*char *arg0,*arg1...,*argn;*

*char *argv[];*

*char *envp[]; ,*

Описание

Функции **spawn** создают и запускают на выполнение новые (под)процессы.

Должно быть доступно достаточное место в памяти для загрузки и выполнения порождаемого процесса.

Аргумент *modeflag* определяет, что будет происходить с вызвавшим процессом (процессом-отцом) после запуска подпроцесса (процесса-сына).

Следующие значения для аргумента *modeflag* определены в файле *process.h*:

P_WAIT — приостановить процесс-родитель, пока выполнение порождаемого процесса не завершится.

P_NOWAIT — продолжать выполнять процесс-родитель вместе с порождаемым процессом (не реализовано в MS-DOS).

P_OVERLAY — наложить порождаемый процесс на процесс-родитель, затерев родитель (такой же эффект, как при вызове *exec*). Только значения P_WAIT и P_OVERLAY могут быть использованы в настоящее время.

Значение P_NOWAIT зарезервировано для будущих расширений. Использование значения P_NOWAIT вызовет ошибку.

Значение аргумента *pathname* задает адрес строки, определяющей имя файла, который будет выполняться как порождаемый процесс. *Pathname* может определять полный путь (с корня), частичный путь (с текущего каталога) или просто имя файла.

Если попытка найти файл окончится неуспешно, то добавится расширение .EXE и будет предпринята еще одна попытка открыть файл. Если *pathname* содержит расширение, то вторая попытка не предпринимается. Если *pathname* заканчивается точкой, то поиск ведется для *pathname* без расширения.

Функции **spawnlp**, **spawnlpe**, **spawnvp** и **spawnvpe** ведут поиск файла по *pathname* (используя стандартные процедуры) в каталогах, определяемых в переменной окружения PATH.

Аргументы передаются порождаемому процессу заданием одного или более указателей на символьные строки как аргументы при *spawn*-вызове. Эти символьные строки формируют список аргументов для порождаемого процесса.

Сумма длин строк, формирующих список аргументов, не должна превышать 128 байтов. Нулевой символ ('\0') в конце каждой строки не учитывается, но символы-пробелы (' '), разделяющие аргументы, учитываются.

Аргументы-указатели могут быть переданы как отдельные аргументы (*spawnl*, *spawnle*, *spawnlp*, *spawnlpe*) или как массив указателей (*spawnv*, *spawnve*, *spawnvp*, *spawnvpe*).

Минимум один аргумент, *arg0* или *argv[0]*, должен быть передан порождаемому процессу. По соглашению, этот аргумент является копией *pathname*, хотя и отличное значение не вызовет ошибочной ситуации.

Под MS-DOS версий ниже 3.0 передаваемое значение *arg0* или *argv[0]* недоступно для порождаемого процесса. Под MS-DOS версий 3.0 и выше доступно.

Вызов функций **spawnl**, **spawnle**, **spawnlp**, **spawnlpe** обычно используется, когда число аргументов известно заранее. Аргумент *arg0* обычно является указателем на символьные строки, формирующие новый список аргументов. Следующим за *argn* должен быть NULL-указатель, означающий конец списка аргументов.

Вызовы **spawnv**, **spawnvp**, **spawnve**, **spawnvpe** обычно используются, когда число аргументов переменно. Указатели на аргументы передаются как массив *argv*. Аргумент *argv[0]* обычно является указателем на pathname, а от *argv[1]* до *argv[n]* — указатели на символьные строки, формирующие новый список аргументов. Аргумент *argv[n+1]* должен быть NULL-указатель, означающий конец списка аргументов.

Файлы, которые были открыты перед вызовом **spawn**, остаются открытыми в порождаемом процессе.

При вызове функций **spawnl**, **spawnlp**, **spawnv**, **spawnvp** порождаемый процесс унаследует и окружение родителя.

Вызов функций **spawnle**, **spawnlpe**, **spawnve** или **spawnvpe** позволяет пользователю вносить изменения в окружение порождаемого процесса, передав список окружающих характеристик через аргумент *envp*.

Аргумент *envp* является массивом символьных указателей, каждый элемент которого (за исключением последнего) указывает на строку, завершающуюся нулевым символом ('\0'), определяющую переменную окружения. Такая строка обычно имеет форму:

NAME=value

NAME — имя переменной окружения, а *value* является строковым значением, в которое переменная устанавливается. (Заметим, что *value* не заключается в двойные кавычки.) Последним элементом массива *envp* должен быть NULL. Когда *envp* содержит только NULL, порождаемый процесс унаследует окружение от родителя.

Возвращаемое значение

Статус завершения порождаемого процесса.

Статус завершения равен 0, если процесс завершится нормально.

Статус завершения может быть установлен в ненулевое значение, если порождаемый процесс специально вызвал *exit* команду с ненулевым аргументом.

По соглашению, положительный статус завершения сигнализирует о ненормальном завершении: *abort* или прерывание.

Значение -1 сигнализирует об ошибке (порождаемый процесс не стартовал); при этом переменной *errno* присваивается одно из следующих значений:

E2BIG — список аргументов превышает 128 байтов, или область, требуемая для информации окружения, превышает 32 байта.

EINVAL — неверный аргумент *modeflag*.

ENOENT — файл или каталог на пути к нему не найдены.

ENOEXEC — определяемый файл невыполняемый, или имеет формат, неприемлемый для

выполнения.

ENOMEM — нет достаточной доступной памяти, чтобы выполнить порождаемый процесс.

Замечание. **Spawn-**вызовы не наследуют двоичный/текстовый режим работы с открытым файлом. Если порождаемый процесс использует файлы, унаследованные от родителя в режиме, отличном от режима по умолчанию, необходимо воспользоваться функцией **setmode**, чтобы установить требуемый режим работы.

Смотри также

abort, excel, execle, execlp, execlpe, execv, execve, execvp, execvpe, exit, _exit, onexit, system

SPRINTF (TC & MSC & ANSI)

#include <stdio.h>

int sprintf(buffer, format_string[, argument...]);

*char *buffer;*

*char *format_string;*

Описание

Функция **sprintf** форматирует и выводит последовательности символов и значений аргументов в символьный массив, адрес которого задается значением аргумента *buffer*.

Каждый аргумент преобразуется и выводится согласно соответствующей спецификации формата в строке, адрес которой задается значением параметра *format_string*.

Строка описания формата вывода интерпретируется так же, как и для функции **printf**; смотри описание функции **printf**.

Возвращаемое значение

Число символов, помещенных в строку, адрес которой задается значением параметра *buffer*.

Смотри также

fprintf, printf, sscanf

SQRT (TC & MSC & ANSI)

#include <math.h>

double sqrt(x);

double x;

Описание

Функция **sqrt** вычисляет квадратный корень от вещественного аргумента *x*.

Возвращаемое значение

Результат вычисления квадратного корня.

Значение 0, если значение *x* отрицательное; при этом переменной *errno* присваивается значение EDOM и печатается сообщение об ошибке DOMAIN. Обработку ошибки можно изменить, используя функцию **matherr**.

Смотри также

exp, log, matherr, pow

SRAND (TC & MSC & ANSI)

#include <stdlib.h>

void srand(seed);

unsigned seed;

Описание

Функция **srand** устанавливает начальное значение для генератора псевдослучайных целых чисел.

Рекомендуется использовать значение 1 в качестве аргумента для переинициализации генератора. При других значениях для *seed* устанавливается случайная стартовая точка.

Для получения случайных чисел используется функция **rand**.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

rand

SSCANF (TC & MSC & ANSI)

#include <stdio.h>

int sscanf(buffer, format_string[, argument...]);

*char *buffer;*

*char *format_string;*

Описание

Функция **sscanf** читает данные из символьного массива, адрес которого задается значением параметра *buffer*, в переменные, адреса которых задаются аргументами *arguments*.

Каждый аргумент должен быть указателем на переменную с типом, соответствующим типу, определенному в строке описания формата, адрес которой задается значением параметра *format_string*.

Строка описания формата управляет интерпретацией входных полей и имеет тот же вид, что и для функции **scanf**.

Смотри описание функции **scanf**.

Возвращаемое значение

Число полей, которые были успешно преобразованы и присвоены. Возвращаемое значение не включает поля, которые были прочитаны, но не присвоены.

Значение EOF, если была попытка прочитать символ конца строки.

Значение 0, если не было присвоенных полей.

Смотри также

fscanf, scanf, sprintf

SSIGNAL (TC 1.5)

#include <signal.h>

*int(*ssignal(sig, func))();*

int sig;

*int (*func)();*

Описание

Функция **ssignal** позволяет процессу выбрать одну из возможностей для управления сигналами прерывания от операционной системы.

Значение аргумента *sig* определяет, для какого сигнала задается реакция. Номер сигнала — это целое число в диапазоне от 1 до 15.

Аргумент *func* должен иметь значение одной из констант SIG_DEL или SIG_IGN (определенных в *signal.h*) или быть адресом функции.

Действия при получении сигнала зависят от значения *func* следующим образом:

SIG_IGN Сигнал прерывания игнорируется. Это значение никогда не должно задаваться для SIGFPE, т.к. состояние процесса с плавающей точкой становится неопределенным.

SIG_DFL Вызванный процесс завершается, и управление передается в MS_DOS. Все файлы, открытые процессом, закрываются, но буфера не скидываются.

Адрес для сигнала SIGINT: функция, на которую указывает *func*, получает аргумент SIGINT и выполняется. Если функция возвращает управление, вызванный процесс возобновляет выполнение непосредственно за той точкой, в которой был принят сигнал прерывания.

Замечание. В системе программирования TC версии 2.0 и в системе программирования MSC для тех же целей используется функция **signal**.

Возвращаемое значение

Возвращается предыдущее значение *func* для данного сигнала *sig*.

Если номер сигнала указан неверно, возвращается значение SIG_DFL.

Смотри также

gsignal, signal, abort, exit, _exit, _fpreset, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

STACKAVAIL (MSC)

#include <malloc.h> /*используется только для описания функции*/

unsigned int stackavail();

Описание

Функция **stackavail** возвращает приблизительный размер памяти, доступной для динамического распределения через функцию **alloca** (в программном стеке), в байтах.

Возвращаемое значение

Размер в байтах, как беззнаковое целое значение.

Смотри также

alloca, freed, memavl

STAT (TC & MSC)

#include <sys\types.h>

#include <sys\stat.h>

int stat(pathname, buffer);

*char *pathname;*

*struct stat *buffer;*

Описание

Функция **stat** получает информацию о файлах и каталогах, которые определяются именем, хранящимся в строке, адрес которой задается значением параметра *pathname*.

Получаемая информация записывается в структуру, адрес которой задается значением параметра *buffer*.

Тип структуры *stat*, определенный в файле *sys\stat.h*, содержит следующие поля:

struct stat

{

short st_dev;

short st_ino;

short st_mode;

short st_nlink;

int st_uld;

int st_gid;

short st_rdev;

long st_size;

long st_atime;

long st_mtime;

long st_ctime;

};

где:

st_mode — битовая строка, определяющая вид информации в файле. Бит S_IFDIR

устанавливается, если *pathname* определяет каталог; бит S_IFREG устанавливается, если *pathname* определяет обычный файл. Бит S_IWRITE устанавливается, если пользователь имеет право на запись в файл. Бит S_IREAD устанавливается, если пользователь имеет право на чтение из файла;

st_dev — номер устройства, на котором расположен файл;

st_rdev — то же, что и *st_dev*;

st_nlink — всегда 1;

st_size — размер файла в байтах;

st_atime — время последней модификации файла;

st_mtime — то же, что и *st_atime*;

st_ctime — то же, что и *st_atime*.

Остальные поля в структуре *stat* не содержат значимой информации для MS-DOS.

Возвращаемое значение

Значение 0, если информация о статусе файла получена.

Значение -1, если имя файла или имя пути не могут быть найдены; переменной *errno* присваивается значение ENOENT.

Смотри также

access, fstat

_STATUS87 (TC & MSC)

#include <float.h>

unsigned int _status87();

Описание

Функция **_status87** получает слово состояния процессора с плавающей точкой 8087/800287. Также учитывается информация, полученная обработчиком исключительных ситуаций для операций с плавающей точкой.

Таким образом, опрашивается состояние целиком пакета функций работы с плавающей точкой.

Возвращаемое значение

Слово, определяющее состояние библиотеки работы с плавающей точкой.

Значение отдельных битов описано в файле *float.h*. Смотри файл *float.h* для получения полного описания битов в возвращаемом функцией значении.

Смотри также

_clear87, _control87

STIME (TC)

#include <time.h> /*используется только для описания функции*/

int time(timeptr);

*long *timeptr;*

Описание

Функция **stime** устанавливает системное время и дату на основании значения переменной, адрес которой определяется значением параметра *timeptr*, трактуя значение этой переменной как количество секунд, прошедших с момента 00:00:00 по Гринвичу первого января 1970 г.

Возвращаемое значение

Функция возвращает значение 0.

Смотри также

asctime, ftime, gmtime, localtime, utime, time

STPCPY (TC)

#include <string.h> /*используется только для описания функции*/

char stpcpy(string1, string2);

*char *string1;*

*char *string2;*

Описание

Функция **stpcpy** копирует строку, адрес которой задается значением параметра *string2*, включая завершающий нулевой байт ('\0') по адресу, определяемому значением параметра *string1*, и возвращает указатель на последний записанный символ (всегда нулевой байт).

Проверка на переполнение не выполняется.

Функция **stpcpy** идентична **strcpy**, за исключением того, что функция **strcpy** возвращает значение *string1*, а функция **stpcpy** возвращает значение *string1+strlen(string2)*.

Функция **strlen** описана далее.

Возвращаемое значение

Функция возвращает указатель на последний записанный символ (всегда указывает на нулевой байт).

Смотри также

strcpy

STRCAT, STRCHR, STRCMP, STRCPY, STRCSPN (TC & MSC & ANSI), STRDUP, STRICMP (TC & MSC), STRCMPI (MSC)

#include <string.h> /*используется только для описания функции*/

*char *strcat(string1, string2);*

*char *string1;*

*char *string2;*

*char *strchr(string, sim);*

*char *string;*

int sim;

int strcmp(string1, string2);

*char *string1;*

*char *string2;*

int strcmpi(string1, string2);

*char *string1;*

*char *string2;*

char strcpy(string1, string2);

*char *string1;*

*char *string2;*

int strcspn(string1, string2);

*char *string1;*

*char *string2;*

*char *strdup(string);*

*char *string;*

int stricmp(string1, string2);

*char *string1;*

*char *string2;*

Описание

Функции **strcat**, **strchr**, **strcmp**, **strcmpi**, **strcpy**, **strcspn**, **strdup** и **stricmp** оперируют со строками, которые завершаются нулевым символом ('\0').

Проверка на переполнение не выполняется, когда строка копируется или производится дописывание в конец строки.

Функция **strcat** добавляет строку, адрес которой задается значением аргумента *string2*, в конец строки, адрес которой определяется значением аргумента *string1*, записывая в конец строки-результата нулевой символ, и возвращает указатель на сцепленную строку (значение аргумента *string1*).

Функция **strchr** возвращает указатель на первое местонахождение символа, имеющего код *sim*, в строке, адрес которой задается значением аргумента *string*. Символ sim может быть нулевым символом ('\0'); тогда поиск ведется для нулевого символа. Функция возвращает NULL, если символ не найден.

Функция **strcmp** сравнивает строки, адреса которых задаются значениями аргументов *string1* и *string2*, лексографически и возвращает значение, определяющее их соотношение:

меньше 0 string1 меньше, чем string2

0 string1 идентично string2

больше 0 string1 больше, чем string2

Функции **strcmpi** и **stricmp** являются менее чувствительными версиями функции **strcmp** (функция **strcmpi** доступна только в системе программирования MSC). Строки, адреса которых задаются значениями аргументов *string1* и *string2*, сравниваются таким образом, что заглавные и прописные буквы считаются эквивалентными.

Функция **strcpy** копирует строку, адрес которой определяется значением аргумента *string2*, включая завершающий нулевой символ, по адресу, определяемому аргументом *string1*, и возвращает значение аргумента *string1*.

Функция **strcspn** возвращает индекс первого символа в строке, адрес которой задается значением аргумента *string1*, который принадлежит набору символов, содержащихся в строке, адрес которой задается значением аргумента *string2*. Это значение эквивалентно длине начальной подстроки *string1*, которая не содержит ни одного символа из *string2*. Завершающий нулевой символ не учитывается при поиске. Если *string1* начинается с символа из *string2*, то возвращается значение 0.

Функция **strdup** получает область памяти (через вызов функции **malloc**) для копирования строки и возвращает указатель на область памяти, в которую скопирована строка. Возвращается значение NULL, если не удалось выделить память через вызов функции **malloc**.

Возвращаемое значение

Описано выше.

Смотри также

strncat, strncmp, strncpy, strnicmp, strrchr, strspn

STRERROR (TC & MSC & ANSI), _STRERROR (TC)

Использование (MSC)

#include <string.h> /*используется только для описания функции*/

*char *strerror(string);*

*char *string;*

int errno;

int sys_nerr;

char sys_errlist[sys_nerr];

Использование (TC)

#include <string.h>

#include <stdio.h>

*char *_strerror(string);*

*char *string;*

*char *strerror(numerr);*

int numerr;

Описание (функции strerror для СП MSC и функции _strerror для СП TC)

Если значение параметра *string* равно NULL, функция возвращает указатель на строку, которая содержит системное сообщение об ошибке для последнего вызова библиотечной функции, во время которого произошла ошибка; в конце строки стоит символ новой строки ('\n').

Если значение параметра *string* не равно NULL, то функция возвращает указатель на строку, содержащую, по порядку: строку, адрес которой задается значением параметра *string*, двоеточие, пробел, системное ошибочное сообщение для последнего вызова библиотечной функции, во время которого произошла ошибка; в конце строки стоит символ новой строки ('\n').

Строка, адрес которой определяется значением параметра *string*, должна иметь длину не более 94 байтов.

В отличие от функции **perror**, описываемая функция сама не печатает никаких сообщений.

Чтобы напечатать сообщение, возвращаемое функцией **strerror**, в поток *stdout*, необходимо вызвать **printf**:

```
if((access("datafile",2))==-1)

printf(strerror(NULL));
```

Действительный код ошибки помещается в переменную *errno*, которая описывается на внешнем уровне (пользователь не должен заботиться о ее описании)

К тексту системного сообщения об ошибке можно получить доступ также через переменную *sys_errlist*, которая является массивом сообщений, упорядоченным по кодам ошибок.

Функция **strerror** обеспечивает доступ к нужному ошибочному сообщению, используя значение *errno* как индекс для *sys_errlist*.

Значение переменной *sys_nerr* определяется как максимальное число элементов в массиве *sys_errlist*.

Чтобы результат был правильным, функция **strerror** должна быть вызвана непосредственно за библиотечной командой, которая вернула код возврата, сигнализирующий об ошибке. Иначе значение *errno* может быть затерто последующими вызовами.

Замечание. Под ОС MS-DOS некоторые *errno* значения из списка *errno.h* не используются.

Функция **strerror** печатает пустую строку для тех значений *errno*, которые не используются под ОС MS-DOS.

Описание (функции strerror для СП TC)

Функция возвращает указатель на строку, которая содержит системное ошибочное сообщение, соответствующее коду ошибки *numerr*.

Замечание. В системе программирования TC версии 1.0 функция **strerror** была определена так же, как и в системе программирования MSC.

Возвращаемое значение

Описано выше.

Смотри также

clearerr, ferror, perror

STRLEN (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

int strlen(string);

*char *string;*

Описание

Функция **strlen** возвращает длину в байтах строки, адрес которой определяется значением параметра *string*, не учитывая завершающий нулевой символ ('\0').

Возвращаемое значение

Длина строки.

Нет ошибочных кодов возврата.

STRLWR (TC & MSC)

#include <string.h> /*используется только для описания функции*/

char strlwr(string);

*char *string;*

Описание

Функция **strlwr** преобразует заглавные буквы в строке, адрес которой определяется значением параметра *string*, в строчные. Строка завершается нулевым символом ('\0').

Возвращаемое значение

Указатель на преобразованную строку (значение параметра *string*).

Нет ошибочных кодов возврата.

Смотри также

strupr

STRNCAT, STRNCMP, STRNCPY (TC & MSC & ANSI), STRNSET (TC & MSC), STRNICMP

#include <string.h> /*используется только для описания функции*/

*char *strncat(string1, string2, n);*

*char *string1;*

*char *string2;*

unsigned int n;

int strncmp(string1, string2, n);

*char *string1;*

*char *string2;*

unsigned int n;

int strnicmp(string1, string2, n);

*char *string1;*

*char *string2;*

unsigned int n;

*char *strncpy(string1, string2, n);*

*char *string1;*

*char *string2;*

unsigned int n;

*char *strnset(string, sim, n);*

*char *string;*

int sim;

unsigned int n;

Описание

Функции **strncat**, **strncmp**, **strnicmp**, **strncpy**, **strnset** выполняют операции над строками, рассматривая, самое большее, первые *n* символов или до завершающего строку нулевого символа ('\0').

Функция **strncat** добавляет (дописывает), самое большее, первые *n* символов из строки, адрес которой определяется значением параметра *string2*, в строку, адрес которой определяется значением параметра *string1*, завершая результирующую строку нулевым символом ('\0').

Функция **strncat** возвращает указатель на сцепленную строку (значение параметра *string1*). Если *n* больше длины строки *string2*, то длина строки *string2* используется вместо *n*.

Функция **strncmp** сравнивает, самое большее, первые *n* символов в строках, адреса которых определяются значениями параметров *string1* и *string2*, лексографически и возвращает значение, определяющее соотношение между строками:

значение меньше 0 *string1* меньше, чем *string2*

значение 0 *string1* идентична *string2*

значение больше 0 *string1* больше, чем *string2*.

Функция **strnicmp** является менее чувствительной версией функции **strncmp**: при сравнении заглавные и строчные буквы считаются эквивалентными.

Функция **strncpy** копирует точно *n* символов строки, адрес которой определяется значением параметра *string2*, в строку, адрес которой определяется значением параметра *string1*.

Функция **strncpy** возвращает значение параметра *string1* (адрес строки-результата). Если значение *n* меньше, чем длина строки *string2*, нулевой символ ('\0') не добавляется автоматически в новую строку. Если значение *n* больше, чем длина строки *string2*, то в строку-результат добавляется в конец нулевой символ ('\0').

Функция **strnset** присваивает не более чем первым *n* символам строки, адрес которой задается

возвращает указатель на измененную строку *string*. Если значение *n* больше, чем длина строки *string*, то длина строки *string* используется вместо *n*.

Возвращаемое значение

Описано выше.

Смотри также

strcat, strcmp, strcpy, strset

STRPBRK (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

*char *strpbrk(string1, string2);*

*char *string1;*

*char *string2;*

Описание

Функция **strpbrk** находит первое вхождение в строке, адрес которой задается значением параметра *string1*, любого символа из набора символов, содержащихся в строке, адрес которой задается значением параметра *string2*. Завершающий нулевой символ ('\0') не включается в поиск.

Возвращаемое значение

Указатель на первое местоположение любого символа из *string2* в *string1*. Значение NULL, если нет общих символов.

Смотри также

strchr, strrchr

STRRCHR (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

*char *strrchr(string, sim);*

*char *string;*

int sim;

Описание

Функция **strrchr** находит последнее вхождение символа *sim* в строке, адрес которой задается значением параметра *string*. Завершающий нулевой символ включается в поиск.

Возвращаемое значение

Указатель на последнее местоположение символа *sim* в строке *string*.

Значение NULL, если заданный символ не найден в строке.

Смотри также

strchr, strpbrk

STRREV (TC & MSC)

#include <string.h> /*используется только для описания функции*/

*char *strrev(string);*

*char *string;*

Описание

Функция **strrev** переписывает заново строку, адрес которой задается значением параметра *string*, меняя порядок следования символов в строке на противоположный. Завершающий нулевой символ ('\0') остается на месте.

Возвращаемое значение

Указатель на измененную строку (копия значения параметра *string*).

Смотри также

strcpy, strset

STRSET (TC & MSC)

#include <string.h> /*используется только для описания функции*/

*char *strset(string, sim);*

*char *string;*

int sim;

Описание

Функция **strset** присваивает новое значение всем символам строки, адрес которой задается значением параметра *string*, за исключением завершающего нулевого символа ('\0'). Новое значение задается параметром *sim*.

Возвращаемое значение

Указатель на измененную строку (копия значения параметра *string*).

Смотри также

strnset

STRSPN (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

int strspn(string1, string2);

*char *string1;*

*char *string2;*

Описание

Функция **strspn** возвращает индекс первого символа строке, адрес которой задается значением параметра *string1*, который не принадлежит набору символов, содержащихся в строке, адрес которой задается значением параметра *string2*.

Это значение эквивалентно длине начальной подстроки строки *string1*, которая содержит только символы из набора символов в строке *string2*.

Нулевой символ, завершающий строку *string2*, не рассматривается. Если строка *string1* начинается с символа не из набора *string2*, функция **strspn** возвращает значение 0.

Возвращаемое значение

Значение, определяющее позицию первого символа в *string1*, не входящего в набор, определяемый *string2*.

Смотри также

strcspn

STRSTR (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

*char *strstr(string1, string2);*

*char *string1;*

*char *string2;*

Описание

Функция **strstr** возвращает указатель на первое вхождение подстроки, которая содержится в символьном массиве, адрес которого задается значением параметра *string2*, в строке, адрес которой определяется значением параметра *string1*.

Возвращаемое значение

Указатель по строке *string1*, на символ, с которого начинается первое вхождение подстроки *string2* в строке *string1*.

Значение NULL, если вхождение не найдено.

Смотри также

strcspn

STRTOD, STRTOL (TC & MSC & ANSI)

#include <stdlib.h>

double strtod(nptr, endptr);

*char *nptr;*

*char **endptr;*

long strtol(nptr, endptr, base);

*char *nptr;*

*char **endptr;*

int base;

Описание

Функции **strtod** и **strtol** преобразуют строку символов в значение типа *double* или *long int* соответственно.

Строка символов является последовательностью символов, которая может быть интерпретирована как числовое значение определенного типа. Эти функции прекращают чтение строки с первого символа, который они не могут распознать как часть числа (это может быть и нулевой символ ('\0'), завершающий строку).

Для функции **strtol** этим последним символом может быть также первый символ-цифра, код которого больше или равен коду '0'+*base*.

Если значение параметра *endptr* не равно NULL, то оно определяет указатель, который содержит адрес символа, на котором надо прекратить преобразование (просмотр строки).

Функция **strtod** ожидает, что значение параметра *nptr* — это адрес строки следующего вида:

[пробелы][знак][цифры][.цифры][{d|D|e|E}[знак]цифры]

Первый символ, который не соответствует этой форме, прекращает дальнейший просмотр строки.

Функция **strtol** ожидает, что значение параметра *nptr* — это адрес строки следующего вида:

[пробелы][знак][0][x][цифры]

Если значение параметра *base* лежит между 2 и 36, то оно используется как основание для преобразования числа (как основание системы исчисления, в которой записано число в строке).

Если значение параметра *base* равно 0, то начальные символы строки, на которую указывает nptr, используются, чтобы определить основание: если первый символ есть '0' и второй символ есть цифры '1'-'7', то строка интерпретируется как восьмеричное целое; если первый символ есть '0' и второй символ есть '*x*' или '*X*', то строка интерпретируется как шестнадцатеричное целое; если первый символ есть '1'-'9', то строка интерпретируется как десятичное целое.

Возвращаемое значение

Функция **strtod** возвращает значение числа с плавающей точкой.

Функция **strtod** возвращает значение HUGE, если произошло переполнение.

Функция **strtol** возвращает значение типа *long*, значение 0 в случае ошибки.

В обоих случаях при ошибке переменной *errno* присваивается значение ERANGE.

Смотри также

atof, atol, strtoul

STRTOK (TC & MSC & ANSI)

#include <string.h> /*используется только для описания функции*/

*char *strtok(strtng1, string2);*

*char *string1;*

*char *string2;*

Описание

Функция **strtok** принимает параметр *string1*, задающий адрес строки из нуля или более символов,

и параметр *string2*, задающий адрес строки, рассматриваемой как набор символов-разделителей для строки *string1*.

Символы из *string1* при вызове **strtok** группируются в слова.

При первом вызове **strtok** для заданного значения параметра *string1* производится возврат адреса первого символа *string1*. Чтобы найти начало следующего слова в *string1*, необходимо вызвать функцию **strtok** с NULL-значением аргумента *string1*.

Набор разделителей может различаться от вызова к вызову, так что *string2* может иметь любые значения.

Замечание. Вызов **strtok** будет модифицировать строку *string1* при каждом вызове, при этом вставляется нулевой символ ('\0') после очередного слова в строке *string1*.

Возвращаемое значение

При первом вызове **strtok** она возвращает указатель на первый символ в *string1*. При последующем вызове, с NULL-значением параметра *string1*, функция **strtok** возвращает указатель на следующее слово в строке.

Значение NULL возвращается, когда нет более слов. Все слова завершаются нулевым символом.

Смотри также

strcspn, strspn

STRTOUL (TC 2.0 & ANSI)

#include <stdlib.h>

unsigned long strtoul(nptr, endptr, base);

*char *nptr;*

*char **endptr;*

int base;

Описание

Функция **strtoul** подобна функции **strtol** и преобразует строку символов в значение типа *unsigned long*.

Строка символов является последовательностью символов, которая может быть интерпретирована как числовое значение определенного типа. Прекращается чтение строки с первого символа, который они не могут распознать как часть числа (это может быть и нулевой символ ('\0'), завершающий строку).

Для функции **strtoul** этим последним символом может быть также первый символ-цифра, код которого больше или равен коду '0'+*base*.

Если значение параметра *endptr* не равно NULL, то значение параметра *endptr* определяет адрес указателя, содержащего адрес символа, на котором надо прекратить преобразование (просмотр строки).

Функция **strtoul** ожидает, что значение параметра *nptr* — это адрес строки следующего вида:

[пробелы][0][x][цифры]

Если значение параметра *base* лежит между 2 и 36, то оно используется как основание для преобразования числа (как основание системы исчисления, в которой записано число в строке).

Если значение параметра *base* равно 0, то начальные символы строки, на которую указывает *nptr*, используются, чтобы определить основание: если первый символ есть '0' и второй символ есть цифры '1'-'7', то строка интерпретируется как восьмеричное целое; если первый символ есть '0' и второй символ есть '*x*' или '*X*', то строка интерпретируется как шестнадцатеричное целое; если первый символ есть '1'-'9', то строка интерпретируется как десятичное целое.

Возвращаемое значение

Функция **strtoul** возвращает значение типа *unsigned long*, значение 0 в случае ошибки, при этом переменной *errno* присваивается значение ERANGE.

Смотри также

atol, strtol

STRUPR (TC & MSC)

#include <string.h> /*используется только для описания функции*/

*char *strupr(string);*

*char *string;*

Описание

Функция **strupr** преобразует строчные буквы в строке, адрес которой определяется значением параметра *string*, в заглавные. Строка завершается нулевым символом ('\0').

Возвращаемое значение

Указатель на преобразованную строку (значение параметра *string*).

Нет ошибочных кодов возврата.

Смотри также

strlwr

SWAB (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

void swab(source, destination, n);

*char *source;*

*char *destination;*

int n;

Описание

Функция **swab** копирует *n* байтов из области памяти, адрес которой определяется значением параметра *source*, меняя местами каждую пару смежных байтов, в область памяти, адрес которой задается значением параметра *destination*. Целое значение *n* задает количество байтов для пересылки (копирования).

Функция **swab** обычно используется для преобразования форматов данных от одной ЭВМ к другой, когда используется различный порядок следования байтов в слове.

Возвращаемое значение

Функция **swab** не возвращает значения.

Смотри также

fgetc, fputc

SYSTEM (TC & MSC & ANSI)

#include <process.h> /*используется только для описания функции*/

#include <stdlib.h> /*используется либо файл process.h, либо файл stdlib.h*/

int system(string);

*char *string;*

Описание

Функция **system** рассматривает строку, адрес которой задается значением параметра *string*, как директиву для командного интерпретатора ОС и выполняет строку как команду MS-DOS.

Функция **system** посылает значения переменных окружения COMSPEC и PATH процессу MS-DOS COMMAND.COM, который используется, чтобы выполнить *string*-команду.

Возвращаемое значение

Значение 0, если команда успешно выполнена.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

E2BIG — список аргументов для команды превышает 128 байтов, или область, требуемая для информации окружения, превышает 32 байта.

ENOENT — файл COMMAND.COM не найден.

ENOEXEC — файл COMMAND.COM имеет неправильный формат и невыполняем.

ENOMEM — нет достаточной доступной памяти для выполнения команды; или доступная память испорчена (существуют неправильные блоки, означающие, что процесс, который сделал вызов, был неправильно расположен).

Смотри также

execl, execle, execlp, execv, execve, execvp, exit, exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

TAN, TANH (TC & MSC & ANSI)

#include <math.h>

double tan(x);

double tanh(x);

double x;

Описание

Функции **tan** и **tanh** возвращают значение тангенса и гиперболического тангенса соответственно от значения аргумента *x*.

Возвращаемое значение

Функция **tan** возвращает значение тангенса от значения аргумента *x*.

Если значение *x* близко *pi/2* или *-pi/2*, происходит частичная утрата результата и переменной *errno* присваивается значение ERANGE.

Если значение *x* так близко *pi/2* или *-pi/2*, что происходит утрата результата, то функция возвращает 0, переменной *errno* присваивается значение ERANGE и печатается ошибочное сообщение TLOSS.

Функция **tanh** возвращает гиперболический тангенс от *x*.

Смотри также

acos, asin, atan, atan2, cos, cosh, sin, sinh, matherr

TELL (TC & MSC)

#include <io.h> /*используется только для описания функции*/

long tell(handle);

int handle;

Описание

Функция **tell** используется для получения текущей позиции (внутреннего) указателя файла, связанного с дескриптором *handle* (ввод/вывод низкого уровня). Позиция представляется как смещение в байтах от начала файла.

Возвращаемое значение

Смещение до текущей позиции в файле, как длинное целое.

Значение -1L, если задано недействительное значение аргумента *handle*; при этом переменной *errno* присваивается значение EBADF.

Для устройств типа терминала и принтера возвращаемое значение не определено.

Смотри также

ftell, lseek, fseek, fgetpos

TEMPNAM (MSC), TMPNAM (TC 2.0 & MSC & ANSI)

#include <stdio.h>

*char *tmpnam(string);* /*функция доступна в системе программирования MSC*/

*char *string;*

*char *tempnam(dir, prefix);* /*функция доступна в системе программирования MSC и в системе

программирования TC версии 2.0*/

*char *dir;*

*char *prefix;*

Описание

Функция **tmpnam** генерирует имя временного файла, который будет использоваться как временный файл. Это имя помещается в *string*.

Если значение аргумента *string* равно NULL, то память для строки выделяется (в функции **tmpnam**) с использованием функции **malloc**. В случае использования **malloc** ответственность за последующее освобождение памяти возлагается на пользователя.

Строка символов, которую генерирует функция **tmpnam**, состоит из цифровых символов от '0' до '9'; числовое значение, представляемое строкой, может быть от 1 до 65535 (это означает, что к функции можно обратиться 65535 раз).

Функция **tempnam** позволяет создать пользователю временный файл в другом каталоге. *Prefix* является префиксом к имени файла. Функция просматривает файл с заданным именем в каталогах в следующем порядке:

Условие	***Используемый каталог для tempnam***
Переменная окружения TMP установлена, и каталог, определенный в TMP, существует	Каталог, определенный в TMP
Переменная окружения TMP не установлена или каталог, определенный в TMP, не существует	Аргумент *dir* для **tempnam**
Значение аргумента *dir* равно NULL, или строка, адрес которой определяется значением аргумента *dir*, задает имя несуществующего каталога	*P_tmpdir* в *stdio.h*
P_tmpnam не существует	*\tmp*

Если все попытки неудачны, то функция tempnam возвращает значение NULL.

Возвращаемое значение

Обе функции возвращают указатель на сгенерированное имя, за исключением случая, когда нет возможности создать это имя, или имя является не уникальным.

Значение NULL, если имя не может быть создано или уже существует.

Смотри также

tmpfile

TIME (TC & MSC & ANSI)

#include <time.h> /*используется только для описания функции*/

long time(timeptr);

*long *timeptr;*

Описание

Функция **time** возвращает число секунд, прошедших с момента 00:00:00 по Гринвичу 1 января 1970 г., согласно часам системы.

timeptr, значение аргумента *timeptr* может быть равным NULL, в этом случае запись возвращаемого значения не производится.

Возвращаемое значение

Число прошедших секунд как значение типа *long*.

Нет ошибочных кодов возврата.

Смотри также

asctime, ftime, gmtime, localtime, utime, ctime, difftime, gettime, sttime, stime

TMPFILE (TC 2.0 & MSC & ANSI)

#include <stdio.h>

*FILE *tmpfile();*

Описание

Функция **tmpfile** открывает (создавая при этом) новый временный файл и возвращает значение указателя потока для этого файла (высокоуровневый ввод/вывод).

Если файл не может быть открыт, функция **tmpfile** возвращает значение NULL.

Создаваемый функцией **tmpfile** временный файл автоматически удаляется, когда происходит нормальное завершение программы или же когда вызывается функция **rmtmp**, при условии, что текущий рабочий каталог не изменился.

Временный файл открывается в режиме "*w+*" (смотри описание функции **fopen**).

Возвращаемое значение

Указатель на поток, связанный с открытым временным файлом.

Значение NULL, если файл не может быть открыт.

Смотри также

tmpnam, tempnam, rmtmp

TOASCII, _TOLOWER, _TOUPPER (TC & MSC), TOLOWER, TOUPPER (TC & MSC & ANSI)

#include <ctype.h>

int toascii(sim);

int tolower(sim);

int _tolower(sim);

int toupper(sim);

int _toupper(sim);

int sim;

Описание

Макросы/функции **toascii**, **tolower**, **_tolower**, **toupper** и **_toupper** преобразуют одиночный символ, код которого задается значением аргумента *sim*, по различным правилам.

Макро **toascii** обнуляет старший 7-й бит значения аргумента *sim*, таким образом преобразованное значение представляет код символа в наборе символов ASCII. Если значение *sim* уже представляет собой код ASCII-символа, то результат макроса — неизмененное значение аргумента *sim*.

Макро **tolower** (в системе программирования MSC) и функция **tolower** (в системе программирования TC) преобразует код символа, задаваемый значением аргумента *sim*, в код соответствующей прописной буквы, если значение *sim* является кодом строчной буквы. Если значение *sim* не является кодом строчной буквы, то результат **tolower** — неизмененное значение аргумента *sim*.

Макро **_tolower** является версией **tolower**, которая используется, когда известно, что значение аргумента *sim* — код строчной буквы.

Результат **_tolower** не определен, если значение символа *sim* не является кодом строчной буквы.

Макро **toupper** (в системе программирования MSC) и функция **toupper** (в системе программирования TC) преобразует код символа, задаваемый значением аргумента *sim*, в код соответствующей строчной буквы, если значение *sim* является кодом прописной буквы. Если значение *sim* не является кодом прописной буквы, то результат **toupper** — неизмененное значение аргумента *sim*.

Макро **_tolower** является версией **tolower**, которая используется, когда известно, что значение аргумента *sim* — код прописной буквы. Результат **_tolower** не определен, если значение символа *sim* не является кодом прописной буквы.

Для системы программирования TC в файле *ctype.h* определены макросы и прототипы:

```
#define _toupper(c)((c)+'A'—'a')

#define _tolower(c)((c)+'a'—'A')

#define toascii(c)((c)&0x7f)

int tolower(int); /*прототип функции*/

int toupper (int); /*прототип функции*/
```

Для системы программирования MSC в файле *ctype.h* определены макросы:

```
#define toupper(c)(islower(c))?_toupper(c):(c)) /*макро*/

#define tolower(c)(isupper(c))?_tolower(c):(c)) /*макро*/

#define _tolower(c)((c)-'A'+'a')

#define toupper(c)((c)-'a'+'A')

#define toascii(c)((c)&0x7f)
```

Возвращаемое значение

Преобразованный символ.

Нет ошибочных кодов возврата.

Смотри также

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

TZSET (TC & MSC)

#include <time.h> /*используется только для описания функции*/

void tzset(void);

int daylight;

long timezone;

*char *tzname[2];*

Описание

Функция **tzset** использует переменную текущей среды окружения TZ (смотри описание операционной системы MS-DOS), чтобы присвоить значения трем переменным: *daylight*, *timezone* и *tzname*.

Эти переменные используются функциями **ftime** и **localtime**, чтобы провести корректировку времени по Гринвичу (Greenwich Mean Time (GMT)) в локальное время для данного региона.

Значение строковой переменной TZ должно состоять из трехбуквенного имени зоны, такого как PST (Pacific Standart Time), за которым следует число (возможно со знаком), задающее различие в часах между GMT и локальным временем.

За числом может следовать трехбуквенное имя *daylight saving time* зоны, такое как PDT (Pacific Daylight Saving Time). Например, строка "PST8PDT" представляет правильное значение переменной окружения TZ для зоны Pacific time.

Следующие значения присваиваются переменным *daylight*, *timezone* и *tzname* при вызове **tzset**:

Timezone различие в секундах между временем по Гринвичу (GMT) и локальным временем.

Daylight ненулевое значение, если *daylight saving time* определено в переменной среды TZ; иначе значение 0.

Tzname[0] строковое значение трехбуквенного имени зоны из переменной среды TZ.

Tzname[1] строковое значение трехбуквенного имени зоны *daylight saving time* и пустая строка, если поле *daylight saving time* не включено в значение переменной среды TZ.

Если переменная окружения TZ не установлена, то по умолчанию устанавливается значение "PST8PDT", которое соответствует времени зоны Pacific.

По умолчанию *daylight* равняется 1; *timezone* равняется 28800; *tzname[0]* равняется -"PST"; и *tzname[1]* равняется "PDT".

Возвращаемое значение

Функция не возвращает значения.

Смотри также

asctime, ftime, localtime, ctime, gmtime, stime, time

ULTOA (TC & MSC)

#include <stdlib.h> /*используется только для описания функции*/

char ultoa(value, string, radix);

unsigned long value;

*char *string;*

int radix;

Описание

Функция **ultoa** преобразует значение аргумента *value* типа беззнаковое длинное целое в строку, завершающуюся нулевым символом ('\0'), и записывает результат в память по адресу, задаваемому значением аргумента *string*.

Значение аргумента *radix* определяет систему исчисления для преобразования значения *value*; значение аргумента *radix* должно быть в пределах от 2 до 36.

Длина строки-результата может быть до 33 символов.

Возвращаемое значение

Значение аргумента *string*.

Нет ошибочных кодов возврата.

Смотри также

itoa, ltoa

UMASK (MSC)

#include <sys\types.h>

#include <sys\stat.h>

#include <io.h>

int umask(pmode);

int pmode;

Описание

Функция **umask** устанавливает маску доступа к файлам (для текущего процесса) на основе значения аргумента *pmode*.

Маска способа доступа используется для модификации способа доступа для новых файлов, создаваемых посредством функций **creat**, **open** или **sopen**.

Если бит в маске равен 1, то соответствующий бит в файлах, запрашивающих значение доступа, устанавливается в 0 (что соответствует значению "не разрешено"). Если бит в маске равен 0, то соответствующий бит остается неизменным.

Способ доступа для нового файла устанавливается после того, как произойдет первое закрытие файла.

Аргумент *pmode* является константным выражением, содержащим одну или обе явные константы S_IWRITE и S_IREAD, определенные в *sys\stat.h*. Когда заданы обе константы, они разделяются операцией ИЛИ (|).

Значение аргумента *pmode*:

S_IWRITE Запись не разрешена (файл только для чтения)

S_IREAD Чтение не разрешено (файл только для записи)

Например, если в маске установлен бит записи, то новый файл будет доступен только для чтения.

Замечание. Под MS-DOS все файлы доступны для чтения: нет необходимости задавать доступ только для записи.

Возвращаемое значение

Предыдущее значение *pmode*.

Нет ошибочных кодов возврата.

Смотри также

chmod, creat, _creat, mkdir, open, _open

UNGETC (TC & MSC & ANSI)

#include <stdio.h>

int ungetc(sim, stream);

int sim;

*FILE *stream;*

Описание

Функция **ungetc** возвращает символ, код которого задается значением аргумента *sim*, обратно в заданный вводной поток *stream* (ввод/вывод высокого уровня).

Поток *stream* должен быть буферизован и открыт для чтения.

Последующая операция чтения из потока *stream* первым символом считает возвращенный символ.

Функция возвращает ошибочное значение, если еще не было операций чтения из потока *stream* или если символ невозможно вернуть в поток.

Символы, возвращенные в поток посредством **ungetc**, могут быть потеряны, если функция **fseek** или **rewind** будет вызвана раньше, чем возвращенный символ будет вновь считан из потока.

Замечание 1. Попытка вернуть в вводной поток значение EOF игнорируется.

Замечание 2. Предварительно должна быть выполнена хотя бы одна операция чтения из потока.

Замечание 3. Нельзя подряд (без чтения из потока) вернуть в поток более одного символа, в случае повторного вызова **ungetc** для потока ранее возвращенный символ будет утерян.

Возвращаемое значение

Значение аргумента *sim*.

Значение EOF, если попытка вернуть символ оказалась неудачной.

Смотри также

getc, getchar, putc, putchar

UNGETCH (TC & MSC)

#include <conio.h> /*используется только для описания функции*/

int ungetch(sim);

int sim;

Описание

Функция **ungetch** возвращает символ, код которого задается значением аргумента *sim*, обратно на консоль с тем, чтобы он был подан на вход следующей операции чтения.

Попытка вернуть символ будет неудачна, если функция будет вызвана два раза подряд без считывания символа с консоли между этими вызовами (буфер возвращенных символов позволяет хранить только один символ).

Возвращаемое значение

Значение аргумента *sim*, если операция выполнена успешно.

Значение EOF, если попытка вернуть символ неудачна.

Смотри также

cscanf, getch, getche, ungetch

UNIXTODOS (TC)

#include <dos.h>

void unixtodos(time, pdate, ptime);

long time;

*struct date *pdate;*

*struct time *ptime;*

Описание

Функция **unixtodos** преобразует время в формате ОС UNIX, представленное значением параметра *time*, и заполняет информацией структуры, адреса которых определяются значениями параметров *pdate* и *ptime*.

Смотри также описание функции **dostounix**.

Возвращаемое значение

Функция не возвращает значения.

Смотри также

dostounix

UNLINK (TC & MSC)

#include <io.h> /*используется только для описания функции*/

#include <stdio.h> /*используется либо io.h, либо stdio.h*/

int unlink(pathname);

*char *pathname;*

Описание

Функция **unlink** используется для удаления файла, имя которого задается строкой, адрес которой определяется значением аргумента *pathname*.

Возвращаемое значение

Значение 0, если удаление файла произведено успешно.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — имя файла определяет каталог или файл, доступный только для чтения

ENOENT — файл или путь не найдены.

Смотри также

close, remove

UTIME (MSC)

#include <sys\types.h>

#include <sys\utime.h>

int utime(pathname, times);

*char *pathname;*

*struct utimbuf *times;*

Описание

Функция устанавливает для файла, имя которого задается строкой, адрес которой определяет значение параметра *pathname*, время последней модификации.

Процесс должен иметь доступ к файлу по записи, иначе время не может быть изменено.

Хотя структура *utimbuf* содержит несколько полей для записи времени, под MS-DOS можно задать только время последней модификации.

Если значение аргумента *times* равно NULL, время модификации устанавливается равным текущему времени. В противном случае, значениие *times* определяет адрес структуры типа *utimbuf* (тип структуры определен в файле *sys\utime.h*).

Время модификации устанавливается из поля *modtime* этой структуры.

Возвращаемое значение

Значение 0, если время модификации для файла изменено.

Значение -1 сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — имя файла определяет каталог или файл, доступный только для чтения.

EMFILE — слишком много открытых файлов (файл должен открываться, чтобы изменить время его модификации).

ENOENT — файл или каталог на пути к нему не найдены.

Смотри также

asctime, ctime, fstat, ftime, gmtime, localtime, stat, time

VA_ARG, VA_END, VA_START (TC & MSC & ANSI)

Использование (MSC)

#include <varargs.h> /*Требуется для совместимости с UNIX V*/

#include <stdarg.h> /*Требуется для совместимости с ANSI C*/

/*Используется только один из этих файлов*/

type va_arg(arg_ptr, type);

void va_end(arg_ptr);

void va_start(arg_ptr);

void va_start(arg_ptr, prev_param);

va_list arg_ptr;

type /*макроимена*/

prev_param

va_alist

va_dcl

Использование (TC)

#include <stdarg.h> /*Требуется для совместимости с ANSI C*/

void va_start(arg_ptr, prev_param);

type va_arg(arg_ptr, type);

void va_end(arg_ptr);

va_list arg_ptr;

type /*макроимена*/

prev_param

va_alist

Описание

Макросы **VA_START**, **VA_ARG** и **VA_END** предоставляют возможность доступа к аргументам функции, когда функции передается переменное число аргументов.

В MSC можно использовать две версии макросов: макросы, определенные в файле *varargs.h*, совместимы с UNIX SYSTEM V, а макросы, определенные в файле *stdarg.h*, соответствуют ANSI C стандарту. В TC используется только версия *stdarg.h*.

Обе версии предполагают, что функции передается фиксированное число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы описываются как простые формальные параметры функции, и к ним доступ осуществляется по имени.

Доступ к необязательным аргументам осуществляется через макросы, определенные в файле *varargs.h* или *stdarg.h*, которые устанавливают указатель на первый необязательный аргумент в списке аргументов, перебирают аргументы из списка и переустанавливают указатель, когда аргумент обработан.

В операционной системе UNIX SYSTEM V макросы, определенные в *varargs.h*, используются следующим образом:

1) Любые обязательные аргументы для функции могут быть описаны как параметры обычным образом.

2) Последний параметр для функции представляет список необязательных аргументов. Этот параметр должен иметь имя *va_alist* (не перепутайте с *va_list*, который определяется как тип для *va_alist*).

3) Макро **va_dcl** появляется после определения функции и перед открывающей левой скобкой функции. Эта маска определяет, как завершить описание параметра *va_alist*, включением завершающей точки с запятой; следовательно, точка с запятой не должна следовать за *va_dcl*.

4) В пределах функции макро **va_start** устанавливает *arg_ptr* на начало списка необязательных аргументов, передающихся функции. Макро **va_start** должно быть использовано раньше, чем будет в первый раз использовано **va_arg**. Аргумент *arg_ptr* должен иметь тип *va_list*.

5) Макро **va_arg** делает следующее:

— находит значение заданного *type* по адресу, определяемому *arg_ptr*,

— передвигает указатель *arg_ptr* (т.е. присваивает указателю новое значение) на следующий аргумент в списке, используя размер *type*, чтобы определить, где начинается следующий аргумент.

Макро **va_arg** может быть использовано любое число раз в пределах функции последовательной выборки аргументов из списка.

6) После того как все аргументы найдены, **va_end** устанавливает значение указателя равным NULL.

В системе программирования MSC в файле *varargs.h* содержатся следующие описания:

```
typedef char *va_list;

#define va_dcl va_list va_alist;

#define va_start(ap) ap=(va_list)&va_alist

#define va_arg(ap,t)((t*)(ap+=sizeof(t)))[-l]

#define va_end(ap) ap=NULL
```

Вариант макроопределений, предусмотренный ANSI-C, определенный в файле *stdarg.h*, предполагает несколько отличный способ использования макро.

1) Все обязательные аргументы функции описываются как параметры обычным образом. Макро **va_dcl** с *stdarg.h* не используется.

2) Макро **va_start** устанавливает *arg_ptr* на первый необязательный аргумент в списке аргументов, передаваемых функции. Аргумент *arg_ptr* должен иметь тип *va_list*. Аргумент *prev_param* является именем требуемого параметра, предшествующего первому необязательному аргументу из списка аргументов. Макро **va_start** должно быть использовано раньше первого использования **va_arg**.

3) Макро **va_arg** делает следующее:

— находит значение заданного *type* из расположения, заданного в *arg_ptr*;

— увеличивает *arg_ptr*, чтобы тот указывал на следующий аргумент в списке, используя размер *type*, чтобы определить, где начинается следующий аргумент.

Макро **va_arg** может быть использовано любое число раз в пределах функции для последовательного выделения аргументов из списка.

4) После того как все аргументы найдены, *va_end* устанавливает значение указателя равным NULL.

В системе программирования TC в файле *stdarg.h* содержатся следующие описания (не удивляйтесь, ЭТО НЕ БУДЕТ РАБОТАТЬ, воспользуйтесь приведенными ниже описаниями для системы программирования MSC (опишите их в своем файле или исправьте включаемый файл).

```
typedef void *va_list;

#define va_start(ap, parmN) (ap=...)

#define va_arg(ap, type) (*((type *)(ap))++)

#define va_end(ap)

#define _va_ptr(...)
```

В системе программирования MSC в файле *stdarg.h* содержатся следующие описания:

```
typedef char *va_list;

#define va_start(ap,v) ap=(va_list)&v+sizeof(v)

#define va_arg(ap,t) ((t*)(ap+=sizeof(t)))[-1]

#define va_end(ap) ap=NULL
```

Возвращаемое значение

Макро **va_arg** возвращает текущий аргумент; **va_start** и **va_end** не возвращают значения.

Смотри также

vfprintf

VFPRINTF, VPRINTF, VSPRINTF (TC & MSC & ANSI)

Использование (MSC)

```
#include <stdio.h>
```

#include <varargs.h> /*для совместимости с UNIX V*/

#include <stdarg.h> /*для совместимости с ANSI C*/

/*используется только один из этих файлов*/

int vfprintf(stream, format_string, arg_ptr);

int vprintf(format_string, arg_ptr);

int vsprintf(buffer, format_string, arg_ptr);

*FILE *stream;*

*char *buffer;*

*char *format_string;*

va_list arg_ptr;

Использование (TC)

#include <stdio.h>

#include <stdarg.h> /*для совместимости с ANSI C*/

int vfprintf(stream, format_string, arg_ptr);

int vprintf(format_string, arg_ptr);

int vsprintf(buffer, format_string, arg_ptr);

*FILE *stream;*

*char *buffer;*

*char *format_string;*

va_list arg_ptr;

Описание

Функции **vfprintf**, **vprintf** и **vsprintf** форматизируют и выводят данные в поток *stream*, в стандартный вывод или буфер соответственно.

Эти функции подобны функциям **fprintf**, **printf** и **sprintf**, но допускают (принимают как параметр) указатель на список аргументов — выводимых значений.

Строка, адрес которой определяется значением параметра *format_string*, интерпретируется так же, как *format_string* для функции **printf**; смотри описание **printf** для получения описания строки *format_string*.

Параметр *arg_ptr* имеет тип *va_list*, который определен в *varargs.h* и *stdarg.h* (смотри описание макросов **va_arg**, **va_end**, **va_start**).

Значение параметра *arg_ptr* задает адрес размещения последовательности аргументов, которые преобразуются и выводятся в соответствии со спецификациями формата вывода, задаваемыми строкой *format_string*.

Возвращаемое значение

Число записанных символов.

Смотри также

fprintf, printf, sprintf, va_arg, va_end, va_start

VFSCANF, VSCANF, VSSCANF (TC)

#include <stdio.h>

#include <stdarg.h> /*для совместимости с ANSI C*/

int vfscanf(stream, format_string, arg_ptr);

int vscanf(format_string, arg_ptr);

int vsscanf(buffer, format_string, arg_ptr);

*FILE *stream;*

*char *buffer;*

*char *format_string;*

va_list arg_ptr;

Описание

Функции **vfscanf**, **vscanf** и **vsscanf** вводят и преобразуют к необходимому формату данные из потока stream, из стандартного ввода или из буфера соответственно.

Эти функции подобны функциям **fscanf**, **scanf** и **sscanf**, но принимают как параметр указатель на список аргументов-адресов переменных, которым присваиваются вводимые значения.

Строка, адрес которой определяется значением параметра *format_string*, интерпретируется так же, как *format_string* для функции **scanf**; смотри описание функции **scanf** для получения описания строки *format_string*.

Параметр *arg_ptr* имеет тип *va_list*, который определен в *varargs.h* и *stdarg.h* (смотри описание макросов **va_arg**, **va_end**, **va_start**).

Значение параметра *arg_ptr* задает адрес размещения последовательности аргументов, значения которых определяют адреса переменных, в которые записывают вводимые и преобразуемые в соответствии со спецификациями формата ввода, задаваемыми строкой *format_string*, данные.

Возвращаемое значение

Число успешно обработанных полей формата ввода.

Смотри также

vfprintf, vprintf, vsprintf, va_arg, va_end, va_start, scanf, sscanf, fscanf

WRITE (TC & MSC), _WRITE (TC)

#include <io.h> /*используется только для описания функции*/

int write(handle, buffer, count);

int handle;

*char *buffer;*

unsigned int count;

int _write(handle, buffer, count); /*функция _write доступна*/

int handle; /*только в системе программирования TC*/

*char *buffer;*

unsigned int count;

Описание

Функция **write** записывает *count* байтов из области памяти, адрес которой определяется значением параметра *buffer*, в файл, связанный с дескриптором *handle* (ввод/вывод низкого уровня). Операция записи начинается с текущей позиции указателя файла.

Если файл открыт для расширения (O_APPEND), операция записи начинается с позиции конца файла. После операции записи указатель файла увеличивается на число реально записанных в файл байтов (это число может быть больше значения *count*, если файл открыт в текстовом режиме, смотри описание функции **open** для описания преобразований, выполняемых в текстовом режиме).

Функция **_write** записывает в файл информацию, всегда полагая, что файл открыт в двоичном режиме. Кроме того, если файл открыт в режиме добавления (O_APPEND), перед записью не производится позиционирование на конец файла.

Возвращаемое значение

Число байтов, действительно записанных в файл. Возвращаемое значение может не равняться значению аргумента *count*.

Значение -1 — сигнализирует об ошибке; при этом переменной *errno* присваивается одно из следующих значений:

EACCES — файл доступен только для чтения или защищен от записи;

EBADF — недействительный дескриптор файла *handle*;

ENOSPC — нет памяти на устройстве.

Если записывается более 32 Кбайтов (максимальное число, представимое переменной типа *int*) в файл, возвращаемое значение должно быть типа *unsigned int*. Однако максимальное число байтов для записи в файл равняется 65534, так как число 65535 (или 0xFFFF) интерпретируется как -1 и служит признаком ошибки.

Смотри также

fwrite, open, read

10 ПРИЛОЖЕНИЕ А. Перечень особенностей реализации языка Си в различных версиях СП MSC и СП TC

1. В СП TC добавлен специальный символ **?** для представления символа знак вопроса (раздел 1.1.4).

2. В СП TC шестнадцатеричное байтовое значение константы может начинаться не только префиксом **x**, как в СП MSC префиксом **X.** Вслед за префиксом **x** в версии 4 СП MSC может следовать одна или две шестнадцатеричные цифры, а в версии 5 СП MSC — до трех шестнадцатеричных цифр (раздел 1.1.4).

3. В СП TC при задании константы можно указать суффикс **U** (или **u**), означающий, что константа имеет тип **unsigned int** (раздел 1.2.1).

4. В СП TC константе присваивается тип **unsigned long**, если ее значение превышает 65535, независимо от наличия или отсутствия суффикса **U**, а в СП MSC в этом случае константе присваивается тип **long** (раздел 1.2.1).

5. В СП TC можно явно присвоить константе тип **float**, добавив к ней суффикс **f** или **F** (раздел 1.2.2).

6. Помимо односимвольных констант, в СП TC реализованы двухсимвольные константы (раздел 1.2.3).

7. В СП MSC версии 5 и в СП TC 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 (раздел 1.2.4).

8. В СП TC реализована опция компиляции, позволяющая хранить в памяти только одну из нескольких идентичных строк (раздел 1.2.4).

9. В идентификаторах версии 1.5 СП TC допускается символ **$**, однако идентификатор не может с него начинаться.

10. В СП TC реализованы следующие ключевые слова, которые отсутствуют в СП MSC:

asm	_cs	_CH
interrupt	_ds	_CL
	_es	_CX
	_ss	_DH
	_AH	_DL
	_AL	_DX
	_AX	_BP
	_BH	_DI
	_BL	_SI
	_BX	_SP

Ключевое слово **fortran** реализовано только в СП MSC. В версии 4 СП MSC ключевые слова **const** и **volatile** зарезервированы, но использовать их невозможно. В версии 5 СП MSC ключевое слово **volatile** реализовано лишь синтаксически, а **const** — полностью (как синтаксически, так и семантически). В СП TC ключевые слова **const** и **volatile** реализованы полностью. Ключевое слово **interrupt** в СП MSC версии 4 не реализовано (раздел 1.4).

11. В СП TC существует опция компиляции, допускающая вложенные комментарии (раздел 1.5).

12. Тип **long float** реализован только в версии 4 СП MSC и эквивалентен типу **double**. В версии 5 СП MSC и в СП TC реализован тип **long double**, причем в версии 5 СП MSC и версии 1.5 СП TC он эквивалентен типу **double**, а в версии 2.0 СП TC является самостоятельным плавающим типом (раздел 3.1).

13. В СП TC существует опция компиляции, задающая выравнивание всех объектов, занимающих более одного байта, на границу четного адреса (раздел 3.2).

14. В СП TC использование модификаторов **near**, **far**, **huge** ограничено: они могут быть записаны только перед идентификатором функции или перед признаком указателя — звездочкой (раздел 3.3.3.1).

15. В СП MSC, в отличие от СП TC, недопустима инициализация **const** объектов, имеющих класс памяти **auto** (раздел 3.3.3.2).

16. Если с помощью операции приведения типа преобразовать указатель на **const** к указателю на тип, отличный от **const**, то СП TC позволит выполнить присваивание объекту через преобразованный указатель (раздел 3.3.3.2).

17. В СП TC указатель типа **huge** хранится в нормализованном формате (раздел 3.3.3.4).

18. В СП MSC модификатор **huge** применяется только к массивам, размер которых превышает 64К. В СП TC недопустимы массивы больше 64К, а модификатор **huge** применяется к функциям и указателям для спецификации того, что адрес функции или указуемого объекта имеет тип **huge** (раздел 3.3.3.4).

19. В СП TC, если в одном объявлении класса памяти **auto** или **register** либо внутри структуры содержатся описатели нескольких объектов, порядок их размещения в памяти будет обратным (раздел 3.4.3).

20. Для битового поля в версии 4 СП MSC спецификация типа должна задавать беззнаковый целый тип (**unsigned int**). В версии 5 СП MSC спецификация типа может задавать как знаковый, так и беззнаковый целый тип, причем любого размера — **char**, **int**, **long**. При этом для неименованного битового поля выравнивание будет производиться на границу того типа, который задан спецификацией. Однако знаковый целый тип для битовых полей реализован лишь синтаксически, а в выражениях битовые поля участвуют как беззнаковые значения.

В СП TC битовое поле может иметь либо тип **unsigned int**, либо тип **signed int**, причем знак, в отличие от версии 5 СП MSC, учитывается при вычислениях (раздел 4.4.3).

21. В СП MSC каждый элемент структуры, тип которого отличен от **char** или **unsigned char**, выравнивается в памяти на границу 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;

— для того чтобы структура содержала четное число байтов, в конец структуры может быть добавлен лишний байт.

В версии 4 СП MSC элемент структуры, представляющий собой структуру нечетной длины, дополняется лишним байтом в конце, чтобы его длина стала четной. В версии 5 СП MSC это дополнение лишним байтом производится только в том случае, когда тип следующего элемента структуры требует его размещения с четного адреса (раздел 3.4.3).

22. СП MSC, в отличие от СП TC, не допускает в объединении битовые поля (раздел 3.4.4).

23. В СП TC нельзя получить значение по операции косвенной адресации, примененной к указателю на тип **void**. В СП MSC в этом случае выдается предупреждающее сообщение (раздел 3.4.6).

24. В версии 5 СП MSC, а также в СП TC реализован метод объявления прототипов функций (раздел 3.5).

25. Компиляторы языка Си по-разному реагируют на наличие в исходном файле двух объявлений одной переменной с различными спецификациями класса памяти (раздел 3.6.1).

26. Инициализация переменных составных типов (массив, структура, объединение), имеющих класс памяти **auto**, запрещена в СП MSC, но допускается в СП TC. В СП TC также допустима инициализация переменных класса памяти **auto** с модификатором **const** (раздел 3.7).

27. В СП TC не обязательно заключать инициализатор объединения в фигурные скобки (раздел 3.7.2).

28. Область действия тега, объявленного в составе абстрактного имени типа, распространяется в СП MSC до конца охватывающего блока, а в СП TC — до конца тела функции (раздел 3.8.3).

29. В директивах препроцессора СП TC позволяет использовать операцию **sizeof** (раздел 4.2.9).

30. Операция унарного плюса реализована полностью только в СП TC. В СП MSC версии 5 она реализована только синтаксически (раздел 4.3.2).

31. В СП MSC версии 4 операндом операции **sizeof** может быть имя абстрактного типа данных либо L-выражение. В СП MSC и в СП TC, помимо перечисленного, допустимо произвольное выражение. Применение операции **sizeof** к идентификатору функции в СП TC считается ошибкой, а в СП MSC эквивалентно определению размера указателя на функцию (раздел 4.3.2).

32. В операции сдвига преобразования по умолчанию выполняются в СП MSC над обоими операндами совместно, а в СП TC независимо над каждым операндом (раздел 4.3.5).

33. В операции присваивания в СП MSC версии 5 преобразование значения от типа **unsigned long** к типу **double** производится напрямую, без промежуточного преобразования к типу **long** (раздел 4.7.1).

34. В СП MSC для того, чтобы присвоить указатель на данные указателю на функцию (или обратно), необходимо выполнить явное приведение его типа (раздел 4.7.1).

35. Выражение переключения в операторе переключателя **switch** 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 значение все равно преобразуется к типу **int** (раздел 5.10).

36. В версии 5 СП MSC и в СП TC поддерживается форма записи заголовка определения функции, основанная на методе прототипов (раздел 6.2). -

37. В СП MSC версии 5 и в СП TC реализован модификатор типа функции **interrupt** (раздел 6.2.2).

38. Если функции передается больше фактических аргументов, чем объявлено имен в списке типов аргументов в ее предварительном объявлении, и этот список не завершен многоточием, то компилятор выдаст предупреждающее сообщение в СП MSC и сообщение об ошибке в СП TC (раздел 6.4.1).

39. Если список типов аргументов в объявлении функции 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 выдает предупреждающее сообщение в СП MSC и сообщение об ошибке в СП TC (раздел 6.4.1).

40. В СП MSC версии 5 и в СП TC в строке, содержащей препроцессорную директиву, перед символом # допустимы пробельные символы (раздел 7.1).

41. В СП MSC версии 5 и в СП TC реализованы две специальные препроцессорные операции: склейка лексем (##) и преобразование макроаргумента (#) (раздел 7.2.2).

42. В СП TC имеется возможность задавать имя пути в директиве **#include** с помощью макроопределения (раздел 7.3).

43. СП TC, в отличие от СП MSC, позволяет использовать операцию **sizeof** в ограниченном константном выражении в препроцессорных директивах **#if** и **#elif** (раздел 7.4.1).

44. В СП TC реализована директива обработки ошибок **#error** и пустая директива # (разделы 7.6, 7.7).

45. В СП MSC не реализованы псевдопеременные _DATE_ и _TIME_ (раздел 7.9).

46. Организация моделей памяти в СП TC имеет ряд отличий от СП MSC (раздел 8.3).

11 ПРИЛОЖЕНИЕ Б. СООБЩЕНИЯ ОБ ОШИБКАХ

Это приложение описывает значения, которые могут быть присвоены переменной **errno**, когда происходит ошибка при вызове библиотечной функции. Заметим, что не все библиотечные функции в случае ошибки присваивают значение переменной **errno**.

Сообщение об ошибке связано с каждым значением **errno**. Это сообщение может быть напечатано с помощью функции **perror**.

Значение переменной **errno** отражает характер ошибки, произошедшей во время вызова библиотечной функции, при котором было присвоено значение переменной **errno**. Значение переменной **errno** автоматически не очищается при последующем успешном вызове.

Таким образом, для получения точных результатов необходимо выполнить проверку на ошибку и напечатать сообщение об ошибке, если требуется, сразу после вызова.

В файле **errno.h** содержится определение значений **errno**. Однако не все определения, заданные в файле **errno.h** используются под ОС MS-DOS.

В настоящем приложении описываются только те значения переменной **errno**, которые используются в ОС MS-DOS. Для получения полного списка значений **errno** см. файл **errno.h**.

```
#define EZERO       0      /*Error 0*/
#define EINVFNC     1      /*Invalid function number*/
#define ENOFILE     2      /*File not found*/
#define ENOPATH     3      /*Path not found*/
#define ECONTR      7      /*Memory blocks destroyed*/
#define EINVMEM     9      /*Invalid memory block address*/
#define EINVENV     10     /*Invalid environment*/
#define EINVFMT     11     /*Invalid format*/
#define EINVACC     12     /*Invalid access code*/
#define EINVDAT     13     /*Invalid data*/
#define EINVDRV     15     /*Invalid drive specified*/
#define ECURDIR     16     /*Attempt to remove CurDir*/
#define ENOTSAM     17     /*Not same device*/
#define ENMFILE     18     /*No more files*/

#define ENOENT      2      /*No such file or directory
#define EMFILE      4      /*Too many open files*/
#define EACCES      5      /*Permission denied*/
#define EBADF       6      /*Bad file number*/
#define ENOMEM      8      /*Not enough core*/
#define ENODEV      15     /*No such device*/
#define EINVAL      19     /*Invalid argument*/
#define E2BIG       20     /*Arg list too long*/
#define ENOEXEC     21     /*Exec format error*/
#define EXDEV       22     /*Cross-device link*/
#define EDOM        33     /*Math argument*/
#define ERANGE      34     /*Result too large*/
#define EEXIST      35     /*File already exists*/
```

Таблица

Значения переменной errno и их смысл

Значение	Сообщение	Описание
E2BIG	Arg list too long (список аргументов слишком длинный)	Список аргументов превышает 128 байтов, или память, требуемая для информации окружения, превышает 32 Кбайта.
EACCES	Permission denied (доступ невозможен)	Доступ невозможен: разрешение файлов не позволяет указанный доступ. Эта ошибка может происходить в ряде обстоятельств; при попытке получить доступ к файлу (или, в таком же случае, оглавлению) таким образом, который несовместим с атрибутами файла. Например: попытка чтения из файла который не открыт, попытка записи в файл, открытый только для чтения, или открыть оглавление вместо файла. В версиях MS-DOS 3.0 и выше, EACCES также может указывать на нарушение запирания или разделения. Ошибка может также встретиться при попытке переименовать файл или удалить существующее оглавление.
EBADF	Bad file number (плохой номер дескриптора)	Handle указанного файла не является действительным значением handle для файла или относится к неоткрытому илу; или была попытка записать в йл или устройство, открытые только для чтения (или наоборот).
EDEADLOCK	Resource deadlock would occur (произошел тупик разделения ресурсов)	Запирающее нарушение: файл не может быть заперт после 10 попыток (MS-DOS версия 3.0 и позже).
EDOM	Math argument, (аргумент математический)	Аргумент к математической функции не входит в область функции.
EEXIST	File exists (файл существует)	Флаги O_CREAT и O_EXCL указаны, когда открывается файл, а файл с таким именем уже существует.
EINVAL	Invalid argument	Недействительная величина, которая дана для одного из аргументов к функции. Например, величина дана для начала, но указатель файла позиционирован раньше начинающегося в файле.
EMFILE	Too many open files (слишком много открытых файлов)	Нет более доступных дескрипторов, так как слишком много открытых файлов.
ENOENT	No such file or directory (нет такого файла или каталога)	Указанные файл или оглавление не существуют или не могут быть найдены. Это сообщение возможно, когда указанный файл не существует, или компонента существующего оглавления не указана в пути.
ENOEXEC	Exec format error (ошибка формата exec)	Попытка выполнения файла, который не является выполняемым или имеет недействительный формат для выполнения.
ENOMEM	Not enough core, (нет достаточного ядра)	Нет достаточной памяти, годной к использованию. Это сообщение может происходить, когда недостаточно имеющейся памяти для выполнения дитя процесса, или когда запрос на получение памяти в sbrk или getcwd не может быть удовлетворен.
ENOSPC QQQQ	No space left on device (нет пространства, отпущенного в устройстве)	Больше нет пространства, доступного для записи на устройстве (например, диск полон).

ERANGE	Result too large (результат слишком большой)	Аргумент в математической функции слишком большой, результат частично или вообще потерян. Эта ошибка может появиться в функции, когда аргумент больше, чем ожидается. (Например, когда аргумент имя-пути в функции getcwd длиннее, чем ожидается).
EXDEV	Cross-device link	При попытке сделать передачу файла с одного устройства на другое (используя функцию rename).
		Математические ошибки
DOMAIN		Аргумент функции вне сферы этой функции.
OVERFLOW		Результат слишком большой, чтобы быть представленным в возвращаемом значении функции.
PLOSS		Произошла частичная потеря значимости.
SING		Специфичный аргумент: аргумент функции имеет незаконную величину (например, передача величины 0 в функцию, которая запрашивает ненулевую величину).
TLOSS		Полная потеря значимости.
UNDERFLOW		Результат слишком мал.

12

13 СПИСОК ЛИТЕРАТУРЫ

1. **Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку Си**: Пер. с англ. - М.: Финансы и статистика, 1985.

2. **Иванов А. Г. Язык программирования Си: Предварительное описание** // Прикладная информатика. - 1985. - вып. 1. – С. 68-113.

3. **Инструментальная мобильная операционная система ИНМОС** / М. И. Беляков, А. Ю. Ливеровский, В. П. Семик, В. И. Шяудкулис. — М.: Финансы и статистика, 1985.

4. **Либеров А.Б., Субботин Д.М. Язык программирования Си** // Методические материалы и документация по пакетам парикладных программ. – Вып. 46. – М.: МЦНТИ, 1986.

5. **Баурн С. Операционная система UNIX**. — М.: Мир, 1986.

6. **Хэнкок Л., Кригер М. Введение в программирование на языке Си**: Пер. с англ. - М.: Радио и связь, 1986.

7. **Банахан М., Раттер Э. Введение в операционную систему UNIX**: Пер. с англ. — М.: Радио и связь, 1986.

8. **Берри Р., Микинз Б. Язык Си: Введение для программистов** / Пер. с англ и предисл. Д. Б. Подшивалова - М.: Финансы и статистика, 1988.

9. **Болски М.И. Язык программирования Си. Справочник**: Пер. с англ. — М.: Радио и связь, 1988.

10. **Уэйт М., Прата С., Мартин Д. Язык Си. Руководство для начинающих**: Пер. с англ. - М.. Мир, 1988.

11. **Джехани Н. Программирование на языке Си**: Пер. с англ. — М.: Радио и связь, 1988.

12. **Языки программирования Ада, Си, Паскаль. Сравнение и оценка** / Под ред. А.Р. Фьюэра, Н. Джехани: Пер. с англ. — М.: Радио и связь, 1989.

13. **Хендрикс Д. Компилятор языка Си для микроЭВМ**: Пер. с англ. — М.: Радио и связь, 1989.

14. **Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си**: Пер. с англ. — М.: Радио и связь, 1989.

15. **Microsoft C for the MS-DOS operating system**. Microsoft Corporation, 1986.

16. **Microsoft C for the MS-DOS operating system**. Microsoft Corporation, 1987.

17. **TURBO C. User's Guide**. Borland International, Inc. 1987.

18. **TURBO C. Reference Guide**. Borland International, Inc. 1987.

19. **TURBO C. User's Guide**. Borland International, Inc. 1988.

С. О. Бочков
Д. М. Субботин

ЯЗЫК ПРОГРАММИРОВАНИЯ Си ДЛЯ ПЕРСОНАЛЬНОГО КОМПЬЮТЕРА

Центр обучения советско-американского СП "ДИАЛОГ" совместно с издательством "Радио и связь" готовит к выпуску серию монографий по вопросам программирования, применения и технического обслуживания ПК.

План выпуска включает в себя три раздела:

1. Программное обеспечение.
2. Аппаратные средства ПК, техническое обслуживание, ремонт ПК и периферийных устройств.
3. Применение (использование) ПК.

В разделе книг по программному обеспечению ПК: Операционные системы для ПК (OS MS-DOS и OS/2); Многопользовательские операционные системы (UNIX, XENIX, QNX); Компиляторы языков программирования; Системное программирование в среде MS-DOS и др.

В разделе книг по аппаратным средствам ПК: Профессиональные ПК общего назначения моделей IBM PC (XT/AT/386/P5/2); Аппаратные и программные средства сетей на ПК; Методы контроля работы и диагностики неисправностей ПК и др.

В разделе книг по применению ПК: Использование в делопроизводстве, автоматизации административно-управленческой деятельности, в САПР конструктора (на основе пакета AutoCAD), инженера-системотехника (пакет PCAD); Прикладные программные системы (WORD, WORKS, CHART, WINDOWS); Автоматизация редакционно-издательской деятельности; Базы знаний и экспертные системы; Обработка изображений и деловая графика.

Книга "Язык программирования Си для персонального компьютера" является первой книгой серии. В ближайшее время выйдет книга "Язык программирования Паскаль для персонального компьютера".

ISBN 5-256-00974-5