

621.398(083)

Б-795

• РАДИО И СВЯЗЬ •

СПРАВОЧНИК

М.И. БОЛСКИ

ЯЗЫК
ПРОГРАММИРОВАНИЯ
Си



ЯЗЫК
ПРОГРАММИРОВАНИЯ
Си

THE C PROGRAMMER'S HANDBOOK

M.I.BOLSKY

SYSTEMS TRAINING CENTER

PRENTICE-HALL, INC.

СПРАВОЧНИК

М.И.БОЛСКИ

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

ПЕРЕВОД С АНГЛИЙСКОГО
С.В.ДЕНИСЕНКО



МОСКВА „РАДИО И СВЯЗЬ”

1988

ББК 32.973

Б 79

УДК 681.3.06

Болски М. И.

Б 79 Язык программирования Си. Справочник: Пер. с англ. — М.: Радио и связь, 1988. — 96 с.: ил.

ISBN 5-256-00171-X

Рассмотрены конструкции языка программирования Си, приведены соответствующие примеры и необходимые пояснения. Описаны варианты реализации языка для различных типов ЭВМ, даны рекомендации по его использованию, обеспечивающие мобильность программ. Кратко описаны библиотечные функции языка, имеющиеся в операционной системе UNIX System V.

Для широкого круга программистов.

Б 2405000000-029 156-88
046 (01)-88

ББК 32.973

Редакция переводной литературы

Справочное издание

М. И. БОЛСКИ

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

Заведующая редакцией *О. В. Толкачева*, редактор *М. Г. Коробочкина*, художественный редактор *Т. В. Бусарова*, обложка художника *Н. А. Пашуро*, технический редактор *И. Л. Ткаченко*, корректор *Л. С. Глаголева*

ИБ № 2084

Подписано в печать 22.10.87 Формат 70x100/32 Бумага офс. № 2
Гарнитура "Пресс-роман" Печать офсетная Усл. печ. л. 3,90
Усл.кр.-отт. 4,23 Уч.-изд. л. 4,22 Доп. тираж 140 000 экз.
Изд. № 22268 Заказ № 360 Цена 30 к.
Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Московская типография № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 129041 Москва, Б. Переяславская ул., д. 46

ISBN 5-256-00171-X (рус.)

ISBN 0-13-110073-4 (англ.)

© 1985 by Bell Telephone Laboratories, Incorporated

© Перевод на русский язык, предисловие к русскому изданию
дополнительный список литературы и примечания переводчика.

Издательство "Радио и связь", 1988

Содержание

Предисловие к русскому изданию.	7
1. Введение.	9
2. Общий синтаксис.	10
2.1. Формат (10). 2.2. Комментарии (10). 2.3. Идентификаторы (10). 2.4. Зарезервированные слова (11).	
3. Основные типы данных.	11
3.1. Целые константы (12). 3.2. Длинные целые константы (13). 3.3. Константы с плавающей точкой (13). 3.4. Символьные константы (14). 3.5. Строковые константы (14). 3.6. Перечислимые константы (15). 3.7. Размер данных (15).	
4. Операции и выражения.	17
4.1. Выражения (17). 4.2. Метаобозначения операндов (18). 4.3. Арифметические операции (18). 4.4. Операция присваивания (21). 4.5. Операции отношения (23). 4.6. Логические операции (24). 4.7. Побитовые операции (25). 4.8. Адресные операции (26). 4.9. Операции над массивами (27). 4.10. Операции над структурами или объединениями (27). 4.11. Другие операции (28). 4.12. Приоритеты и порядок выполнения операций (29). 4.13. Порядок обработки операндов (31). 4.14. Арифметические преобразования в выражениях (31).	
5. Операторы.	32
5.1. Формат и вложенность (32). 5.2. Метка оператора (32). 5.3. Составной оператор (32). 5.4. Оператор-выражение (33). 5.5. Оператор завершения break (33). 5.6. Оператор продолжения continue (34). 5.7. Оператор возврата return (34). 5.8. Оператор перехода goto (34). 5.9. Условный оператор if-else (34). 5.10. Оператор-переключатель switch (36). 5.11. Оператор цикла while (37). 5.12. Оператор цикла do-while (38). 5.13. Оператор цикла for (38).	
6. Функции.	39
6.1. Определение функции (39). 6.2. Вызов функции (40). 6.3. Функции main (42).	
7. Описания.	42
7.1. Основные типы (43). 7.2. Указатели и массивы (43). 7.3. Структуры (44). 7.4. Поля бит в структурах (45). 7.5. Объединения (45). 7.6. Перечисления (46). 7.7. Переименование типов (47). 7.8. Определение локальных переменных (47). 7.9. Определение глобальных переменных (49). 7.10. Инициализация переменных (49). 7.11. Описание внешних объектов (51).	
8. Препроцессор.	52
8.1. Замена идентификаторов (52). 8.2. Макросы (53). 8.3. Включение файлов (53). 8.4. Условная компиляция (54).	

8.5. Номер строки и имя файла (55).	
9. Структура программы	55
10. Библиотека ввода-вывода	60
10.1. Доступ к файлам (61). 10.2. Доступ к каналам (62).	
10.3. Состояние файла (62). 10.4. Форматированный ввод-вывод (63). 10.5. Ввод-вывод строк (63). 10.6. Ввод символа (63). 10.7. Вывод символа (64). 10.8. Блочный ввод-вывод (64).	
11. Другие библиотеки.	65
11.1. Выполнение команд языка shell (65). 11.2. Временные файлы (65). 11.3. Обработка строк (66). 11.4. Проверка символов (67). 11.5. Преобразование символов (68). 11.6. Преобразование строки в число (69). 11.7. Доступ к аргументам (69). 11.8. Распределение памяти (70).	
12. Форматированный вывод	70
12.1. Спецификация преобразования (71). 12.2. Спецификация вывода символа (72). 12.3. Спецификация вывода строки (72). 12.4. Спецификация вывода целого числа со знаком (72). 12.5. Спецификация вывода целого числа без знака (73). 12.6. Спецификация вывода числа с плавающей точкой (73).	
13. Форматированный ввод	74
13.1. Спецификация преобразования (76). 13.2. Пустые символы (76). 13.3. Литеральные символы (76). 13.4. Спецификация ввода символа (76). 13.5. Спецификация ввода строки (77). 13.6. Спецификация ввода целого числа (77). 13.7. Спецификация ввода числа с плавающей точкой (77). 13.8. Спецификация ввода по образцу (77).	
14. Мобильность программ на языке Си	78
14.1. Верификатор lint (78). 14.2. Зависимость от компилятора (79). 14.3. Зависимость от ЭВМ (79). 14.4. Хорошо организованные программы (85). 14.5. Мобильность файлов данных (88).	
Приложение. Набор символов кода ASCII	89
Список литературы	92
Дополнительный список литературы	92
Предметный указатель	93

Язык программирования Си был разработан в начале семидесятых годов как инструментальное средство для реализации операционной системы UNIX на ЭВМ PDP-11, однако его популярность быстро переросла рамки конкретной ЭВМ, конкретной операционной системы и конкретных задач системного программирования. В настоящее время любая инструментальная операционная система не может считаться полной, если в ее состав не входит компилятор языка Си.

В некотором смысле язык Си — самый универсальный, так как кроме набора средств, присущих современным языкам программирования высокого уровня (структурность, модульность, определяемые типы данных), в него включены средства для программирования почти на уровне ассемблера (использование указателей, побитовые операции, операции сдвига). Большой набор операторов и операций позволяет писать компактные и эффективные программы. Однако такие мощные средства требуют от программиста осторожности, аккуратности и хорошего знания языка со всеми его преимуществами и недостатками.

Предлагаемый русскому читателю справочник предназначен в первую очередь для программистов-практиков, но он будет полезен всем, кто хочет расширить свой кругозор в области программирования. Этот справочник ориентирован не столько на последовательное чтение, сколько на повседневную работу за терминалом ЭВМ. Все конструкции языка Си, независимо от частоты использования, синтаксической и семантической сложности, описаны одинаково кратко, но исчерпывающе.

Справочник предполагает некоторое знакомство читателя с программированием вообще и с основными понятиями языка Си в частности. Поэтому начинающим программистам мы рекомендуем вначале ознакомиться с учебником по языку Си [Д9] и классическим описанием, ставшим фактическим стандартом языка Си, [Д6]. Краткое описание языка можно также найти в книгах [Д2, Д4, Д5].

Данный справочник несомненно будет большим подспорьем программисту в его нелегком труде. Все конструкции языка описаны неформально, но довольно строго, и проиллюстрированы короткими

ми, тщательно подобранными примерами. Приведены особенности реализации языка Си на разных ЭВМ. При переводе были добавлены данные о советских ЭВМ, для которых имеются компиляторы языка Си.

Очень кратко описаны стандартные функции, входящие в библиотеки языка Си, что, по-видимому, связано со стремлением ограничить объем книги. Идеологически (и генетически) библиотечные функции языка Си связаны с операционной системой UNIX, поэтому в дополнительный список литературы включены книги, описывающие эту операционную систему.

Более подробное описание библиотечных функций можно найти в книгах [Д2, Д5] и, конечно, в документации для пользователей конкретной операционной системы. Надо отметить, что реализации некоторых функций для разных ЭВМ и разных операционных систем могут отличаться. В частности, требуется осторожность при обработке русских текстов. Достаточно сказать, что во всех стандартных в СССР символьных кодах (ДКОИ, КОИ-8, КОИ-7) русские буквы расположены не в алфавитном порядке.

Дополнительную ценность книге придают описанные в последней главе методы разработки мобильных (т. е. переносимых на другие ЭВМ и другие операционные системы) программ на языке Си. Хотя изложенные рекомендации не гарантируют полной мобильности программ, тем не менее нарушение этих правил в процессе разработки лишает программу права носить высокое звание промышленного программного продукта.

С. В. Денисенко

1. ВВЕДЕНИЕ

Компилятор языка программирования Си работает под управлением операционной системы (ОС) UNIX, а также других операционных систем. Большая часть информации, изложенной далее, применима для языка Си в любой операционной системе.

При использовании языка Си в ОС UNIX обычно используются следующие команды операционной системы:

<i>ar</i>	— архивация библиотечных функций;
<i>cc</i>	— компиляция и загрузка;
<i>lint</i>	— проверка синтаксиса и типов (см. также с. 78);
<i>make</i>	— поддержание связанной группы программ;
<i>sdb</i>	— символьная отладка.

Курсы по языку программирования Си и ОС UNIX организуются по всему миру, в том числе и по запросам пользователей. За более подробной информацией обращайтесь по следующим адресам:

В США
AT&T Customer Education
P.O. Box 2000
Hopewell, NJ 08525

За пределами США
AT&T International
P.O. Box 7000B
Basking Ridge, NJ 07920 USA

Для получения документации по ОС UNIX, программного обеспечения и автоматизированных пособий обращайтесь по следующим адресам:

В США
AT&T Technologies
Software Sales and Marketing
P.O. Box 25000
Greensboro, NC 27420

За пределами США
См. выше

Автор выражает глубокую признательность О. Берну и П. Г. Маттеусу, сделавшим много ценных замечаний, а также Д. М. Андерсону, с большим мастерством осуществившему набор этого справочника.

2. ОБЩИЙ СИНТАКСИС

2.1. Формат

Пробелы, символы табуляции, перевода на новую строку и перевода страницы используются как разделители. Вместо одного из таких символов может использоваться любое их количество.

Для повышения читабельности текста рекомендуется использовать символы табуляции.

2.2. Комментарии

Комментарии начинаются парой символов `/*`, заканчиваются парой символов `*/`.

Разрешены везде, где допустимы пробелы.

Примеры

```
/* Однострочный комментарий */
```

```
/*
```

```
* Многострочный комментарий
```

```
*/
```

2.3. Идентификаторы

Идентификаторы используются как имена переменных, функций и типов данных.

Допустимые символы: цифры 0 – 9, латинские прописные и строчные буквы a – z, A – Z, символ подчеркивания (`_`).

Первый символ не может быть цифрой.

Идентификатор может быть произвольной длины, но в некоторых ЭВМ не все символы учитываются компилятором и загрузчиком (см. таблицу ниже).

Примеры

```
NAME1 name1 Total_5 Paper
```

Внешние идентификаторы: число значимых символов и вид букв (прописные/строчные) могут различаться даже на однотипных ЭВМ в зависимости от используемых компиляторов и загрузчиков.

Замечание. Ожидается, что в дальнейшем в качестве стандартной будет принята длина идентификаторов больше восьми символов.

Тип ЭВМ	Длина внешних идентификаторов (число символов), вид букв
3B Computer	8, прописные и строчные
DEC PDP-11	7, прописные и строчные
DEC VAX-11	— ” —
HONEYWELL 6000	6, прописные
IBM 360/370	7, прописные
INTERDATA 8/32	8, прописные и строчные
MOTOROLA 68000	> 8, прописные и строчные
NSC 16000	— ” —
ZILOG 8000	— ” —
INTEL 80286	— ” —

2.4. Зарезервированные слова

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

Замечание. `sizeof` — это операция выполняющаяся во время компиляции. Описание `typedef` используется для определения сокращенной формы описания существующего типа данных. В некоторых реализациях, кроме того, зарезервированы слова `asm` и `fortran`.

3. ОСНОВНЫЕ ТИПЫ ДАННЫХ

К основным типам данных относятся целые числа (`int`, `short`, `long`, `unsigned`), символы (`char`) и числа с плавающей точкой (`float`, `double`).

На их основе строятся производные типа данных (см. с. 42). В этом разделе описаны синтаксис констант и объем памяти, занимаемой основными типами данных.

3.1. Целые константы

Десятичные:

цифры 0 – 9;
первой цифрой не должен быть 0.

Примеры

12 111 956 1007

З а м е ч а н и е. Если значение превышает наибольшее машинное целое со знаком, то оно представляется как длинное целое.

Восьмеричные:

цифры 0 – 7;
начинаются с 0.

Примеры

012 = 10 (десятичное);

0111 = 73 (десятичное);

076 = 62 (десятичное);

0123 = $1*64 + 2*8 + 3 = 83$ (десятичное).

З а м е ч а н и е. Если значение превышает наибольшее машинное целое без знака, то оно представляется как длинное целое.

Шестнадцатеричные: цифры 0 – 9, буквы a – f или A – F для значений 10 – 15;

начинаются с 0x или 0X.

Примеры

0x12 = 18 (десятичное);

0X12 = 18 (десятичное);

0x2f = 47 (десятичное);

0XA3 = 163 (десятичное);

0x1B9 = $1*256 + 11*16 + 9 = 441$ (десятичное).

З а м е ч а н и е. Если значение превышает наибольшее машинное целое без знака, то оно представляется как длинное целое.

3.2. Длинные целые константы

Длинная целая константа явно определяется латинской буквой l или L, стоящей после константы.

Примеры

Длинная десятичная: 12l = 12 (десятичное);

956L = 956 (десятичное);

Длинная восьмеричная: 012l = 10 (десятичное);

076L = 62 (десятичное);

Длинная шестнадцатеричная: 0x12l = 18 (десятичное);

0XA3L = 163 (десятичное).

3.3. Константы с плавающей точкой

Константа с плавающей точкой всегда представляется числом с плавающей точкой двойной точности, т. е. как имеющая тип double, и состоит из следующих частей:

целой части – последовательности цифр;

десятичной точки;

дробной части – последовательности цифр;

символа экспоненты e или E;

экспоненты в виде целой константы (может быть со знаком).

Любая часть (но не обе сразу) из нижеследующих пар может быть опущена:

целая или дробная часть;

десятичная точка или символ e (E) и экспонента в виде целой константы.

Примеры

345. = 345 (десятичное);

3.14159 = 3.14159 (десятичное);

2.1E5 = 210000 (десятичное);

.123E3 = 123 (десятичное);

4037e-5 = .04037 (десятичное).

3.4. Символьные константы

Символьная константа состоит из одного символа кода ASCII¹, заключенного в апострофы (см. с. 89).

Примеры

'A' 'a' '7' '\$'

Специальные (управляющие) символьные константы

Новая строка (перевод строки)	HL (LF)	'\n'
Горизонтальная табуляция	HT	'\t'
Вертикальная табуляция	VT	'\v'
Возврат на шаг	BS	'\b'
Возврат каретки	CR	'\r'
Перевод формата	FF	'\f'
Обратная косая	'\'	'\\'
Апостроф	'\''	'\"'
Кавычки	'\"'	'\'''
Нулевой символ (пусто)	NUL	'\0'

Кроме этого любой символ может быть представлен последовательностью трех восьмеричных цифр: '\ddd'.

З а м е ч а н и е. Символьные константы считаются данными типа int.

3.5. Строковые константы

Строковая константа представляется последовательностью символов кода ASCII, заключенной в кавычки: "...".

Примеры

"This is a character string"

"Это строковая константа"

"A" "1234567890" "0" "\$"

Строковая константа — это массив символов, заключенный в кавычки; она имеет тип char [].

В конце каждой строки компилятор помещает нулевой символ '\0', отмечающий конец данной строки.

¹ Американский стандартный код для обмена информацией. — Прим. перев.

Каждая строковая константа, даже если она идентична другой строковой константе, сохраняется в отдельном месте памяти. Если необходимо ввести в строку символ кавычек (") , то перед ним надо поставить символ обратной косой (\). В строку могут быть введены любые специальные символьные константы, перед которыми стоит символ \.

Символ \ и следующий за ним символ новой строки игнорируются.

3.6. Перечислимые константы

Имена, указанные в описании перечислимых констант, трактуются как целые константы (см. с. 46).

3.7. Размер данных

Следующая таблица дает размер в битах основных типов данных для различных ЭВМ¹. Адрес слова определяется по байту с нулевым номером.

Основные типы данных	3B Computer Код ASCII	DEC PDP-11 Код ASCII	DEC VAX Код ASCII
char	8	8	8
int	32	16	32
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
Диапазон float	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
Диапазон double	$\pm 10^{\pm 308}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
Порядок байт в слове	0123	1032	3210

¹ Аналогичные данные для советских ЭВМ, на которых работает компилятор языка Си, представлены в таблице на с. 17. — Прим. перев.

Основные типы данных	HONEYWELL 6000 Код ASCII	IBM 360/370 Код EBCDIC	INTER- DATA 8/32 Код ASCII
char	9	8	8
int	36	32	32
short	36	16	16
long	36	32	32
float	36	32	32
double	72	64	64
Диапазон float	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$
Диапазон double	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$
Порядок байт в слове	0123	0123	

З а м е ч а н и е. Следующая таблица представляет предварительные данные. Отдельные позиции (обозначенные **) могут быть изменены.

Основные типы данных	MOTOROLA 68000 Код ASCII	NSC 16000 Код ASCII	ZILOG 8000 Код ASCII	INTEL 80286 Код ASCII
char	8	8	8	8
int	32**	32	16**	16
short	16	16	16	16
long	32	32	32	32**
float	32	32	32	32
double	32	64	64**	64
Диапазон float	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
Диапазон double	$\pm 10^{\pm 38}$	$\pm 10^{\pm 308}$	$\pm 10^{\pm 308}$	$\pm 10^{\pm 308}$
Порядок байт в слове	0123	3210	0123	3210

Основные типы данных	CM 4, CM 1420 КОИ-8	ЕС ЭВМ ДКОИ	CM 1700 КОИ-8
char	8	8	8
int	16	32	32
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
Диапазон float	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 38}$
Диапазон double	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 38}$
Порядок байт в слове	1032	0123	3210

Код КОИ-8 включает в качестве подмножества код ASCII, код ДКОИ включает в качестве подмножества код EBCDIC.

4. ОПЕРАЦИИ И ВЫРАЖЕНИЯ

4.1. Выражения

Выражение состоит из одного или большего числа операндов и символов операций.

Пр и м е р ы

a++ b = 10 x = (y*z)/w

З а м е ч а н и е. Выражение, заканчивающееся точкой с запятой, является оператором (см. с. 32).

4.2. Метаобозначения операндов

Некоторые операции требуют операндов определенного вида. Вид операнда обозначается одной из следующих букв:

- e** — любое выражение;
- v** — любое выражение, ссылающееся на переменную, которой может быть присвоено значение. Такие выражения называются адресными.

Префикс указывает тип выражения. Например, **ie** обозначает произвольное целое выражение. Далее описываются все возможные префиксы:

- i** — целое число или символ;
- a** — арифметическое выражение (целое число, символ или число с плавающей точкой);
- p** — указатель;
- s** — структура или объединение;
- sp** — указатель на структуру или объединение;
- f** — функция;
- fp** — указатель на функцию.

Обозначение **stet** указывает на имя элемента структуры или объединения.

З а м е ч а н и е. Если в выражении должно быть несколько операндов, то они отличаются номерами, например: **ae1 + ae2**.

4.3. Арифметические операции

- +** **Использование:** **ae1 + ae2**
Сумма значений **ae1** и **ae2**.

П р и м е р

i = j + 2;

Устанавливает **i** равным **j** плюс 2.

- +** **Использование:** **pe + ie**

Адрес переменной типа **pe**, больший на **ie** адреса, заданного указателем **pe**.

П р и м е р

last = arname + arsize - 1;

Присваивает переменной **last** адрес последнего элемента массива **arname**.

- Использование:** **ae1 - ae2**

Разность значений **ae1** и **ae2**.

П р и м е р

i = j - 3;

- **Использование:** **pe - ie**

Адрес переменной типа **pe**, меньший на **ie** адреса, заданного указателем **pe**.

П р и м е р

first = last - arsize + 1;

- **Использование:** **pe1 - pe2**

Число переменных типа **pe** в диапазоне от **pe2** до **pe1**.

П р и м е р

arsize = last - first;

- **Использование:** **-ae**

Изменение знака **ae**.

П р и м е р

x = -x;

- *** **Использование:** **ae1 * ae2**

Произведение значений **ae1** и **ae2**.

П р и м е р

z = 3 * x;

- /** **Использование:** **ae1 / ae2**

Частное от деления **ae1** на **ae2**.

П р и м е р

i = j / 5;

- %** **Использование:** **ae1 % ae2**

Остаток от деления (деление по модулю) **ae1** на **ae2**.

П р и м е р

minutes = time % 60;

З а м е ч а н и е: при выполнении операций **++** и **--** появляется побочный эффект — изменяется значение переменной, используемой в качестве операнда.

- ++** **Использование:** **iv++**

Увеличение **iv** на 1. Значением этого выражения является значение **iv** до увеличения.

П р и м е р

j = i++;

++ Использование: `rv++`

Увеличение указателя `rv` на 1, так что он будет указывать на следующий объект того же типа. Значением этого выражения является значение `rv` до увеличения.

Пример

```
*ptr++ = 0;
```

Присвоить значение 0 переменной, на которую указывает `ptr`, затем увеличить значение указателя `ptr` так, чтобы он указывал на следующую переменную того же типа.

++ Использование: `++iv`

Увеличение `iv` на 1. Значением этого выражения является значение `iv` после увеличения.

Пример

```
i = ++j;
```

++ Использование: `++rv`

Увеличение `rv` на 1. Значением этого выражения является значение `rv` после увеличения.

Пример

```
*++ptr = 0;
```

-- Использование: `iv--`

Уменьшение `iv` на 1. Значением этого выражения является значение `iv` до уменьшения.

Пример

```
j = i--;
```

-- Использование: `rv--`

Уменьшение указателя `rv` на 1 так, что он будет указывать на предыдущий объект того же типа. Значением этого выражения является значение `rv` до уменьшения.

Пример

```
atpos = r--;
```

-- Использование: `--iv`

Уменьшение `iv` на 1. Значением этого выражения является значение `iv` после уменьшения.

Пример

```
i = --j;
```

-- Использование: `--rv`

Уменьшение `rv` на 1. Значением этого выражения является значение `rv` после уменьшения.

Пример

```
prepos = --p;
```

4.4. Операция присваивания

З а м е ч а н и е. Значением выражения, в которое входит операция присваивания, является значение левого операнда после присваивания.

= Использование: `v = e`

Присваивание значения `e` переменной `v`.

Пример

```
x = y;
```

З а м е ч а н и е. Следующие операции объединяют арифметические или побитовые операции с операцией присваивания.

+= Использование: `av += ae`

Увеличение `av` на `ae`.

Пример

```
y += 2;
```

Увеличение переменной `y` на 2.

+= Использование: `rv += ie`

Увеличение `rv` на `ie`.

Пример

```
r += n;
```

-= Использование: `av -= ae`

Уменьшение `av` на `ae`.

Пример

```
x -= 3;
```

-= Использование: `rv -= ie`

Уменьшение `rv` на `ie`.

Пример

```
ptr -= 2;
```


***=** Использование: `av *= ae`

Умножение `av` на `ae`.

Пример

`timesx *= x;`

/= Использование: `av /= ae`

Деление `av` на `ae`.

Пример

`x /= 2;`

%= Использование: `iv %= ie`

Значение `iv` по модулю `ie`.

Пример

`x %= 10;`

>>= Использование: `iv >>= ie`

Сдвиг двоичного представления `iv` вправо на `ie` бит.

Пример

`x >>= 4;`

<<= Использование: `iv <<= ie`

Сдвиг двоичного представления `iv` влево на `ie` бит.

Пример

`x <<= 1;`

&= Использование: `iv &= ie`

Побитовая операция И двоичных представлений `iv` и `ie`.

Пример

`remitems &= mask;`

^= Использование: `iv ^= ie`

Побитовая операция исключающее ИЛИ двоичных представлений `iv` и `ie`.

Пример

`control ^= seton;`

|= Использование: `iv |= ie`

Побитовая операция ИЛИ двоичных представлений `iv` и `ie`.

Пример

`additems |= mask;`

4.5. Операции отношения

З а м е ч а н и е. Логическое значение Ложь представляется целым нулевым значением, а значение Истина представляется любым ненулевым значением.

Значением выражений, содержащих операции отношения или логические операции, является 0 (Ложь) или 1 (Истина).

== Использование: `ie1 == ie2`

Истина, если `ie1` равно `ie2`; иначе — Ложь.

Пример

`if (i == 0)`

`break;`

== Использование: `pe1 == pe2`

Истина, если значения указателей `pe1` и `pe2` равны.

!= Использование: `ie1 != ie2`

Истина, если `ie1` не равно `ie2`.

Пример

`while (i != 0)`

`i = func;`

!= Использование: `pe1 != pe2`

Истина, если значения указателей `pe1` и `pe2` не равны.

Пример

`if (p != q)`

`break;`

< Использование: `ae1 < ae2`

Истина, если `ae1` меньше, чем `ae2`.

Пример

`if (x < 0)`

`printf ("negative");`

< Использование: `pe1 < pe2`

Истина, если значение `pe1` (т. е. некоторый адрес) меньше, чем значение `pe2`.

Пример

`while (p < q)`

`*p++ = 0;`

Пока адрес, заданный *p*, меньше, чем адрес, заданный *q*, присваивать значение 0 переменной, на которую указывает *p*, и увеличивать значение *p* так, чтобы этот указатель указывал на следующую переменную.

- <=** Использование: *ae1 <= ae2*
Истина, если *ae1* меньше или равно *ae2*.
- <=** Использование: *pe1 <= pe2*
Истина, если *pe1* меньше или равно *pe2*.
- >** Использование: *ae1 > ae2*
Истина, если *ae1* больше, чем *ae2*.
- Пример**
if (*x* > 0)
printf ("positive");
- >** Использование: *pe1 > pe2*
Истина, если значение *pe1* (т. е. некоторый адрес), больше, чем значение *pe2*.
- Пример**
while (*p* > *q*)
**p*-- = 0;
- >=** Использование: *ae1 >= ae2*
Истина, если *ae1* больше или равно *ae2*.
- >=** Использование: *pe1 >= pe2*
Истина, если значение *pe1* больше или равно значению *pe2*.

4.6. Логические операции

- !** Использование: *!ae* или *!pe*
Истина, если *ae* или *pe* ложно.
- Пример**
if (!good)
printf ("not good");
- ||** Использование: *e1 || e2*
Логическая операция ИЛИ значений *e1* и *e2*. Вначале проверяется значение *e1*; значение *e2* проверяется только в том случае,

если значение *e1* – Ложь. Значением выражения является Истина, если истинно значение *e1* или *e2*.

Пример

```
if (x < A || x > B)
    printf ("out of range");
```

&& Использование: *e1 && e2*

Логическая операция И значений *e1* и *e2*. Вначале проверяется значение *e1*; значение *e2* проверяется только в том случае, если значение *e1* – Истина. Значением выражения является Истина, если значения *e1* и *e2* – Истина.

Пример

```
if (p != NULL && *p > 7)
    n++;
```

Если *p* – не нулевой указатель и значение переменной, на которую указывает *p*, больше, чем 7, то увеличить *n* на 1. Обратите внимание, что если значение указателя *p* равно NULL (0), то выражение **p* не имеет смысла.

4.7. Побитовые операции

~ Использование: *~ie*

Дополнение до единицы значения *ie*. Значение выражения содержит 1 во всех разрядах, в которых *ie* содержит 0, и 0 во всех разрядах, в которых *ie* содержит 1.

Пример

```
opposite = ~mask;
```

>> Использование: *ie1 >> ie2*

Двоичное представление *ie1* сдвигается вправо на *ie2* разрядов. Сдвиг вправо может быть арифметическим (т. е. освобождающиеся слева разряды заполняются значением знакового разряда) или логическим в зависимости от реализации, однако гарантируется, что сдвиг вправо целых чисел без знака будет логическим и освобождающиеся слева разряды будут заполняться нулями.

Пример

```
x = x >> 3;
```

<< Использование: ie1 << ie2

Двоичное представление ie1 сдвигается влево на ie2 разрядов; освобождающиеся справа разряды заполняются нулями.

Пример

```
fourx = x << 2;
```

& Использование: ie1 & ie2

Побитовая операция И двоичных представлений ie1 и ie2. Значение выражения содержит 1 во всех разрядах, в которых и ie1 и ie2 содержат 1, и 0 во всех остальных разрядах.

Пример

```
flag = ((x & mask) != 0);
```

Использование: ie1 | ie2

Побитовая операция ИЛИ двоичных представлений ie1 и ie2. Значение выражения содержит 1 во всех разрядах, в которых ie1 или ie2 содержит 1, и 0 во всех остальных разрядах.

Пример

```
attrsum = attr1 | attr2;
```

Использование: ie1 ^ ie2

Побитовая операция исключающее ИЛИ двоичных представлений ie1 и ie2. Значение выражения содержит 1 в тех разрядах, в которых ie1 и ie2 имеют разные двоичные значения, и 0 во всех остальных разрядах.

Пример

```
diffbits = x ^ y;
```

4.8. Адресные операции**& Использование: &v**

Значением выражения является адрес переменной v.

Пример

```
intptr = &n;
```

*** Использование: *pe**

Значением выражения является переменная, адресуемая указателем pe.

Пример

```
*ptr = c;
```

*** Использование: *fpe**

Значением выражения является функция, адресуемая указателем fpe.

Пример

```
fpe = funcname;  
(*fpe) (arg1, arg2);
```

4.9. Операции над массивами**[] Использование: pe [ie]**

Значением выражения является переменная, отстоящая на ie переменных от адреса, заданного pe. Это значение эквивалентно значению выражения $*(pe + ie)$.

Пример

```
aname[i] = 3;
```

Присвоить значение 3 i-му элементу массива aname.

Обратите внимание, что первый элемент массива описывается выражением `aname[0]`.

4.10. Операции над структурами или объединениями**Использование: sv.smem**

Значением выражения является элемент smem структуры или объединения sv.

Пример

```
product.p_revenue = 50;
```

Присвоить значение 50 элементу p_revenue структурной переменной product.

-> Использование: spe -> smem

Значением выражения является элемент smem структуры (или объединения), на которую(ое) указывает spe. Это значение эквивалентно значению выражения $*(spe).smem$

Пример

```
prodptr -> p_revenue = 2;
```

Присвоить значение 2 элементу p_revenue структурной переменной, на которую указывает prodptr.

4.11. Другие операции

?: **Использование:** $ae ? e1 : e2$ или $re ? e1 : e2$
Если истинно ae или re , то выполняется $e1$; иначе выполняется $e2$. Значением этого выражения является значение выражения $e1$ или $e2$.

Пример

$abs = (i \leq 0) ? -i : i;$

Использование: $e1, e2$

Сначала выполняется выражение $e1$, потом выражение $e2$. Значением всего выражения является значение выражения $e2$.

Пример

$for (i = A, j = B; i < j: i++, j--)$
 $p[i] = p[j];$

sizeof **Использование:** $sizeof(e)$

Число байт, требуемых для размещения данных типа e . Если e описывает массив, то в этом случае e обозначает весь массив, а не только адрес первого элемента, как во всех остальных операциях.

sizeof **Использование:** $sizeof(тип)$

Число байт, требуемых для размещения объектов типа $тип$.

Пример

$n = sizeof(арname) / sizeof(int);$

Число элементов в массиве целых чисел, определяемое как число байт в массиве, поделенное на число байт, занимаемых одним элементом массива.

(тип) **Использование:** $(тип) e$

Значение e , преобразованное в тип данных $тип$.

Пример

$x = (float)n / 3;$

Целое значение переменной n преобразуется в число с плавающей точкой перед делением на 3.

() **Использование:** $fe(e1, e2, \dots, eN)$

Вызов функции fe с аргументами $e1, e2, \dots, eN$.

Значением выражения является значение, возвращаемое функцией. Обратите внимание, что порядок выполнения выражений $e1, \dots, eN$ не гарантируется (см. с. 39).

Пример

$x = \text{sqrt}(y);$

4.12. Приоритеты и порядок выполнения операций

Для каждой группы операций в нижеследующей таблице приоритеты одинаковы. Чем выше приоритет группы операций, тем выше она расположена в таблице. Порядок выполнения определяет группировку операций и операндов (слева направо или справа налево), если отсутствуют скобки и операции относятся к одной группе.

Примеры

Выражение $a * b / c$ эквивалентно выражению $(a * b) / c$, так как операции выполняются слева направо.

Выражение $a = b = c$ эквивалентно выражению $a = (b = c)$, так как операция выполняется справа налево.

()	Вызов функции (с. 28)	Слева направо
[]	Выделение элемента массива (с. 27)	
.	Выделение элемента структуры или объединения (с. 27)	
->	Выделение элемента структуры (объединения), адресуемой (го) указателем (с. 27)	
!	Логическое отрицание (с. 24)	Справа налево
~	Побитовое отрицание (с. 25)	
-	Изменение знака (с. 19)	
++	Увеличение на единицу (с. 19)	
--	Уменьшение на единицу (с. 20)	
&	Определение адреса (с. 26)	
*	Обращение по адресу (с. 26)	
(тип)	Преобразование типа (с. 28)	
sizeof	Определение размера в байтах (с. 28)	

* / %	Умножение (с. 19) Деление (с. 19) Деление по модулю (с. 19)	Слева направо
+ -	Сложение (с. 18) Вычитание (с. 19)	Слева направо
<< >>	Сдвиг влево (с. 26) Сдвиг вправо (с. 25)	Слева направо
< <= > >=	Меньше, чем (с. 23) Меньше или равно (с. 24) Больше, чем (с. 24) Больше или равно (с. 24)	Слева направо
== !=	Равно (с. 23) Не равно (с. 23)	Слева направо
&	Побитовая операция И (с. 26)	Слева направо
^	Побитовая операция исключающее ИЛИ (с. 26)	Слева направо
	Побитовая операция ИЛИ (с. 26)	Слева направо
&&	Логическая операция И (с. 25)	Слева направо
	Логическая операция ИЛИ (с. 24)	Слева направо
?:	Условная операция (с. 28)	Справа налево
= *= <<= >>= &= ^= =	Присваивание (с. 21–22) /= %= += -= <<= >>= &= ^= =	Справа налево
Операция запятая (с. 28)		Слева направо

4.13. Порядок обработки операндов

Для четырех операций (&& || ?: ,) гарантируется, что левый операнд будет обрабатываться первым. Для остальных операций порядок обработки может быть разным на разных компиляторах. Это означает, что если программа, отлаженная на некоторой ЭВМ, зависит от негарантированного порядка обработки операндов, то на другой ЭВМ с другим компилятором она может выполняться неправильно.

Пример

```
v = (x = 5) + (++x);
```

Если порядок обработки операндов в операции + слева направо, то переменная v получит значение 11 (5 + 6) и значение x будет равно 6.

Если порядок справа налево, значение v зависит от значения x, которое эта переменная имела до выполнения выражения; например, если значение x было равно 0, то значение v станет равным 6 (5 + 1) и значение x станет равным 5.

Предупреждение. Если вы присваиваете переменной значение в любом выражении (включая вызов функции), то не используйте эту переменную снова в том же выражении. Например, если в предыдущем примере необходим порядок обработки слева направо, сделайте так:

```
x = 5;  
v = x + (x + 1);  
++x;
```

4.14. Арифметические преобразования в выражениях

Прежде всего каждый операнд типа char или short преобразуется в значение типа int и операнды типа unsigned char или unsigned short преобразуются в значение типа unsigned int¹.

Затем если один из операндов имеет тип double, то другой преобразуется в значение типа double и результат будет иметь тип double.

¹ Кроме того, операнды типа float до начала операции преобразуются в значение типа double [Д6]. — Прим. перев.

Иначе если один из операндов имеет тип `unsigned long`, то другой преобразуется в значение типа `unsigned long` и таким же будет тип результата.

Иначе если один из операндов имеет тип `long`, то другой преобразуется в значение типа `long` и таким же будет тип результата.

Иначе если один из операндов имеет тип `long`, а другой – тип `unsigned int`, то оба операнда преобразуются в значение типа `unsigned long` и результат будет иметь тип `unsigned long`.

Иначе если один из операндов имеет тип `unsigned`, то другой преобразуется в значение типа `unsigned` и результат будет иметь тип `unsigned`.

Иначе оба операнда должны быть типа `int` и таким же будет тип результата.

5. ОПЕРАТОРЫ

5.1. Формат и вложенность

Формат. Один оператор может занимать одну или более строк. Два или большее количество операторов могут быть расположены на одной строке.

Вложенность. Операторы, управляющие порядком выполнения (`if`, `if-else`, `switch`, `while`, `do-while` и `for`), могут быть вложены друг в друга.

5.2. Метка оператора

Метка может стоять перед любым оператором, чтобы на этот оператор можно было перейти с помощью оператора `goto`.

Метка состоит из идентификатора, за которым стоит двоеточие (`:`). Областью определения метки является данная функция.

Пример

```
ABC2: x = 3;
```

5.3. Составной оператор

Составной оператор (блок) состоит из одного или большего числа операторов любого типа, заключенных в фигурные скобки (`{ }`).

После закрывающейся фигурной скобки не должно быть точки с запятой (`;`).

Пример

```
{ x = 1; y = 2; z = 3; }
```

5.4. Оператор-выражение

Любое выражение, заканчивающееся точкой с запятой (`;`), является оператором. Далее следуют примеры операторов-выражений.

Оператор присваивания

Идентификатор = *выражение*;

Пример

```
x = 3;
```

Оператор вызова функции

Имя_функции (*аргумент1*, ..., *аргументN*);

Пример

```
fclose (file);
```

Пустой оператор

Состоит только из точки с запятой (`;`).

Используется для обозначения пустого тела управляющего оператора.

5.5. Оператор завершения `break`

`break;`

Прекращает выполнение ближайшего вложенного внешнего оператора `switch`, `while`, `do` или `for`. Управление передается оператору, следующему за заканчиваемым. Одно из назначений этого оператора – закончить выполнение цикла при присваивании некоторой переменной определенного значения.

Пример

```
for (i = 0; i < n; i++)
    if ((a[i] = b[i]) == 0)
        break;
```


5.6. Оператор продолжения continue

continue;

Передаёт управление в начало ближайшего внешнего оператора цикла while, do или for, вызывая начало следующей итерации. Этот оператор по действию противоположен оператору break.

Пример

```
for (i = 0; i < n; i++) {
    if (a[i] != 0)
        continue;
    a[i] = b[i];
    k++;
}
```

5.7. Оператор возврата return

return;

Прекращает выполнение текущей функции и возвращает управление вызвавшей программе.

return *выражение*;

Прекращает выполнение текущей функции и возвращает управление вызвавшей программе с передачей значения *выражения*.

Пример

return x+y;

5.8. Оператор перехода goto

goto *метка*;

Управление безусловно передается на оператор с меткой *метка*. Используется для выхода из вложенных управляющих операторов. Область действия ограничена текущей функцией.

Пример

goto ABC;

5.9. Условный оператор if-else

if (*выражение*)
 оператор

Если *выражение* истинно, то выполняется *оператор*.
Если *выражение* ложно, то ничего не делается.

Пример

```
if (a == x)
    temp = 3;
temp = 5;
```

if (*выражение*)
 оператор1

else

оператор2

Если *выражение* истинно, то выполняется *оператор1* и управление передается на оператор, следующий за *оператором2* (т. е. *оператор2* не выполняется).

Если *выражение* ложно, то выполняется *оператор2*.

Часть else оператора может опускаться. Поэтому во вложенных операторах if с пропущенной частью else может возникнуть неоднозначность. В этом случае else связывается с ближайшим предыдущим оператором if в том же блоке, не имеющим части else.

Примеры

Часть else относится ко второму оператору if:

```
if (x > 1)
    if (y == 2)
        z = 5;
    else
        z = 6;
```

Часть else относится к первому оператору if

```
if (x > 1) {
    if (y == 2)
        z = 5;
} else
    z = 6;
```

Вложенные операторы if:

```
if (x == 'a')
    y = 1;
```

```

else if (x == 'b') {
    y = 2;
    z = 3;
} else if (x == 'c')
    y = 4;
else
    printf("ERROR");

```

5.10. Оператор-переключатель switch

```

switch (выражение) {
    case константа: операторы
    case константа: операторы
    ...
    default: операторы
}

```

Сравнивает значение *выражения* с *константами* во всех вариантах case и передает управление *оператору*, который соответствует значению *выражения*.

Каждый вариант case может быть помечен целой или символьной *константой*, или константным выражением. Константное выражение не может включать переменные или вызовы функций¹.

Примеры

Правильно: case 3+4:

Неправильно: case X+Y:

Операторы, связанные с меткой default, выполняются, если ни одна из *констант* в операторах case не равна значению *выражения*.

Вариант default не обязательно должен быть последним.

Если ни одна *константа* не соответствует значению *выражения* и вариант default отсутствует, то не выполняется никаких действий.

Ключевое слово case вместе с *константой* служат просто метками, и если будут выполняться операторы для некоторого варианта case, то далее будут выполняться операторы всех последующих вариан-

¹ Константное выражение вычисляется в период компиляции. —

тов до тех пор, пока не встретится оператор break. Это позволяет связывать одну последовательность операторов с несколькими вариантами.

Никакие две *константы* в одном операторе-переключателе не могут иметь одинаковые значения.

Пример

```

switch (x) {
    case 'A':
        printf("CASE A\n");
        break;
    case 'B':
    case 'C':
        printf("CASE B or C\n");
        break;
    default:
        printf("NOT A, B or C\n");
        break;
}

```

Наиболее общая синтаксическая форма оператора switch:

switch (выражение) оператор

Пример

```

switch (x)
case 2:
case 4:
    y = 3;

```

5.11. Оператор цикла while

```

while (выражение)
    оператор

```

Если *выражение* истинно, то *оператор* выполняется до тех пор, пока *выражение* не станет ложным.

Если *выражение* ложно, то управление передается следующему оператору.

З а м е ч а н и е. Значение *выражения* определяется до выполнения *оператора*. Следовательно, если *выражение* ложно с самого начала, то *оператор* вообще не выполняется.

П р и м е р

```
while (k < n) {
    y = y * x;
    k++;
}
```

5.12. Оператор цикла do-while

do

оператор

while (*выражение*);

Оператор выполняется. Если *выражение* истинно, то *оператор* выполняется и вычисляется значение *выражения*; это повторяется до тех пор, пока *выражение* не станет ложным.

Если *выражение* ложно, то управление передается следующему оператору.

З а м е ч а н и е. Значение *выражения* определяется после выполнения *оператора*. Поэтому *оператор* выполняется хотя бы один раз. Оператор do-while проверяет условие в конце цикла.

Оператор while проверяет условие в начале цикла.

П р и м е р

```
x = 1;
do
    printf("%d\n", power(x, 2));
while (++x <= 7);
```

5.13. Оператор цикла for

```
for (выражение1;
    выражение2;
    выражение3)
    оператор
```

Выражение1 описывает инициализацию цикла.

Выражение2 — проверка условия завершения цикла. Если оно истинно, то выполняется *оператор* тела цикла for,

выполняется *выражение3*,

все повторяется, пока *выражение2* не станет ложным.

Если оно ложно, цикл заканчивается и управление передается следующему оператору.

Выражение3 вычисляется после каждой итерации.

Оператор for эквивалентен следующей последовательности операторов:

```
выражение1;
while (выражение2) {
    оператор
    выражение3;
}
```

П р и м е р

```
for (x = 1; x <= 7; x++)
    printf("%d\n", power(x, 2));
```

Любое из трех или все три *выражения* в операторе for могут отсутствовать, однако разделяющие их точки с запятыми (;) опускать нельзя.

Если опущено *выражение2*, то считается, что оно постоянно истинно. Оператор for(;;) представляет собой бесконечный цикл, эквивалентный оператору while(1).

Каждое из *выражений1–3* может состоять из нескольких выражений, объединенных оператором запятая (,).

П р и м е р

```
for (i=0, j=n-1; i < n; i++, j--)
    a[i] = a[j];
```

6. ФУНКЦИИ

6.1. Определение функции

Функция определяется описанием типа результата, формальных параметров и составного оператора (блока), описывающего выполняемые функцией действия.

Пример

double	тип результата
linfunc (x, a, b)	имя функции список параметров
double x;	описание параметров
double a;	
double b;	
{	составной оператор
return (a*x + b);	возвращаемое значение
}	

Оператор return может не возвращать никакого значения или возвращает значение выражения, стоящего в этом операторе.

Значение выражения при необходимости преобразуется к типу результата функции.

Функция, которая не возвращает значения, должна быть описана как имеющая тип void.

Пример

```
void
errmesg(s)
char * s;
{
    printf("***%s\n", s);
}
```

6.2. Вызов функции

Существуют два способа вызова функции:

имя_функции (e1, e2, ..., eN)

(**указатель_на_функцию*) (e1, e2, ..., eN)

Указатель_на_функцию — это переменная, содержащая адрес функции. Адрес функции может быть присвоен указателю оператором

указатель_на_функцию = *имя_функции*;

Аргументы (фактические параметры) передаются по значению, т. е. каждое выражение e1, ..., eN вычисляется и значение передается функции, например, загрузкой в стек.

Порядок вычисления выражений и порядок загрузки значений в стек не гарантируются.

Во время выполнения не производится проверка числа или типа аргументов, переданных функции. Такую проверку можно произвести с помощью программы lint до компиляции (см. с. 78).

Вызов функции — это выражение, значением которого является значение, возвращаемое функцией.

Описанный тип функции должен соответствовать типу возвращаемого значения. Например, если функция linfunc возвращает значение типа double, то эта функция должна быть описана до вызова:

```
extern double linfunc();
```

З а м е ч а н и е. Такое описание не определяет функцию, а только описывает тип возвращаемого значения; оно не нужно, если функция определена в том же файле до ее вызова (см. с. 42).

Примеры

Правильно: extern double linfunc();
 float y;
 y = linfunc (3.05, 4.0, 1e-3);

Значение функции перед присваиванием переменной y преобразуется из типа double в тип float.

Неправильно: float x;
 float y;
 x = 3.05;
 y = linfunc (x, 4, 1e-3);

Тип аргументов не соответствует типу параметров, описанных в определении функции, а именно: константа 4 имеет тип int, а не double. В результате аргументы, загруженные в стек, имеют неправильный тип и формат, поэтому значения, выбираемые из стека, бессмысленны и значение, возвращаемое функцией, не определено. Кроме того, если тип функции не описан, то считается, что возвращаемое значение имеет тип int. Поэтому, даже если функция linfunc возвращает правильное значение типа double, выражение, представляющее вызов функции, получит бессмысленное значение типа int (например, старшая половина значения double).

6.3. Функция main

Каждая программа начинает работу с функции `main()`. Во время выполнения программы можно передавать аргументы через формальные параметры `argc` и `argv` функции `main`. Переменные среды языка оболочки `shell` передаются программе через параметр `envp`.

Пример

```
/*
 * программа печатает значения фактических параметров,
 * а затем переменных среды
 */
main (argc, argv, envp)
int argc; /* число параметров */
char **argv; /* вектор параметров-строк */
char **envp; /* вектор переменных среды */
{
    register int i;
    register char **p;
    /* печать значений параметров */
    for (i = 0; i < argc; i++)
        printf ("arg %i:%s\n", i, argv [i]);
    /* печать значений переменных среды */
    for (p = envp; *p != (char*)0; p++)
        printf ("%s\n", *p);
}
```

З а м е ч а н и е. Параметры `argv` и `envp` могут быть описаны также следующим образом:

```
char *argv[];
char *envp[];
```

7. ОПИСАНИЯ

Описания используются для определения переменных и для объявления типов переменных и функций, определенных в другом месте. Описания также используются для определения новых типов дан-

ных на основе существующих типов. Описание не является оператором.

7.1. Основные типы

Примеры

```
char c;
int x;
```

Основными типами являются:

<code>char</code>	— символ (один байт);
<code>int</code>	— целое (обычно одно слово);
<code>unsigned</code>	— неотрицательное целое (такого же размера, как целое);
<code>short</code>	— короткое целое (слово или полуслово);
<code>long</code>	— длинное целое (слово или двойное слово);
<code>float</code>	— число с плавающей точкой (ординарной точности);
<code>double</code>	— число с плавающей точкой (двойной точности);
<code>void</code>	— отсутствие значения (используется для нейтрализации значения, возвращаемого функцией).

Символы (`char`) в зависимости от компилятора могут быть со знаком или без знака. Рассматриваемые как целые, символы со знаком имеют значения от -127 до 128 , а символы без знака — от 0 до 256 .

Некоторые реализации допускают явный тип `unsigned char`.

Данные целого типа `int` могут иметь такой же диапазон, как данные типа `long` или `short`.

Описание типа `unsigned` эквивалентно описанию типа `unsigned int`. Описание `unsigned` может сочетаться с описанием типа `char`, `short` или `long`, формируя описания типов `unsigned char`, `unsigned short`, `unsigned long`.

Описания типов `short` и `long` эквивалентны описаниям типов `short int` и `long int`.

Диапазон данных типа `long` обычно в два раза больше диапазона данных типа `short`.

7.2. Указатели и массивы

З а м е ч а н и е. Допустимо бесконечно большое число различных типов указателей и массивов. Далее следуют типовые примеры.

Указатель на основной тип**Пример**

```
char *p;
```

Переменная *p* является указателем на символ, т. е. этой переменной должен присваиваться адрес символа.

Указатель на указатель**Пример**

```
char **t;
```

Переменная *t* — указатель на указатель символа.

Одномерный массив**Пример**

```
int a[50];
```

Переменная *a* — массив из 50 целых чисел.

Двухмерный массив**Пример**

```
char m[7][50];
```

Переменная *m* — массив из семи массивов, каждый из которых состоит из 50 символов.

Массив из семи указателей**Пример**

```
char *r[7];
```

Массив *r* состоит из указателей на символы.

Указатель на функцию**Пример**

```
int (*f)();
```

f — указатель на функцию, возвращающую целое значение.

7.3. Структуры

Структура¹ объединяет логически связанные данные разных типов.

Структурный тип данных определяется следующим описанием:

```
struct имя_структуры {
    описания_элементов
};
```

¹ Иногда называют записью. — Прим. перев.

Пример

```
struct dinner {
    char      *place;
    float     cost;
    struct dinner *next;
};
```

Структурная переменная описывается с помощью структурного типа.

Примеры

```
struct dinner week_days [7]; /* массив структур */
struct dinner best_one;    /* одна структурная переменная */
struct dinner *p;          /* указатель на структурную переменную */
```

7.4. Поля бит в структурах

Поле бит — это элемент структуры, определенный как некоторое число бит, обычно меньшее, чем число бит в целом числе. Поля бит предназначены для экономного размещения в памяти данных небольшого диапазона.

Пример

```
struct bfeg {
    unsigned int bf_fld1 : 10;
    unsigned int bf_fld2 : 6;
};
```

Данная структура описывает 10-битовое поле, которое для вычислений преобразуется в значение типа `unsigned int`, и 6-битовое поле, которое обрабатывается как значение типа `unsigned int`.

7.5. Объединения

Объединение¹ описывает переменную, которая может иметь любой тип из некоторого множества типов.

¹ Иногда называют смесью. — Прим. перев.

Определение объединенного типа данных аналогично определению структурного типа данных:

```
union имя_объединения {
    описания_элементов
};
```

Пример

```
union bigword {
    long bg_long;
    char *bg_char [4];
};
```

Данные типа union bigword занимают память, необходимую для размещения наибольшего из своих элементов, и выравниваются в памяти к границе, удовлетворяющей ограничениям по адресации как для типа long, так и для типа char * [4].

Описание переменной объединенного типа

Пример

```
union bigword x;
union bigword *p;
union bigword a[100];
```

7.6. Перечисления

Данные перечислимого типа относятся к некоторому ограниченному множеству данных.

Определение перечислимого типа данных

```
enum имя_перечислимого_типа {
    список_значений
};
```

Каждое значение данного перечислимого типа задается идентификатором.

Пример

```
enum color {
    red, green, yellow
};
```

Описание переменной перечислимого типа

Пример

```
enum color chair;
enum color suite [40];
```

Использование переменной перечислимого типа в выражении

Пример

```
char = red;
suite [5] != yellow
```

7.7. Переименование типов

Формат

```
typedef старый_тип новый_тип
```

Примеры

```
typedef long large;
/* определяется тип large, эквивалентный типу long */
typedef char *string;
/* определяется тип string, эквивалентный типу char * */
```

Переименование типов используется для введения осмысленных или сокращенных имен типов, что повышает понятность программ, и для улучшения переносимости программ (имена одного типа данных могут различаться на разных ЭВМ).

7.8. Определение локальных переменных

З а м е ч а н и е 1. Постоянные переменные, сохраняемые в некоторой области памяти, инициализируются нулем, если явно не заданы начальные значения. Временные переменные, значения которых сохраняются в стеке или регистре, не получают начального значения, если оно не описано явно.

З а м е ч а н и е 2. Все описания в блоке должны предшествовать первому оператору.

Автоматические переменные

Пример

```
{
    int x; /* x — это автоматическая переменная */
    ...
}
```


Автоматическая переменная является временной, так как ее значение теряется при выходе из блока. Областью определения является блок, в котором эта переменная определена. Переменные, определенные в блоке, имеют приоритет перед переменными, определенными в охватывающих блоках.

Регистровые переменные

Пример

```
{
register int y;
...
}
```

Регистровые переменные являются временными, их значения сохраняются в регистрах, если последние доступны. Доступ к регистровым переменным более быстрый. В регистрах можно сохранять любые переменные, если размер занимаемой ими памяти не превышает разрядности регистра. Если компилятор не может сохранить переменные в регистрах, он трактует их как автоматические. Областью действия является блок. Операция получения адреса & не применима к регистровым переменным.

Формальные параметры

Примеры

int func(x);	int func(x)
int x;	register int x;
{	{
...	...
}	}

Формальные параметры являются временными, так как получают значения фактических параметров, передаваемых функции. Областью действия является блок функции. Формальные параметры должны отличаться по именам от внешних переменных и локальных переменных, определенных внутри функции. В блоке функции формальным параметрам могут быть присвоены некоторые значения.

Статические переменные

Пример

```
{
static int flag;
...
}
```

Статические переменные являются постоянными, так как их значения не теряются при выходе из функции. Любые переменные в блоке, кроме формальных параметров функции, могут быть определены как статические. Областью действия является блок.

7.9. Определение глобальных переменных

Глобальные переменные

Пример

```
int Global_flag;
```

Определяются на том же уровне, что и функции, т. е. не локальны ни в каком блоке. Постоянные. Инициализируются нулем, если явно не задано другое начальное значение. Областью действия является вся программа. Должны быть описаны во всех файлах программы, в которых к ним есть обращения. **З а м е ч а н и е.** Некоторые компиляторы требуют, чтобы глобальные переменные были определены только в одном файле и описаны как внешние в других файлах, где они используются (см. с. 51). Должны быть описаны в файле до первого использования.

Статические переменные

Пример

```
static int File_flag;
```

Постоянные. Областью действия является файл, в котором данная переменная определена. Должны быть описаны до первого использования в файле.

7.10. Инициализация переменных

Любая переменная, кроме формальных параметров или автоматических массива, структуры или объединения, при определении может быть инициализирована.

Любая постоянная переменная инициализируется нулем (0)¹, если явно не задано другое начальное значение.

В качестве начального значения может использоваться любое константное выражение.

Основные типы

Примеры

```
int i = 1;
float x = 3.145e - 2;
```

Массивы

Примеры

```
int a[] = {1, 4, 9, 16, 25, 36};
char s[20] = {'a', 'b', 'c'};
```

Список значений элементов массива должен быть заключен в фигурные скобки.

Если задан размер массива, то значения, не заданные явно, равны 0.

Если размер массива опущен, то он определяется по числу начальных значений.

Строки

Пример

```
char s[] = "hello";
```

Это описание эквивалентно описанию

```
char s[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Структуры

Пример

```
struct person {
    int height;
    char gender;
};
struct person x = {70, 'Y'};
struct person family[] = {
    {73, 'X'},
    {68, 'Y'},
    {50, 'X'}
};
```

¹ Это значит, что если переменная целая, то ее начальное значение равно 0, если символьная, то '\0', если это число с плавающей точкой, то 0.0. — Прим. перев.

Список значений для каждой структурной переменной должен быть заключен в фигурные скобки, хотя, если число значений соответствует числу структуры, это не обязательно.

Значения присваиваются элементам структуры в порядке размещения элементов в определении структурного типа.

Список значений может быть неполным, в этом случае неинициализированные элементы получают в качестве значения 0.

Пример

```
struct person people[10] = {
    {68},
    {71},
    {74}
};
```

Элементам height первых трех структурных переменных массива присваиваются явные значения; остальные переменные получают значение 0.

7.11. Описание внешних объектов

Тип внешних объектов (т. е. переменных или функций), определенных в другой компоненте программы, должен быть явно описан. Отсутствие такого описания может привести к ошибкам при компиляции, компоновке или выполнении программы.

При описании внешнего объекта используйте ключевое слово `extern`.

Примеры

```
extern int Global_var;
extern char *Name;
extern int func();
```

Можно опускать длину внешнего одномерного массива.

Пример

```
extern float Num_array[];
```

Поскольку все функции определены на внешнем уровне, то для описания функции внутри блока прилагательное `extern` избыточно и часто опускается.

Пример

```

{
...
int func();
...
}

```

Функция, не возвращающая значения, должна описываться как имеющая тип `void`. Если тип функции явно не задан, считается, что она имеет тип `int`. Областью действия описания на внешнем уровне является остаток файла; внутри блока областью действия является данный блок. Обычно внешние описания располагаются в начале файла.

Некоторые компиляторы допускают описание переменных на внешнем уровне без прилагательного `extern`. Многократные описания внешних переменных компоновщик сводит к одному определению.

8. ПРЕПРОЦЕССОР

Если в качестве первого символа в строке программы используется символ `#`, то эта строка является командной строкой препроцессора (макропроцессора). Командная строка препроцессора заканчивается символом перевода на новую строку. Если непосредственно перед концом строки поставить символ обратной косой (`\`), то командная строка будет продолжена на следующую строку программы.

8.1. Замена идентификаторов

`#define идентификатор строка`

Пример

```
#define ABC 100
```

Заменяет каждое вхождение идентификатора `ABC` в тексте программы на `100`.

`#undef идентификатор`

Пример

```
#undef ABC
```

Отменяет предыдущее определение для идентификатора `ABC`.

8.2. Макросы

З а м е ч а н и е. Во избежание ошибок при вычислении выражений параметры макроопределения необходимо заключать в скобки.

`#define идентификатор1 (идентификатор2, ...) строка`

Пример

```
#define abs(A) (((A) > 0) ? (A) : -(A))
```

Каждое вхождение выражения `abs(arg)` в тексте программы заменяется на `((arg) > 0) ? (arg) : -(arg)`, причем параметр макроопределения `A` заменяется на `arg`.

Пример

```
#define nmem (P, N)\
(P) -> p_mem [N].u_long
```

Символ `\` продолжает макроопределение на вторую строку. Это макроопределение уменьшает сложность выражения, описывающего массив объединений внутри структуры.

8.3. Включение файлов

З а м е ч а н и е. Командная строка `#include` может встречаться в любом месте программы, но обычно все включения размещаются в начале файла исходного текста.

`#include <имя_файла>`

Пример

```
#include <math.h>
```

Препроцессор заменяет эту строку содержимым файла `math.h`. Угловые скобки обозначают, что файл `math.h` будет взят из некоторого стандартного каталога (обычно это `/usr/include`). Текущий каталог не просматривается.

`#include "имя_файла"`

Пример

```
#include "ABC"
```

Препроцессор заменяет эту строку содержимым файла `ABC`. Так как имя файла заключено в кавычки, то поиск производится в текущем каталоге (в котором содержится основной файл исходного текста). Если в текущем каталоге данного

файла нет, то поиск производится в каталогах, определенных именем пути в опции `-I` препроцессора. Если и там файла нет, то просматривается стандартный каталог.

8.4. Условная компиляция

Командные строки препроцессора используются для условной компиляции различных частей исходного текста в зависимости от внешних условий.

`#if` *константное_выражение*

Пример

```
#if ABC + 3
```

Истина, если константное выражение `ABC + 3` не равно нулю.

`#ifdef` *идентификатор*

Пример

```
#ifdef ABC
```

Истина, если идентификатор `ABC` определен ранее командой `#define`.

`#ifndef` *идентификатор*

Пример

```
#ifndef ABC
```

Истина, если идентификатор `ABC` не определен в настоящий момент.

`#else`

...

`#endif`

Если предшествующие проверки `#if`, `#ifdef` или `#ifndef` дают значение **Истина**, то строки от `#else` до `#endif` игнорируются при компиляции.

Если эти проверки дают значение **Ложь**, то строки от проверки до `#else` (а при отсутствии `#else` — до `#endif`) игнорируются. Команда `#endif` обозначает конец условной компиляции.

Пример

```
#ifdef DEBUG
```

```
    fprintf(stderr, "location: x = %d\n", x);
```

```
#endif
```

8.5. Номер строки и имя файла

`#line` *целая_константа* *"имя_файла"*

Пример

```
#line 20 "ABC"
```

Препроцессор изменяет номер текущей строки и имя компилируемого файла. Имя файла может быть опущено.

9. СТРУКТУРА ПРОГРАММЫ

Программа, описанная в следующем примере, вводит до `MAXLINES` строк со стандартного входа, сортирует строки в лексикографическом порядке, возрастающем или уменьшающемся в зависимости от признака, передаваемого функции `main()` через аргумент `argv`. Затем программа записывает отсортированные строки на стандартный выход. Стандартными входом и выходом (`stdin` и `stdout`) могут быть терминал, канал или некоторый файл.

bblsort.h

A

```
#define MAXLINES 100
#define LINE_SIZE (132 + 1)
```

main.c

B

```
#include <stdio.h>
#include "bblsort.h"
```

C

```
char Line[MAXLINES][LINE_SIZE]; /* буфер строк */
int  Revflg;                     /* признак направления
                                * сортировки */
```

/*

* сортировка строк текста в лексикографическом порядке

*/

```

D  main(argc, argv)
   char **argv; /* аргументы вызова программы */
   int  argc;   /* число аргументов */
   {
E      int  rdlines();
      void bblsort(), wrlines();

F      int numlines;

G      Revflg = (argc > 1 && argv[1][0] == '-');
      numlines = rdlines();
      bblsort(numlines);
      wrlines(numlines);
   }
   /*
   * запись строк со стандартного выхода
   */

H   static int
   rdlines()
   {
      char      *fgets();

I      register int i;

      for (i = 0; i < MAXLINES; i++)
         if (fgets(Line[i], LINESIZE, stdin)

J          == (char *)NULL )
            break;
      return (i);
   }
   /*
   * запись строк на стандартный выход
   */

```

```

K   static void
   wrlines(n)

L   register int n; /* число строк */
   {
      register int i;

      for (i = 0; i < n; i++)
         fputs(Line[i], stdout);
   }

bblsort.c

M   # include "bblsort.h"

N   extern char Line[][LINESIZE];

   /*
   * bubble sort
   */

O   void
   bblsort(n)
   register int n; /* число строк */
   {
      int      lexcmp();
      void      swap();
      register int i, j;

      for (i = 1; i <= n - 1; i++)
         for (j = n - 1; j >= i; j--)
            if (lexcmp(j - 1, j))
               swap(j - 1, j);
   }

```

```

/*
 * лексикографическое сравнение двух строк
 */
static int
lexcmp(i, j)

P   register int i, j; /* элементы массива строк */
{

Q       int      strcmp();
       extern int Revflg;

       register int lc;

       lc = strcmp(Line[i], Line[j]);

R       return ((lc < 0 && Revflg)
               || (lc > 0 && !Revflg));
}
/*
 * обмен строк
 */
static void
swap(i, j)
register int i, j; /* элементы массива строк */
{
    char *strcpy();
    char temp[LINESIZE];

    strcpy(temp, Line[i]);
    strcpy(Line[i], Line[j]);
    strcpy(Line[j], temp);
}

```

Пояснения к программе

- A** Поименованные константы, используемые во всей программе, обычно помещаются в отдельный файл, включаемый в другие файлы программы по мере необходимости. Поэтому при изменении этих параметров программы будет затронут только один файл.
- B** Включаемые файлы обычно помещаются в начало некоторого файла программы. Файл `stdio.h` содержит описания файлов `stdin`, `stdout` и константы `NULL`, необходимых для использования функций `fgets()` и `fputs()`.
- C** Описания внешних переменных обычно размещаются в начале файла. В данной программе определены глобальный массив буфера строк и глобальный признак направления сортировки.
- D** Если функция `main()` использует формальные параметры, то они должны быть описаны. Функция `main()` выполняется первой.
- E** Типы функций, вызываемых в теле функции, обычно описываются в начале тела функции.
- F** Переменная `numlines` описана как локальная в блоке автоматическая целая переменная.
- G** По соглашению первый аргумент `argv`, передаваемый функции `main()`, является именем программы; `argv[1]` — это второй аргумент и `argv[1][0]` — это первый символ второго аргумента. Обратите внимание, что перед обращением к `argv[1][0]` проверяется число аргументов `argc`, так как при отсутствии второго аргумента выражение `argv[1][0]` не имеет смысла.
- H** Функция `rdlines()` возвращает целое число прочитанных строк, так как она определена как имеющая тип `int`. Прилагательное `static` указывает, что функция используется только в данном файле.
- I** Переменная `i` определена как локальная в блоке. Описание `register` — это попытка ускорить выполнение цикла `for`.
- J** Поскольку функция `fgets()` возвращает значение типа `char *`, то это значение должно сравниваться с указателем на символ. Поэтому нулевой указатель `NULL` преобразуется к типу `(char *)`.

- К** Функция `wrlines()` не возвращает значения, поэтому она определена как имеющая тип `void`.
- L** Формальный параметр `n` определен как `register` для ускорения цикла.
- M** Включение файла `bblsort.h` определяет поименованную константу `LINESIZE`.
- N** Это описание массива `Line` относится ко всему последующему файлу исходного кода. Здесь описывается тип `Line`, но сам массив определен в предыдущем файле.
- O** Функция `bblsort()` не описывается как `static`, потому что она вызывается функцией `main()`, которая определена в другом файле.
- P** Формальные параметры `i` и `j` объявлены с помощью одного описания. Порядок параметров в таком описании несуществен.
- Q** Глобальный признак `Revflg` должен быть описан как `extern`, чтобы показать, что эта переменная определена в другом файле. Это описание может располагаться в начале текущего файла вместе с описанием массива `Line`.
- R** Значением этого выражения является или Истина (1), или Ложь (0), кодируемые целыми значениями. Поэтому функция `lexstr()` определена как имеющая тип `int`.

10. БИБЛИОТЕКА ВВОДА-ВЫВОДА

Программа, использующая перечисленные ниже функции ввода-вывода, должна включать в себя файл `stdio.h` с помощью команды препроцессора

```
#include <stdio.h>
```

Файл `stdio.h` содержит:

1. Определение типа данных `FILE`.
2. Определения параметров, используемых в макровывозах и вызовах библиотечных функций.

Примеры

`stdin` — стандартный файл ввода;
`stdout` — стандартный файл вывода;

`stderr` — файл вывода сообщений об ошибках;
`NULL` — нулевой (0) указатель;
`EOF` — конец файла.

З а м е ч а н и е. По умолчанию файлы `stdin`, `stdout` и `stderr` связываются с терминалом.

3. Макроопределения:

<code>putc()</code>	<code>ferror()</code>
<code>getc()</code>	<code>clearerr()</code>
<code>putchar()</code>	<code>feof()</code>
<code>getchar()</code>	<code>fileno()</code>

З а м е ч а н и е. Поток ввода-вывода идентифицируется указателем на переменную типа `FILE`. Средства буферизации включаются в поток как часть стандартного пакета ввода-вывода.

10.1. Доступ к файлам

`fopen` — открыть поток ввода-вывода.

Определение: `FILE *fopen (filename, type)`
`char *filename, *type;`

`freopen` — закрыть поток `stream` и открыть файл `newfile`, используя описание этого потока.

Определение: `FILE *freopen (newfile, type, stream)`
`char *, newfile, *type;`
`FILE * stream;`

`fdopen` — связать поток с дескриптором файла, открытым функцией `open`.

Определение: `FILE *fdopen (fildes, type)`
`int fildes;`
`char *type;`

`fclose` — закрыть открытый поток ввода-вывода `stream`.

Определение: `int fclose (stream)`
`FILE *stream;`

`fflush` — записать символы из буфера в выходной поток `stream`.

Определение: `int fflush (stream)`
`FILE *stream;`

fseek — изменить текущую позицию offset в файле stream.

Определение: `int fseek (stream, offset, ptrname)`
`FILE *stream;`
`long offset;`
`int ptrname;`

rewind — переставить указатель текущего байта в потоке на начало файла.

Определение: `void rewind (stream)`
`FILE *stream;`

setbuf — модифицировать буфер потока.

Определение: `void setbuf (stream, buf)`
`FILE *stream;`
`char *buf;`

setvbuf — модифицировать буфер потока.

Определение: `int setvbuf (stream, buf, type, size)`
`FILE *stream;`
`char *buf;`
`int type, size;`

10.2. Доступ к каналам

pclose — закрыть поток, открытый функцией `popen`.

Определение: `int pclose (stream)`
`FILE *stream;`

popen — создать поток как канал обмена между процессами.

Определение: `FILE *popen (command, type)`
`char *command, *type;`

10.3. Состояние файла

clearerr — обнулить признаки ошибки потока.

Определение: `void clearerr (stream)`
`FILE *stream;`

feof — проверить состояние конца файла в потоке.

Определение: `int feof (stream)`
`FILE *stream;`

ferror — проверить состояние ошибки в потоке.

Определение: `int ferror (stream)`
`FILE *stream;`

fileno — связать дескриптор файла, открытого функцией `open` с существующим потоком.

Определение: `int fileno (stream)`
`FILE *stream;`

10.4. Форматированный ввод-вывод

Функции `printf`, `fprintf` и `sprintf` описаны в разд. 12.

Функции `scanf`, `fscanf` и `sscanf` описаны в разд. 13.

10.5. Ввод-вывод строк

fgets — прочитать строку из входного потока, включая символ новой строки.

Определение: `char *fgets (s, n, stream)`
`char *s;`
`int n;`
`FILE *stream;`

gets — прочитать строку из стандартного файла ввода `stdin`.

Определение: `char *gets(s)`
`char *s;`

fputs — записать строку в поток stream.

Определение: `int fputs (s, stream)`
`char *s;`
`FILE *stream;`

puts — записать строку в стандартный файл вывода `stdout`. В конце строки записывается символ новой строки.

Определение: `int puts (s)`
`char *s;`

10.6. Ввод символа

fgetc — прочитать следующий символ из входного потока stream.

Определение: `int fgetc (stream)`
`FILE *stream;`

З а м е ч а н и е. Функции `getc()`, `getchar()`, `ungetc()` являются макроопределениями.

`getc` — прочитать следующий символ из входного потока.

Определение: `int getc (stream)`
`FILE *stream;`

`getchar` — прочитать следующий символ из стандартного файла ввода.

Определение: `int getchar ()`

`ungetc` — вернуть символ во входной поток.

Определение: `int ungetc (c, stream)`
`int c;`
`FILE *stream;`

10.7. Вывод символа

`fputc` — записать символ в поток.

Определение: `int fputc (c, stream)`
`int c;`
`FILE *stream;`

З а м е ч а н и е. Функции `putc()` и `putchar()` являются макроопределениями.

`putc` — записать символ в поток.

Определение: `int putc (c, stream)`
`int c;`
`FILE *stream;`

`putchar` — записать символ в стандартный файл вывода.

Определение: `int putchar (c)`
`int c;`

10.8. Блочный ввод-вывод

`fread` — прочитать из входного потока определенное число байт (символов).

Определение: `int fread (ptr, size, nitems, stream)`
`char *ptr;`
`int size, nitems;`
`FILE *stream;`

`fwrite` — записать определенное число байт в выходной поток.

Определение: `int fwrite (ptr, size, nitems, stream)`
`char *ptr;`
`int size, nitems;`
`FILE *stream;`

11. ДРУГИЕ БИБЛИОТЕКИ

Ниже перечислены некоторые (не все) широко используемые библиотечные функции.

11.1. Выполнение команд языка shell

З а м е ч а н и е. Для выполнения описанной в этом подразделе функции необходимо включить в программу файл `stdio.h` командой

`#include <stdio.h>`

`system` — выполнить команду языка shell, описанную как строка `string`.

Определение: `int system (string)`
`char *string;`

11.2. Временные файлы

З а м е ч а н и е. Для выполнения описанных в этом подразделе функций необходимо включить в программу файл `stdio.h`.

`tmpnam` — создать временное имя файла.

Определение: `char *tmpnam (s)`
`char *s;`

`tempnam` — создать временное имя файла, используя каталог `dir` и префикс файла `prfx`.

Определение: `char *tempnam (dir, prfx)`
`char *dir, *prfx;`

mktemp — создать уникальное имя файла по шаблону, записанному в строке template.

Определение: char *mktemp (template)
char *template;

tmpfile — создать временный файл.

Определение: FILE *tmpfile ()

11.3. Обработка строк

З а м е ч а н и е. Для выполнения описанных в этом подразделе функций необходимо включить в программу файл string.h командой

```
#include <string.h>
```

strcat — сцепить две строки.

Определение: char *strcat (s1, s2)
char *s1, *s2;

strncat — сцепить две строки, причем из второй строки копировать не более n символов.

Определение: char *strncat (s1, s2, n)
char *s1, *s2;
int n;

strcmp — сравнить две строки в лексикографическом порядке.

Определение: int strcmp (s1, s2)
char *s1, *s2;

strncmp — сравнить первые n символов двух строк.

Определение: int strncmp (s1, s2, n)
char *s1, *s2;
int n;

strcpy — копировать строку s2 в строку s1.

Определение: char *strcpy (s1, s2)
char *s1, s2;

strncpy — копировать не более n символов строки s2.

Определение: char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

strlen — определить длину строки (число символов без завершающего нулевого символа).

Определение: int strlen (s)
char *s;

strchr — найти в строке первое вхождение символа c.

Определение: char *strchr (s, c)
char *s;
int c;

strrchr — найти в строке последнее вхождение символа c.

Определение: char *strrchr (s, c)
char *s;
int c;

strpbrk — найти в строке s1 любой из множества символов, входящих в строку s2.

Определение: char *strpbrk (s1, s2)
char *s1, *s2;

strspn — определить длину отрезка строки s1, содержащего символы из множества символов, входящих в строку s2.

Определение: int strspn (s1, s2)
char *s1, *s2;

strcspn — определить длину отрезка строки s1, содержащего символы, не входящие в множество символов строки s2.

Определение: int strcspn (s1, s2)
char *s1, *s2;

strtok — выделить из строки s1 лексемы, разделенные любым из множества символов, входящих в строку s2.

Определение: char *strtok (s1, s2)
char *s1, *s2;

11.4. Проверка символов

З а м е ч а н и е. Макроопределения, описанные в этом подразделе, определены в файле ctype.h, который должен быть включен в программы командой

```
#include <ctype.h>
```

Определение: int *имя_макроса* (c)
int c;

isalpha	Символ с – буква.
isupper	Символ с – прописная буква (в верхнем регистре).
islower	Символ с – строчная буква (в нижнем регистре).
isdigit	Символ с – цифра (0 – 9).
isxdigit	Символ с – шестнадцатеричная цифра (0 – 9), прописная (A – F) или строчная (a – f) буква.
isalnum	Символ с – буква или цифра.
isspace	Символ с – символ пробела, табуляции; перевода строки, новой строки, вертикальной табуляции или перевода формата.
ispunct	Символ с – символ пунктуации, т. е. не управляющий символ, не буква, не цифра и не пробел.
isprint	Символ с – печатный, т. е. имеющий значение (в коде ASCII) от 040 (пробел) до 0176 (тильда).
isgraph	Символ с – графический, т. е. печатный, за исключением пробела.
iscntrl	Символ с – управляющий (0 – 037) или символ удаления (0177).
isascii	Символ с – символ кода ASCII, т. е. его значение лежит в диапазоне от 0 до 0200.

11.5. Преобразование символов

З а м е ч а н и е. Макроопределения, описанные в этом разделе, определены в файле `ctype.h`, который должен быть включен в программу, использующую эти макросы, командой `#include <ctype.h>`

Определение: `int имя_макроса (c)`
`int c;`

<code>toascii</code>	– преобразование целого в символ кода ASCII.
<code>tolower</code>	– преобразование буквы в нижний регистр.
<code>toupper</code>	– преобразование буквы в верхний регистр.
<code>_tolower</code>	– такая же, как <code>tolower</code> , но более быстрая и ограниченная функция.
<code>_toupper</code>	– такая же, как <code>toupper</code> , но более быстрая и ограниченная функция.

11.6. Преобразование строки в число

`strtol` – преобразование строки в длинное целое число.

Определение: `long strtol (str, ptr, base)`
`char *str, **ptr;`
`int base;`

`atol` – преобразование строки в длинное целое число (специальный вариант функции `strtol()`).

Определение: `long atol (str)`
`char *str;`

`atoi` – преобразование строки в целое. Преобразование в тип `int` результата, возвращаемого функцией `atol()`.

Определение: `int atoi (str)`
`char *str;`

`atof` – преобразование строки в число двойной точности с плавающей точкой.

Определение: `double atof (str)`
`char *str;`

`strtod` – преобразование строки в число двойной точности с плавающей точкой.

Определение: `double strtod (str, ptr)`
`char *str, *ptr;`

11.7. Доступ к аргументам

`getenv` – ввести строку, связанную с переменной оболочки.

Определение: `char *getenv (name)`
`char *name;`

`getopt` – ввести следующий символ опций из списка аргументов.

Определение: `int getopt (argc, argv, optstring)`
`int argc;`
`char **argv, *optstring;`

Для записи текущих значений индекса и указателя аргументов используются следующие внешние переменные:

`extern char *optarg;`
`extern int opind, opterr;`

11.8. Распределение памяти

malloc — выделение памяти размером *size* байт.

Определение: `char *malloc (size)`
`unsigned size;`

calloc — выделение памяти и обнуление ее.

Определение: `char *calloc (nelem, elsize)`
`unsigned nelem, elsize;`

realloc — изменение размера ранее выделенной памяти.

Определение: `char *realloc (ptr, size)`
`char *ptr;`
`unsigned size;`

free — освобождение ранее выделенной памяти.

Определение: `void free (ptr)`
`char *ptr;`

12. ФОРМАТИРОВАННЫЙ ВЫВОД

Для описания функций форматированного вывода `printf`, `fprintf`, `sprintf` используются следующие метаобозначения:

- `%` Пробел (символ `%` на самом деле не печатается!).
- `{ }` Используется только один из перечисленных элементов.
- `[]` Используется только один или не используется ни одного из перечисленных элементов.

З а м е ч а н и е. Для использования функций `printf`, `fprintf`, `sprintf` в программу необходимо вставить команду препроцессора

```
#include <stdio.h>
```

Функции `printf`, `fprintf` и `sprintf` имеют переменное число аргументов. Число и типы аргументов должны соответствовать спецификации преобразования в форматной (управляющей) строке.

printf — записать аргументы в стандартный файл вывода `stdout` в соответствии с форматной строкой *format*.

Определение: `int printf (format [,arg]...)`
`char *format;`

fprintf — записать аргументы в поток *stream* в соответствии с форматной строкой *format*.

Определение: `int fprintf (stream, format [,arg]...)`
`FILE *stream;`
`char *format;`

sprintf — записать аргументы в массив символов *s* в соответствии с форматной строкой.

Определение: `int sprintf (s, format [,arg]...)`
`char *s, *format;`

П р и м е р

```
printf ("error no. %d: %s", err, msg);
```

Печатается значение переменной *err* как десятичное целое и значение *msg* как строка. Результат форматированного вывода будет выглядеть следующим образом (с точностью до значения переменных):

```
error no. 13: cannot access file
```

12.1. Спецификация преобразования

`%` $\left[\begin{array}{c} \text{выравнивание} \\ \text{ширина} \end{array} \right] \left[\begin{array}{c} * \\ \text{признаки} \end{array} \right] \left[\begin{array}{c} \text{дополнительные} \\ \text{преобразования} \end{array} \right] \text{символ}$

Выравнивание вправо: по умолчанию.

Выравнивание влево: символ `-`.

Ширина определяет минимальное число выводимых символов. Она может задаваться целым числом; если значение соответствующей переменной превышает явно заданную *ширину*, то выводится столько символов, сколько необходимо. Символ `*` обозначает, что число выводимых символов будет определяться текущим значением переменной.

П р и м е р

```
printf ("%*d", width, number);
```

12.2. Спецификация вывода символа

% [-] [ширина] c

Примеры

%c A
%3c b b A
%-3c A b b

12.3. Спецификация вывода строки

% [-] [ширина] [.точность] s

Точность определяет число печатаемых символов. Если строка длиннее, чем заданная точность, то остаток строки отбрасывается.

Пример

%10s abcdefghijklmn
%-10.5s abcde b b b b b
%10.5s b b b b b abcde

12.4. Спецификация вывода целого числа со знаком

% [-] $\left[\begin{smallmatrix} + \\ \emptyset \end{smallmatrix} \right] [ширина] [l] d$

Для отрицательных чисел автоматически выводится знак - (минус). Для положительных чисел знак + (плюс) выводится только в том случае, если задан признак +; если в спецификации задан пробел \emptyset , то в позиции знака выводится пробел.

Символы преобразования

- l — необходим для данных типа long¹;
- d — определяет вывод данных типа int в десятичном формате со знаком.

Примеры

%d 43
%+d +43
% d b43

¹ Символ l не является самостоятельным символом преобразования, он только модифицирует преобразование, выполняемое по символу d. — Прим. перев.

12.5. Спецификация вывода целого числа без знака

% [-] [#] [ширина] [l] $\left\{ \begin{smallmatrix} u \\ o \\ x \\ X \end{smallmatrix} \right\}$

Символ # определяет вывод начального нуля в восьмеричном формате или вывод начальных 0x или 0X в шестнадцатеричном формате. Символ l необходим для данных типа long.

Символы преобразования

- u — десятичное без знака;
- o — восьмеричное без знака;
- x — шестнадцатеричное без знака;
- X — шестнадцатеричное без знака с прописными буквами A — F.

Примеры (для 32-разрядных чисел)

%u 777626577
%o 5626321721
%#o 05626321721
%x 2e59a3d1
%#X 0X2E59A3D1

12.6. Спецификация вывода числа с плавающей точкой

% [-] $\left[\begin{smallmatrix} + \\ \emptyset \end{smallmatrix} \right] [#] [ширина] [.точность] \left\{ \begin{smallmatrix} f \\ e \\ E \\ g \\ G \end{smallmatrix} \right\}$

Для отрицательных чисел автоматически выводится знак - (минус). Для положительных чисел выводится знак + (плюс), если задан признак +; если в спецификации задан пробел \emptyset , то в позиции знака выводится пробел.

Завершающие нули не выводятся, если в спецификацию не включен признак #. Этот признак также обуславливает вывод десятичной точки даже при нулевой точности.

Точность определяет число цифр после десятичной точки для форматов f, e и E или число значащих цифр для форматов g и G.

Округление делается отбрасыванием. По умолчанию принимается точность в шесть десятичных цифр.

Символы преобразования и формат вывода по умолчанию

- f `[−] ddd.ddd` (число с фиксированной точкой).
- e `[−] d.ddddde{±}dd` (число в экспоненциальном формате).
- E `[−] d.ddddE{±}dd`
- g Наиболее короткий формат из f или e.
- G Наиболее короткий формат из f или E.

Типы аргументов float и double не различаются. Числа с плавающей точкой печатаются в десятичном формате.

Примеры

```
%f    1234.567890
%.1f  1234.6
%E    1.234568E+03
%.3e  1.235e+03
%g    1234.57
```

З а м е ч а н и е. Чтобы вывести символ %, необходимо в форматной строке задать два символа %%.
Пример

```
printf ("%5.2f%%", 99.44);
```

В результате выполнения данной функции будет напечатано 99.44%

13. ФОРМАТИРОВАННЫЙ ВВОД

Для описания функций форматированного ввода scanf, fscanf, sscanf используются следующие метаобозначения:

- ␣ Пробел (символ ␣ на самом деле не печатается!)
- { } Используется только один из перечисленных элементов.
- [] Используется только один или не используется ни одного из перечисленных элементов.

З а м е ч а н и е. Для использования функций, описанных в этом разделе, в программу необходимо включить команду препроцессора

```
#include <stdio.h>
```

Функции scanf, fscanf и sscanf могут иметь переменное число аргументов. Число и типы аргументов должны соответствовать спецификациям преобразования в форматной строке.

scanf – ввести данные из стандартного файла ввода stdin в соответствии с форматной строкой format, присваивая значения переменным, заданным указателями pointer.

Определение: `int scanf (format [,pointer]...)`
`char *format;`

fscanf – ввести данные из потока stream в соответствии с форматной строкой format.

Определение: `int fscanf (stream, format [,pointer]...)`
`FILE *stream;`
`char *format;`

sscanf – читать данные из строки s в соответствии с форматной строкой format.

Определение: `int sscanf (s, format [,pointer]...)`
`char *s, *format;`

Примеры

Входной поток содержит символы:

```
12.45 1048.73 AE405271 438
```

Вызов функции:

```
float x; char id [8+1]; int n;
scanf ("%f%f%8[A-Z0-9]%d", &x, id, &n);
```

Переменной x присваивается значение 12.45, символы 1048.73 пропускаются, переменной id присваивается строка символов "AE405271", переменной n – целое значение 438.

Входной поток содержит символы:

```
25 54.32E-01 monday
```

Вызов функции:

```
int i; float x; char name [50];
scanf ("%d%f%s", &i, &x, name);
```

Переменной i присваивается значение 25, переменной x – значение 5.432, переменной name – строка "monday".

Входной поток содержит:

```
56 789 0123 56ABC
```

Вызов функции:

```
int i; float x; char name [50];
scanf ("%2d%f*d,%[0-9]", &i, &x, name);
```

Переменной *i* присваивается значение 56, переменной *x* — значение 789.0, символы 0123 пропускаются, строка "56" присваивается переменной *name*. Последующий ввод символа из этого потока функцией *getchar* дает значение 'A'.

13.1. Спецификация преобразования

$\% [*] [\text{ширина}] \left[\begin{array}{l} \text{дополнительные} \\ \text{признаки} \end{array} \right] \begin{array}{l} \text{символ} \\ \text{преобразования} \end{array}$

Символ * обозначает пропуск при вводе поля, определенного данной спецификацией; вводимое значение не присваивается никакой переменной.

Ширина определяет максимальное число символов, вводимых по данной спецификации.

13.2. Пустые символы

Пробел или символ табуляции в форматной строке описывает один или более пустых символов. Пустые символы (пробел, символ табуляции, символы новой строки, перевода формата, вертикальной табуляции) во входном потоке в общем случае рассматриваются как разделители полей.

13.3. Литеральные символы

Литеральные символы в форматной строке, за исключением символов пробела, табуляции и символа %, требуют, чтобы во входном потоке появились точно такие же символы.

13.4. Спецификация ввода символа

$\% [*] [\text{ширина}] c$

Ширина определяет число символов, которые должны быть прочитаны из входного потока и присвоены массиву символов. Если *ширина* опущена, то вводится один символ. По данной спецификации можно вводить пустые символы.

13.5. Спецификация ввода строки

$\% [*] [\text{ширина}] s$

Ширина описывает максимальную длину вводимой строки. Строки во входном потоке должны разделяться пустыми символами; ведущие пустые символы игнорируются.

13.6. Спецификация ввода целого числа

$\% [*] [\text{ширина}] \left[\begin{array}{l} l \\ h \end{array} \right] \left\{ \begin{array}{l} d \\ u \\ o \\ x \end{array} \right\}$

Буква *l* определяет тип вводимых данных как *long*, буква *h* — как *short*. По умолчанию принимается тип *int*.

Символы преобразования

- d* — десятичное целое со знаком;
- u* — десятичное целое без знака;
- o* — восьмеричное целое без знака;
- x* — шестнадцатеричное целое без знака.

13.7. Спецификация ввода числа с плавающей точкой

$\% [*] [\text{ширина}] [l] \left\{ \begin{array}{l} f \\ e \\ g \end{array} \right\}$

Буква *l* определяет тип вводимых данных как *double*, по умолчанию принимается тип *float*. Символы преобразования *f*, *e*, *g* являются синонимами.

13.8. Спецификация ввода по образцу

$\% [*] [\text{ширина}] \text{образец}$

Образец (сканируемое множество) определяет множество символов, из которых может состоять вводимая строка, и задается строкой символов, заключенной в квадратные скобки.

Примеры

```
[abcd]
[A321]
```

Непрерывный (в коде ASCII) диапазон символов образца описывается первым и последним символами диапазона.

Примеры

```
[a - z]
[A - F0 - 9]
```

Если на первом месте в образце стоит символ `^`, то вводиться будут все символы из входного потока, кроме перечисленных в образце, т. е. допустимое по этой спецификации множество символов будет дополнительным к описанному.

Пример

```
[^0 - 9]
```

По спецификации преобразования, заданной образцом, вводится строка символов, включая завершающий ее нулевой символ. Ведущие пустые символы не пропускаются.

14. МОБИЛЬНОСТЬ ПРОГРАММ НА ЯЗЫКЕ СИ

Мобильность программ — это свойство, позволяющее выполнять программы на разных ЭВМ, работающих под управлением разных версий ОС UNIX, с минимальными изменениями.

Изложенные в этом разделе рекомендации не являются строгими правилами. Не существует методики, гарантирующей автоматическое получение мобильной программы. Если вы будете использовать эти рекомендации при разработке своих программ, то ваши программы будут более мобильны, а также лучше организованы, более легки для понимания, изменения и поддержания.

14.1. Верификатор lint

Верификатор (программа семантического контроля) `lint` обеспечивает строгую проверку типов и выявляет многие конструкции, ухудшающие мобильность программ, написанных на языке Си.

Если использовать верификатор `lint` на всех этапах разработки, то программный продукт будет гораздо легче переносить на любую версию ОС UNIX. Если вам потребуется переписать старую программу для повышения мобильности, то верификатор `lint` поможет вам выявить все сомнительные места.

Верификатор `lint` — это хорошее сервисное средство для разработки мобильных программ. Однако не надейтесь, что ваша программа является мобильной, если верификатор `lint` "не имеет к ней никаких претензий", и не полагайтесь на него, если программа написана плохо, так как верификатор может пропустить некоторые немобильные конструкции. Следуйте изложенным в этом разделе рекомендациям, улучшающим стиль программ.

14.2. Зависимость от компилятора

Некоторые детали в языке Си не стандартизированы. Это может проявиться в небольших различиях при обработке программы разными компиляторами. Какими бы "малыми" эти различия не были, они могут породить серьезные проблемы при переносе программ с одной вычислительной системы на другую. Например, в описании языка Си не определен порядок вычисления операндов большинства бинарных операций, таких, как сложение и умножение (см. с 31). Следовательно, не определен порядок появления возможных побочных эффектов. Поэтому не делайте никаких предположений о реализации свойств языка, которые строго не определены.

14.3. Зависимость от ЭВМ

Не полагайтесь на определенный размер машинного слова.

Размер данных типа `int` зависит от размера машинного слова, различающегося у разных ЭВМ. Если вы не уверены в результате операций над целыми числами, используйте тип `long`, чтобы избежать проблемы переполнения.

Не гарантируется, что размер данных типа `int` совпадает с размером машинного слова. В языке Си определяется только, что размер данных типа `short` меньше или равен размеру данных типа

int, который, в свою очередь, меньше или равен размеру данных типа long. Размер слова может сказаться на обработке двоичных масок.

Пример

```
#define MASK 0177770 /* неправильно */
int x;
x &= MASK;
```

В этом примере три правых бита целого x будут обнуляться только в том случае, если данные типа int занимают 16 бит. Но если размер данных типа int больше 16 бит, то кроме этого будут обнуляться левые биты данного x. Чтобы избежать таких проблем, используйте следующее макроопределение:

```
#define MASK (~07) /* правильно */
int x;
x &= MASK;
```

Этот пример корректен для всех ЭВМ независимо от размера данных типа int.

Тщательно проверяйте операции сдвига.

Максимальное число бит, которые могут быть сдвинуты вправо или влево, различно на разных ЭВМ. Если заданный в операции сдвиг превысит допустимый максимум, то результаты операции будут непредсказуемы.

Перед сдвигом преобразуйте целые значения к типу unsigned. На некоторых ЭВМ сдвиг выполняется логически, т. е. освобождающиеся разряды обнуляются. На других сдвиг производится арифметически, и освобождающиеся разряды заполняются значением знакового разряда. Однако в языке Си гарантируется, что значения типа unsigned сдвигаются логически.

Используйте поименованные константы.

Использование в программе числовых констант, особенно когда смысл их неочевиден, является плохим стилем программирования. Числовые константы лучше определять в программе символическими именами, связанными с числовыми константами командой препроцессора #define. Такие определения легко находить и модифицировать, если они размещены в некотором стандартном месте. Обычно это начало программы или файл заголовка.

Пример

```
#define SCREENWIDTH 80
```

Такое определение позволяет использовать поименованную константу SCREENWIDTH вместо числа 80.

Определяйте размер объекта операцией sizeof

Для определения размера некоторого объекта часто используют константы, что снижает мобильность программ. Использование операции sizeof позволяет решить эту проблему.

Пример

```
#define NUMELEM(ARRAY)\
(sizeof(ARRAY) / sizeof(*(ARRAY)))
```

Такое макроопределение обеспечивает мобильный способ определения числа элементов в массиве ARRAY.

Не используйте несколько символов в одной символьной константе.

Поскольку символьные константы представляются значениями типа int, определение языка Си позволяет в принципе задать символьную константу, состоящую из нескольких символов. Однако порядок размещения символов в машинном слове различен на разных ЭВМ.

Не полагайтесь на внутреннюю кодировку целых чисел.

Большинство ЭВМ представляет целые числа в дополнительном коде, но некоторые — в обратном коде. Поэтому не используйте возможности, которые предоставляет дополнительный код. Например, сдвиг на 1 бит влево отрицательного числа (чтобы уменьшить его значение в два раза) не приведет к желаемому результату на ЭВМ с обратным двоичным кодом.

Формат чисел с плавающей точкой различен на разных ЭВМ.

Представление чисел с плавающей точкой на разных ЭВМ различно. Поэтому точность результатов арифметических вычислений с плавающей точкой может быть разной на различных ЭВМ. Избегайте или в крайнем случае подробно документируйте все вычисления, зависящие от точности.

Не полагайтесь на определенный порядок и число байт в слове.

Число байт и порядок их размещения в машинном слове различны у разных ЭВМ.

Пример

Следующая функция определена неправильно — на выход будет записан нулевой символ, если какой-то байт в слове имеет меньший адрес, чем младший байт;

```
#define STDOUT 1
putchar(c);           /* неправильно */
int c;
{
    write(STDOUT, (char *) &c, 1);
}
```

В данном примере аргумент `c` должен описываться как имеющий тип `char`; в этом случае преобразование типа данных станет ненужным.

Не полагайтесь на определенное число бит в байте.

Поскольку число бит в байте у разных ЭВМ различно, то не предполагайте, что байт всегда занимает 8 бит. Чтобы определить число бит в байте, используйте поименованную константу, содержащуюся в стандартном файле:

```
/usr/include/values.h
```

З а м е ч а н и е. Все системные файлы-заголовки размещаются в каталоге `/usr/include`.

Будьте осторожны с символами, имеющими знак.

На некоторых ЭВМ символы представляются как целые значения со знаком, и при вычислении выражений данные типа `char` обрабатываются с учетом знака. Для повышения мобильности можно использовать явное описание типа `unsigned char` или преобразовывать символы перед обработкой к типу `unsigned char`. В других случаях надо использовать данные целого типа.

Пример

Если на данной ЭВМ допустимы символы со знаком, то существует опасность, что индексация символом некоторого массива приведет к выходу за его границы:

```
#define TABSIZE 256
char c;
extern char table[TABSIZE];
c = table[c];           /* неправильно */
```

Чтобы избежать этого, опишите переменную `c` как имеющую тип `unsigned char` или индексировать таблицу значением, преобразованным к типу `unsigned char`.

Пример

Следующий фрагмент программы ошибочен — конец файла никогда не будет обнаружен, если символы представляются как беззнаковые значения:

```
#include <stdio.h>
char c;           /* неправильно */
if ((c = getchar()) != EOF)
```

Если значение символа не может быть отрицательным, то переменная `c` никогда не станет равной поименованной константе `EOF`, которая равна `-1`. Библиотечная функция `getchar` (см. с. 64) возвращает значение типа `int`, поэтому `c` надо описать как переменную типа `int`.

Не комбинируйте разные поля бит.

Не используйте поля бит для представления данных на внешних носителях.

Поля бит можно сделать мобильными, если не объединять разные поля. Максимальный размер поля бит зависит от размера машинного слова, и поля бит не могут пересекать границу слова. Кроме того, порядок размещения полей в слове (слева направо или справа налево) зависит от типа ЭВМ.

Использование полей бит для описания размещения данных на внешних носителях делает программу немобильной. Для того чтобы упростить эту проблему, поместите определения полей бит в файл заголовка и укажите, что они зависят от типа ЭВМ.

Используйте операцию преобразования типа для указателей.

В общем случае операции преобразования типа указателя не являются мобильными. Однако можно преобразовать указатель в данное любого целого типа, имеющее достаточно большой размер, и при обратном преобразовании получить исходное значение указателя. Аналогично указатель на некоторый объект может быть преобразован в указатель на меньший объект и обратно.

Учитывайте выравнивание при изменении значения указателя.

Если вы преобразуете указатели из одного типа в другой, то

при выполнении программы может возникнуть ошибка адресации вследствие ограничений на выравнивание данных в машинной памяти. Используйте библиотечную функцию `malloc` (см. с. 70), возвращающую указатель на символ, выравненный в памяти в соответствии с требованиями данной ЭВМ, так что этот указатель может быть преобразован в указатель любого типа.

Следите за сравнением указателей, имеющих знак.

Некоторые ЭВМ выполняют сравнение указателей с учетом знака, другие делают беззнаковое сравнение. Это различие несущественно, если сравнивать указатели, содержащие правильные адреса. Если указателю будет присвоено значение -1 , то в зависимости от ЭВМ оно будет рассматриваться или как наибольшее допустимое значение, или как недопустимое значение (меньше минимально допустимого).

Единственная константа, которую можно "безопасно" присваивать указателю, — это ноль, преобразованный к типу соответствующего указателя.

Следите за переполнением значения указателей.

Арифметические преобразования указателей могут привести к переполнению или потере значимости. Такие циклические преобразования значений (от наибольшего к наименьшему или наоборот) могут возникнуть при адресации массива, расположенного в начале или в конце машинной памяти.

Пример

Этот фрагмент программы показывает возможность появления потери значимости:

```
struct large x [SIZE], *p;      /* неправильно */
for (p = x [SIZE-1]; p >= x; p--)
```

Если массив `x` расположен в начале памяти, то возможна ситуация, при которой `x - 1` будет не меньше, а больше `x` вследствие перехода через нижнюю границу диапазона значений указателей (потери значимости).

Не полагайтесь на конкретную кодировку символов.

Не используйте в программе предположения, что символы в кодовом наборе располагаются последовательно.

Пример

```
char c;
if (c >= 'a' && c <= 'z') /* неправильно */
```

Такая проверка символа `c` на принадлежность к строчным буквам не является мобильной. Чтобы такая проверка правильно выполнялась на других ЭВМ, сделайте так:

```
char c;
if (islower(c)) /* правильно */
```

Библиотечная функция `islower` (см. с. 67) определена в библиотеке стандартных функций ввода-вывода и является машинно-зависимой. Поскольку ее спецификация мобильна, то данная функция обеспечивает мобильную проверку символов.

Учтите, что разные символьные коды могут отличаться по числу входящих в них символов. Не используйте разность значений двух букв для вычисления лексикографического расстояния между ними.

Не используйте программные трюки, зависящие от аппаратуры.

Любое повышение эффективности выполнения программы, достигаемое за счет знания особенностей конкретной ЭВМ, обычно не оправдывает связанную с этим потерю мобильности.

14.4. Хорошо организованные программы

Программа называется хорошо организованной, если ее легко читать, модифицировать, эксплуатировать и, следовательно, переносить на другие ЭВМ.

Все определения, связанные с конкретной операционной средой и конкретной ЭВМ, помещайте в файл заголовка.

Важнейшим средством разработки мобильных программ являются команды препроцессора `#include` и `#define` (см. с. 52). Помещайте все определения типов данных, поименованных констант, макроопределения, используемые более чем одной программой, в единый файл заголовка так, чтобы все возможные изменения были локализованы.

Пример

```
#include <values.h>
```

С помощью этой команды в программу включается стандартный файл заголовка `/usr/include/values.h`, который содержит аппаратные константы.

Используйте файлы заголовка для общих определений одного проекта, т. е. множества связанных программ. Используйте локальные файлы заголовка для отдельных программ.

Используйте файл заголовка для локализации данных, зависящих от операционной среды, таких как имена файлов и режимы выполнения. Все, что может измениться при переходе на другую операционную систему или в рамках той же системы, помещайте в файлы заголовка, где эти данные легко могут быть модифицированы.

Не помещайте в файлы заголовка определения внешних переменных, которые управляют распределением памяти. Используйте файлы заголовка только для определения команд препроцессора и типов данных.

Для локализации программных фрагментов, зависящих от ЭВМ, используйте функции, условную компиляцию и команду `#define`.

Функции, зависящие от конкретной ЭВМ, объедините в отдельный исходный файл. Если таких файлов несколько, то соберите их в отдельном каталоге.

Фрагменты исходного кода, зависящие от аппаратуры, заключайте в команды условной компиляции¹ (см. с. 54).

Пример

Следующий фрагмент программы описывает стек, который может наращиваться в разных направлениях в зависимости от аппаратных особенностей ЭВМ.

```
int *stackptr;
#ifdef MACHINE1
    *--stackptr = datum; /* растет вниз */
#else
    *++stackptr = datum; /* растет вверх */
#endif
```

Для локализации характеристик конкретной ЭВМ можно использовать макроопределения (см. с. 53).

Пример

```
#define BITSPERBYTE 8
#define BITS(TYPE)\
    (sizeof(TYPE) * BITSPERBYTE)
```

Спецификация поименованной константы `BITSPERBYTE` мобильна, реализация — не мобильна. В макроопределении `BITS` мобильна как спецификация, так и реализация¹.

Проверяйте число и тип аргументов, передаваемых функциям.

Убедитесь, что число и тип аргументов, передаваемых функциям при вызове, согласуются с числом и типом формальных параметров этих функций. Даже если при конкретном вызове можно передавать меньше аргументов, чем определено, не опускайте ни одного из них; опишите пустые фактические аргументы, такие как нулевые значения.

В файле `/usr/include/varargs.h` описаны средства для мобильного определения функций с переменным числом аргументов. Например, библиотечная функция `printf` (см. с. 70) реализована с использованием этих средств.

Используйте стандартные библиотечные функции; не определяйте собственные системные вызовы, если без этого можно обойтись.

Стандартные библиотечные функции ОС UNIX обеспечивают большой выбор универсальных процедур. Библиотеки ОС UNIX содержат стандартные функции, обеспечивающие доступ к таким средствам операционной системы, как ввод и вывод. Библиотечные функции обеспечивают повышение мобильности, изолируют вашу программу от возможных изменений в операционной системе (см. также с. 60 — 77).

Тщательно определяйте внешние имена.

Для облегчения эксплуатации и увеличения мобильности программы определяйте все внешние переменные в отдельных исходных файлах. Не забудьте, что все остальные исходные файлы программы должны ссылаться на эти определения с помощью описа-

¹ Спецификацией здесь называется определяемая лексема, а реализацией — определяющее константное выражение. — Прим. перев.

ний extern (см. с. 49 и 51). При этом можно использовать разные методы. Можно поместить эти внешние описания в начало исходных файлов программы (с помощью включения файлов) или описать эти внешние переменные в функциях, которые их используют.

Максимальное число значимых символов в идентификаторах внешних переменных и функций зависит от операционной системы. Кроме того, некоторые операционные системы преобразуют все строчные буквы в прописные. Компоновщик (редактор связей) сообщает о конфликтах имен, но обнаруживает не все возможные ошибки, поэтому не возлагайте на эту программу обязанность разрешать противоречия в именах.

Используйте описание typedef для локализации определения типов данных, зависящих от ЭВМ.

Описание typedef (см. с. 47) обеспечивает локальные определения типов тех данных, которые зависят от конкретной ЭВМ. Если вы измените определение типа, заданное описанием typedef, то соответственно изменятся все переменные, описанные с помощью этого производного типа. Система обеспечивает набор стандартных определений в файле

/usr/include/sys/types.h

Пример

```
typedef unsigned short ino_t; /* индекс файла */
Этот пример показывает типичное использование определения
типа в файле /usr/include/sys/types.h.
```

14.5. Мобильность файлов данных

Для переноса файлов, содержащих двоичные данные, используйте символьный ввод-вывод.

Файлы двоичных данных по сути своей не мобильны, поскольку разные ЭВМ используют разное внутреннее представление данных. К сожалению, не существует простого пути для переноса файлов данных. Порядок байт в слове может привести к серьезным проблемам при переносе данных с одной ЭВМ на другую по принципу "байт в байт". Кроме того, коды символов могут быть разными на разных ЭВМ.

Один из способов решения этой проблемы заключается в разработке специальных программ преобразования для конкретных форматов данных. Другой подход заключается в записи байт, составляющих объект данных, в некотором машинно-независимом порядке. Для передачи символьных данных используйте библиотечные функции printf и scanf (см. с. 70 и 74), хотя это и непрактично.

ПРИЛОЖЕНИЕ. НАБОР СИМВОЛОВ КОДА ASCII

З а м е ч а н и е. Комбинация символов ^@, ^A и т. д. обозначает CTRL/@, CTRL/A и т. д. Набор символов может слегка отличаться на разных терминалах.

Управляющие символы

Код ¹				ASCII
Д	В	Ш		
0	000	0	^@ NUL (Пусто ПУС)	
1	001	01	^A SOH (Начало заголовка НЗ)	
2	002	02	^B STX (Начало текста НТ)	
3	003	03	^C ETX (Конец текста КТ)	
4	004	04	^D EOT (Конец передачи КП)	
5	005	05	^E ENQ (Кто там? КТМ)	
6	006	06	^F ACK (Подтверждение ДА)	
7	007	07	^G BEL (Звонок ЗВ)	
8	010	08	^H BS (Возврат на шаг ВШ)	
9	011	09	^I TAB НТ (Горизонтальная табуляция ГТ)	
10	012	0A	^J LF (Перевод строки ПС)	
11	013	0B	^K VT (Вертикальная табуляция ВТ)	
12	014	0C	^L FF (Перевод формата ПФ)	
13	015	0D	^M CR (Возврат каретки ВК)	

¹ Д — десятичный, В — восьмеричный, Ш — шестнадцатеричный. — Прим. перев.

Д	В	Ш	ASCII
14	016	0E	^N SO (Выход ВЫХ)
15	017	0F	^O SI (Вход ВХ)
16	020	10	^P DLE (Авторегистр1 AP1)
17	021	11	^Q DC1 (X-ON)
18	022	12	^R DC2
19	023	13	^S DC3 (X-OFF)
20	024	14	^T DC4
21	025	15	^U NAK (Отрицание НЕТ)
22	026	16	^V SYN (Синхронизация СИН)
23	027	17	^W ETB (Конец блока КБ)
24	030	18	^X CAN (Аннулирование АН)
25	031	19	^Y EM (Конец носителя КН)
26	032	1A	^Z SUB (Замена ЗМ)
27	033	1B	^[ESC (Авторегистр2 AP2)
28	034	1C	^\\ FS (Разделитель файлов РФ)
29	035	1D	^] GS (Разделитель групп РГ)
30	036	1E	^^ RS (Разделитель записей РЗ)
31	037	1F	^_ US (Разделитель элементов РЭ)

Пробел

Код

Д	В	Ш	ASCII
32	040	20	SPACEBAR SP (Пробел ПРБ)

Печатные символы

Код

Д	В	Ш	ASCII	Д	В	Ш	ASCII
33	041	21	!	36	044	24	\$
34	042	22	"	37	045	25	%
35	043	23	#	38	046	26	&

Д	В	Ш	ASCII	Д	В	Ш	ASCII
39	047	27	'	74	112	4A	J
40	050	28	(75	113	4B	K
41	051	29)	76	114	4C	L
42	052	2A	*	77	115	4D	M
43	053	2B	+	78	116	4E	N
44	054	2C	,	79	117	4F	O
45	055	2D	-	80	120	50	P
46	056	2E	.	81	121	51	Q
47	057	2F	/	82	122	52	R
48	060	30	0	83	123	53	S
49	061	31	1	84	124	54	T
50	062	32	2	85	125	55	U
51	063	33	3	86	126	56	V
52	064	34	4	87	127	57	W
53	065	35	5	88	130	58	X
54	066	36	6	89	131	59	Y
55	067	37	7	90	132	5A	Z
56	070	38	8	91	133	5B	[
57	071	39	9	92	134	5C	\
58	072	3A	:	93	135	5D]
59	073	3B	;	94	136	5E	^
60	074	3C	<	95	137	5F	_
61	075	3D	=	96	140	60	`
62	076	3E	>	97	141	61	a
63	077	3F	?	98	142	62	b
64	100	40	@	99	143	63	c
65	101	41	A	100	144	64	d
66	102	42	B	101	145	65	e
67	103	43	C	102	146	66	f
68	104	44	D	103	147	67	g
69	105	45	E	104	150	68	h
70	106	46	F	105	151	69	i
71	107	47	G	106	152	6A	j
72	110	48	H	107	153	6B	k
73	111	49	I	108	154	6C	l

Д	В	Ш	ASCII	Д	В	Ш	ASCII
109	155	6D	m	119	167	77	w
110	156	6E	n	120	170	78	x
111	157	6F	o	121	171	79	y
112	160	70	p	122	172	7A	z
113	161	71	q	123	173	7B	{
114	162	72	r	124	174	7C	
115	163	73	s	125	175	7D	}
116	164	74	t	126	176	7E	~
117	165	75	u	127	177	7F	DEL,
118	166	76	v				RUB
							(Забой, 3Б)

Список литературы

S. C. Johnson, "LINT, A C Program Checker", UNIX System Programmer's Manual, Volume 2, AT&T Bell Laboratories. Published by Holt, Rinehart, and Winston, New York, 1983, 1979. Pages 278-290.

S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System", The Bell System Technical Journal, July-August 1978, (Volume 57, No. 6, Part 2), pages 2021-2048. AT&T Bell Laboratories, Murray Hill, NJ 07974.

R. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978. (Перевод см. в [Д6].)

T. Plum, C Programming Guidelines, Plum-Hall, Cardiff, NJ, 1984.

Дополнительный список литературы

Д1. Банахан М., Раттер Э. Введение в операционную систему UNIX: Пер. с англ. — М.: Радио и связь, 1985. — 344 с.

Д2. Баурн С. Операционная система UNIX: Пер. с англ. — М.: Мир, 1986. — 463 с.

Д3. Готье Р. Руководство по операционной системе UNIX: Пер. с англ. — М.: Финансы и статистика, 1985. — 232 с.

Д4. Иванов А. Г. Язык программирования Си: Предварительное описание// Прикладная информатика. — 1985. — Вып. 1. — С. 68 — 113.

Д5. Инструментальная мобильная операционная система ИНМОС/ М. И. Беляков, А. Ю. Ливеровский, В. П. Семик, В. И. Шяудкулис. — М.: Финансы и статистика, 1985. — 231 с.

Д6. Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку Си: Пер. с англ. — М.: Финансы и статистика, 1985. — 279 с.

Д7. Кристиан К. Введение в операционную систему UNIX: Пер. с англ. — М.: Финансы и статистика, 1985. — 318 с.

Д8. Томас Р., Йейтс Дж. Операционная система UNIX. Руководство для пользователей: Пер. с англ. — М.: Радио и связь, 1986. — 352 с.

Д9. Хэнкок Л., Кригер М. Введение в программирование на языке Си: Пер. с англ. — М.: Радио и связь, 1986. — 192 с.

Предметный указатель

Аргументы см. Параметры	Включение файлов 53
Библиотеки стандартные 65	Вывод форматированный 70
Блок 32	строка форматная 70
определение локальных переменных 47	функции стандартные 70
Ввод форматированный 74	Выражения 17
строка форматная 75	константные 36
функции стандартные 74	преобразование операндов 31
Ввод-вывод: 60	Доступ:
блочный 64	к каналам 62
доступ к каналам 62	— параметрам среды 42
— — файлам 61	— файлам 61
символьный 63	Зарезервированные слова 11
строковый 63	Идентификаторы 10, 87
Верификатор программ lint 78	

Инициализация:

переменных 49
цикла 38

Класс памяти 11

auto 47
extern 51
register 48
static 49

Код символьный ASCII 14,
84, 89

Комментарии 10

Компиляция условная 54

Константы:

длинные целые 13
перечислимые 15
поименованные 52, 80
с плавающей точкой 13
символьные 14
строковые 14
целые 12

Макроопределения 53

мобильные 86
стандартные 61

Макросы 53

Массивы:

инициализация 50
операции 27
описание 43

Метка:

варианта 36
оператора 32

Мобильность:

программ 78
файлов данных 88

Обработка строк 66

Объединения 45

выравнивание в памяти 46
операции 27
описание 46

Операнды:

метаобозначения 18
порядок обработки 31
преобразования в выраже-
ниях 31

Оператор:

вложенность 32
вызова функции 33
присваивания 33
пустой 33
составной 32
break 33
continue 34
do-while 38
for 38
goto 34
if-else 34
return 34
switch 36
while 37

Оператор-выражение 33

Операции:

адресные 26
— обращения по адре-
су * 26
— получение адреса & 26
арифметические 18
— вычитания — 19
— деления / 19
— получения остатка от
деления % 19
— сложения + 18
— увеличения ++ 19
— — побочные эффекты 19
— уменьшения -- 20
— — побочные эффекты 19
— умножения * 19
логические 24
над массивами [] 27

— объединениями и струк-
турами . -> 27
отношения == != <
<= > >= 23

присваивания = 21

смешанные

— вызова функции () 28
— запятая , 28
— определения размера
sizeof 28
— преобразование типа
(тип) 28
— условная ?: 28

Операционная система UNIX 9,
78

Описание 42

внешних объектов (перемен-
ных и функций) 51, 87
массивов и указателей 43
область действия 48
объединений 45
параметров 48
перечислений 46
полей бит 45
структур 44

Определение:

переменных
— глобальных 49
— локальных 47
типов данных 42, 88
функций 39

Параметры:

макроса 53
функции 40, 87
— main 42

Переменные:

автоматические 47
внешние 49, 51, 86
временные 47

глобальные 49
локальные 47
постоянные 47
регистровые 48
статические 49

Перечисления 46

Поля бит 45, 83

Порядок байт в слове 15, 81

Поток ввода-вывода 61

Преобразования:

арифметические в выраже-
ниях 31
символов 68
строк 69

Препроцессор:

включение файлов 53
замена идентификаторов 52
макросы 53
условная компиляция 54

Приоритеты и порядок выпол-
нения операций 29

Проверка:

символов 67
состояния файла 62

Размер:

данных основных типов 15
машинного слова 79

Символы:

значения 43, 82
набор кода ASCII 89

Синтаксис общий 10

Спецификация преобразова-
ния:

ввода
— метаобозначения 76
— по образцу % [] 77
— разделители полей 76
— символа %s 76
— строки %s 77

- целого числа %d %o %x %u 77
- числа с плавающей точкой %f %e %g 77
- вывода
- метаобозначения 71
- символа %c 72
- строки %s 72
- целого числа %d %o %x %u 72
- числа с плавающей точкой %f %e %g 73

Строка форматная:

- ввода 74
- вывода 70

Строки 14

Структура программы 55

Структуры:

- инициализация 50
- операции 27
- описание 44

Тип данных:

- массив 43
- объединение union 45
- основной 12, 43
- перечисление enum 46
- структура struct 44
- указатель 43

Типы данных:

- определение 42
- переименование 47

Указатели:

- операции 18, 26, 83
- описание 43

Файл:

- временный 65
- заголовка 80, 85
- стандартный 82, 86
- — ctypes.h 67
- — stdio.h 59, 60, 65
- — string.h 66
- стандартный ввода stdin 55, 60
- вывода stdout 55, 60
- диагностики stderr 61

Формат:

- операторов 32
- программы 10

Функции:

- библиотечные (стандартные) 60, 65, 87
- ввода форматированного 74
- ввода-вывода блочного 64
- — символьного 63
- — строкового 63
- вывода форматированного 70
- выполнение команд языка shell 65
- доступа к каналам 62
- — — параметрам среды 69
- — — файлам 61
- обработки строк 66
- преобразования символов 68
- — строк в число 69
- проверки символов 67
- — состояния файлов 62
- распределения памяти 70

Функция main 42

Указатель операций

- | | | |
|------------------|--|---------------|
| () | Вызов функции (подразд. 4.11) | Слева направо |
| [] | Выделение элемента массива (4.9) | |
| . | Выделение элемента структуры или объединения (4.10) | |
| -> | Выделение элемента структуры (объединения), адресуемой (ого) указателем (4.10) | |
| ! | Логическое отрицание (4.6) | Справа налево |
| ~ | Побитовое отрицание (4.7) | |
| - | Изменение знака (4.3) | |
| ++ | Увеличение на единицу (4.3) | |
| -- | Уменьшение на единицу (4.3) | |
| & | Определение адреса (4.8) | |
| * | Обращение по адресу (4.8) | |
| (тип) | Преобразование типа (4.11) | |
| sizeof | Определение размера в байтах (4.11) | |
| * | Умножение (4.3) | Слева направо |
| / | Деление (4.3) | |
| % | Деление по модулю (4.3) | |
| + | Сложение (4.3) | Слева направо |
| - | Вычитание (4.3) | |
| << | Сдвиг влево (4.7) | Слева направо |
| >> | Сдвиг вправо (4.7) | |
| < | Меньше, чем (4.5) | Слева направо |
| <= | Меньше или равно (4.5) | |
| > | Больше, чем (4.5) | |
| >= | Больше или равно (4.5) | |
| == | Равно (4.5) | Слева направо |
| != | Не равно (4.5) | |
| & | Побитовая операция И (4.7) | Слева направо |
| ^ | Побитовая операция исключающее ИЛИ (4.7) | Слева направо |
| | Побитовая операция ИЛИ (4.7) | Слева направо |
| && | Логическая операция И (4.6) | Слева направо |
| | Логическая операция ИЛИ (4.6) | Слева направо |
| ?: | Условная операция (4.11) | Справа налево |
| = | Присваивание (4.4) | Справа налево |
| *= /= %= += -= | | |
| <<= >>= &= ^= = | | |
| , | Операция запятая (4.11) | Слева направо |