Министерство образования и науки Российской Федерации
Озерский технологический институт
(филиал)
Московского инженерно-физического института
(технического университета)

Кафедра прикладной математики

Язык Си

Описание

Озерск, 2005

Язык Си. Описание.
Подготовил Вл. Пономарев.
Озерск: ОТИ МИФИ, 2005. — 62 с.

В пособии содержится описание языка Си для его изучения студентами дневного отделения специальности 230105 «Программное обеспечение ЭВМ». Изложение сопровождается многочисленными примерами кода.

Содержание

Исторические сведения

Язык программирования *Си* был изобретен в 1972 году сотрудником фирмы AT&T *Bell Laboratories* Деннисом Ритчи во время совместной работы с Кеном Томпсоном над операционной системой *UNIX*. Прообразом языка послужил язык *Би* (согласно другим источникам — *BCPL*), разработанный Томпсоном, который, в свою очередь, создан на основе языка *Эй* (в оригинале названиями языков являются первые три буквы латинского алфавита — A, B, C).

Популярность языка обусловлена двумя основными причинами. Во-первых, язык *Си* очень гибок: его можно использовать в различных сферах практических приложений. Во-вторых, большая часть широко распространенной операционной системы *UNIX* написана на языке *Си*[1], который является основным языком программирования в этой системе.

Язык *Си* был разработан как инструмент для программистов-практиков, а главной целью его автора было создание удобного и полезного языка. Несмотря на то, что критерий полезности принимался во внимание при разработке других языков программирования, чаще всего при этом учитывались и другие потребности. Например, главная цель создания языка Паскаль — построение прочных основ обучения принципам программирования. Язык Бейсик создавался так, чтобы его синтаксис был близок к синтаксису английского языка (чтобы его мог применять любой пользователь компьютера).

Однако в повседневной деятельности программиста очень часто встречаются ситуации, когда используемый для написания программы язык не обладает необходимыми для реализации требуемого алгоритма средствами. Получение от программы свойств, связанных, например, с устройствами, требуют от языка возможностей для управления всеми средствами вычислительной машины. И при этом хотелось бы, чтобы язык был простым и мощным, то есть позволял бы просто, лаконично и красиво решить конкретную программистскую проблему.

Свойства языка Си

Язык *Си* является современным языком программирования. Он включает в себя все управляющие конструкции, рекомендуемые практическим и теоретическим программированием. Его структура побуждает программиста использовать в своей работе нисходящее проектирование, структурное программирование и пошаговую разработку модулей. Результатом такого подхода является надежная и читаемая программа.

Язык *Си* является эффективным языком программирования. Его структура позволяет использовать возможности вычислительной машины наилучшим образом. Программа на языке *Си* отличается компактностью в сочетании с выразительностью, лаконичностью и быстротой исполнения.

Язык *Си* — переносимый (мобильный) язык. Это означает, что программа на *Си*, написанная для одной вычислительной системы, может быть легко перенесена на другую вычислительную систему, при этом потребуются лишь незначительные изменения, или не потребуются вовсе.

Язык *Си* — мощный и гибкий. Его с одинаковым успехом можно использовать для решения вычислительной задачи и для мультипликации.

Язык *Си* — удобный язык программирования. Он достаточно структурирован, чтобы поддерживать хороший стиль программирования, и в то же время не связывать программиста всевозможными ограничениями.

---

[1] На языке *Си* написана также большая часть кода операционной системы Windows.

Основные понятия

Язык состоит из таких элементов, как имена, числа, литералы и символы, при помощи которых конструируется программа. Эти элементы называют *лексемами*. Они представляют собой слова, образованные при помощи знаков алфавита. Лексемы отделяются друг от друга *пробельными символами* и другими лексемами, такими, как *операторы* и *пунктуаторы*. Компилятор выделяет лексемы, отбрасывая (пропуская) пробельные символы.

Лексема (*token*) — неделимая смысловая часть исходного текста. Лексема (за исключением строкового литерала и знаковой константы) не может содержать в себе пробельные символы.

Лексема:

Ключевое слово (*keyword*) — ключевые слова имеют специальное назначение — с их помощью получают алгоритмические конструкции (операторы управления), такие, как выбор или цикл, они используются для управления ходом вычислений, связи данных, обозначения основных типов.

Идентификатор (*identifier*) — имя для обозначения функций, переменных, типов и меток операторов.

Константа (*constant*) — в оригинальной литературе словом *constant* обозначают непосредственные числовые значения. В отечественной практике такие константы принято называть *литералами*. Существуют еще знаковые и строковые литералы. Константы в этом смысле — часть литералов, относящаяся к числовым значениям, например, 123.

Строковый литерал (*string-literal*) — непосредственное значение-строка, заканчивающаяся *нулевым знаком* (*null-terminated string*).

Оператор (*operator*) — *символ* из одного или нескольких знаков, предписывающий действие с данными (*операндами*). В отечественной литературе оператор обычно называют *операцией*, а под оператором подразумевают *оператор управления* (*statement*).

Знак пунктуации (*punctuator*)

Это один из символов: `[]` `()` `{}` `*` `,` `:` `=` `;` `...` `#`

Они имеют специальное назначение, как и ключевые слова.

При помощи лексем программист определяет *компоненты программы*.

Компонент программы:

Функция (*function*) — именованный фрагмент исходного текста, описывающий часть или весь алгоритм программы. Функции содержат *исполняемый код*, определяемый операторами управления. Функция — элемент абстракции управления, позволяющий описывать алгоритм менее детально (и более понятно).

Переменная (*variable*) — идентификатор, указывающий на область памяти, значение которой может изменяться по ходу выполнения программы.

Константная переменная (*constant variable*) — идентификатор, указывающий на область памяти, значение которой не может быть изменено во время исполнения программы. В отечественной практике константные переменные принято называть *константами*.

Описание типа (*type definition*) — идентификатор, описывающий тип.

Метка оператора (*label*) — идентификатор, указывающий на конкретную точку в потоке управления функции (*точку управления*).

Алфавит

Алфавит определяет знаки, используемые для написания программы. В качестве алфавита в *Си* используется набор знаков кода ASCII-7. Знаки *пробел*, *табуляция* HT, *перевод строки* LF, *возврат каретки* CR, *перевод формата* FF, *вертикальная табуляция* VT и *новая строка* (*newline*), а также *комментарии* интерпретируются как *интервал* (*white space* — пробел), называются *пробельными* и служат в качестве разделителей. Набор знаков ASCII-7 приведен в таблице.
Для каждого знака указано его числовое значение в системах счисления по основанию 10 (*Dec*) и 16 (*Hex*):

Dec	Hex	Знак	Символ	Dec	Hex	Знак	Dec	Hex	Знак	Dec	Hex	Знак
0	00		NUL	32	20	пробел	64	40	@	96	60	`
1	01	☺	SOH	33	21	!	65	41	A	97	61	a
2	02	☻	STX	34	22	"	66	42	B	98	62	b
3	03	♥	ETX	35	23	#	67	43	C	99	63	c
4	04	♦	EOT	36	24	$	68	44	D	100	64	d
5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
6	06	♠	ACK	38	26	&	70	46	F	102	66	f
7	07	▪	BEL	39	27	'	71	47	G	103	67	g
8	08	◘	BS	40	28	(	72	48	H	104	68	h
9	09	○	HT	41	29	)	73	49	I	105	69	i
10	0A	◙	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	♂	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	♪	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	♫	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	☼	SI	47	2F	/	79	4F	O	111	6F	o
16	10	►	DLE	48	30	0	80	50	P	112	70	p
17	11	◄	DC1	49	31	1	81	51	Q	113	71	q
18	12	‼	DC2	50	32	2	82	52	R	114	72	r
19	13	↕	DC3	51	33	3	83	53	S	115	73	s
20	14	¶	DC4	52	34	4	84	54	T	116	74	t
21	15	§	NAK	53	35	5	85	55	U	117	75	u
22	16	▬	SYN	54	36	6	86	56	V	118	76	v
23	17	↨	ETB	55	37	7	87	57	W	119	77	w
24	18	↑	CAN	56	38	8	88	58	X	120	78	x
25	19	↓	EM	57	39	9	89	59	Y	121	79	y
26	1A	→	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	∟	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	▼	US	63	3F	?	95	5F	_	127	7F	DEL

Первые 32 знака управляют передачей и отображением текста и называются *управляющими*. Наиболее важные:

LF — **Line Feed**, переводит курсор на следующую строку;

FF — **Form feed**, переводит курсор в начало следующей страницы;

CR — **Carriage Return**, переводит курсор в начало текущей строки;

Различают *набор знаков для исполнения* (*execution character set*) и *набор знаков для компиляции* (*source character set*). Первый используется для исполнения программы и представляет все множество знаков (см. приложения).

Второй набор ограничивает знаки, которые можно использовать для написания текста программы, подмножеством основного набора знаков ASCII-7.

Триграфы

Набор знаков для компиляции является подмножеством кодового набора ASCII-7, а сам набор ASCII-7 является надмножеством неизменяемого набора знаков ISO 646-1983. *Триграф* — последовательность из трех знаков, входящих в неизменяемый набор ISO, при помощи которого можно записать один из знаков набора ASCII-7, отсутствующий в неизменяемом наборе. Триграф начинается с двух вопросительных знаков. Есть 9 триграфов:

Триграф	*Обозначаемый знак*
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

Во время первой фазы компиляции знаки триграфа заменяются на один обозначаемый знак. Для исключения неправильной интерпретации последовательности вопросительных знаков следует использовать *escape*-последовательность \?. Для того, чтобы вывести на экран надпись «What??!», нельзя использовать оператор

```
printf("\nWhat??!");
```

так как будет выведено «What|». Здесь последовательность знаков ??! является триграфом и заменяется знаком |. Правильно использовать две *escape*-последовательности \?:

```
printf("\nWhat\?\?!");
```

Многобайтные и широкие символы

Многобайтные (*multibyte character*) символы занимают в памяти два или три байта на каждый отдельный знак из расширенного набора символов, такого, как *Kanji* (символы японского языка) или *Hangul* (символы корейского языка), отличающихся большим количеством знаков (несколько тысяч).

Широкими (*wide character*) называются многонациональные символы расширенного набора, такие, как *À*, имеющие двухбайтное представление. Символы расширенного набора используются многими европейскими языками.

Для описания знаковых констант в *Си* используется тип char, имеющий однобайтное представление. При помощи этого типа можно описать один знак из стандартной кодовой таблицы ASCII-7. Проблема заключается в том, что символы расширенного набора в байтовых кодовых страницах имеют коды со значениями выше 127. Поскольку char — знаковый тип, все знаки расширенного набора автоматически становятся отрицательными.

Для описания широких символов следует использовать тип wchar_t, определенный в заголовочном файле ctype.h. Использование широких символов немного усложняет программу. Так, чтобы определить строку «Hello» широкими символами, нужно использовать префикс L, например:

```
wchar_t s[] = { L'H', L'e', L'l', L'l', L'o', 0 };
```

Элементы массива при этом имеют тип integer.

Широкие символы описываются кодовым набором *Unicode*. Использование широких и многобайтных символов возможно только в среде *Win32*.

Идентификаторы

Программа строится при помощи *компонентов программы* — функций, переменных, констант и типов (а также синтаксических конструкций).

Идентификатор (*identifier*) — имя компонента программы, задаваемое программистом. Каждый компонент должен иметь уникальное имя для того, чтобы компилятор мог отличать один компонент программы от другого.

Идентификатор строится по правилам:

- Первый знак: ***не_цифра***;
- Последующие знаки: ***не_цифра*** или ***цифра***;

  не_цифра — один из `a .. z, A .. Z, _`
  цифра — один из `0 1 2 3 4 5 6 7 8 9`
- Буквы строчного и прописного регистров считаются *различными*;
- Длина идентификатора — максимум 6 знаков для *внешнего* и 31 знак для *внутреннего* (правила ANSI).

Внешний идентификатор служит для связи программных модулей и объявляется при помощи ключевого слова extern. Под *внутренним* здесь понимается идентификатор, действие которого распространяется только на данный программный модуль.

Компиляторы не ограничивают длину идентификаторов, но следует точно знать *распознаваемую* длину, иначе два идентификатора, таких, как direction_left и direction_right некоторые компиляторы воспримут как идентичные из-за совпадения первых восьми знаков. Для компилятора *Borland C++ 3.1* распознаваемая длина идентификатора равна 32, для компилятора *Microsoft C* — 247 знаков. Правила ANSI более жесткие, так как они призваны обеспечить переносимость программ на языке *Си* на различные платформы. Соблюдение этих правил гарантирует успешную компиляцию и связывание.

Для именования компонентов программы следует придерживаться некоторых [постоянных] правил. Их соблюдение позволяет легко ориентироваться в программах как разработчику, так и другому специалисту.

Локальные, определенные внутри функции, идентификаторы лучше обозначать одной-двумя буквами, например:

`a`, `b`, `p`, `q`, `r`, `x`, `y`, `z` — вещественные;
`i`, `j`, `k`, `l`, `m`, `n` — целочисленные;
`s`, `t`, `u`, `v`, `w` — строковые;
`c`, `d`, `e` — знаковые;
`f`, `g`, `h` — логические.

Для параметров циклов for предпочтительны имена i, j, k. Не следует использовать идентификатор-букву o или O, чтобы не спутать её с нулём. Не следует начинать идентификатор с одного или двух знаков подчеркивания "_" , — компиляторы и компоновщики используют подобные имена для своих целей.

Глобальные (определенные вне функций) идентификаторы переменных и идентификаторы типов и меток операторов должны быть полными словами английского языка или сочетаниями слов, возможно соединенных знаком подчеркивания, например: names_counter, namescounter, NamesCounter, namesCounter.

Принцип здесь простой — объявление локального идентификатора находится недалеко от места его использования — тип идентификатора легко найти. Описание глобального идентификатора находится, как правило, далеко. При программировании в системе *Windows* действует также соглашение (называемое венгерской нотацией), по которому идентификатор начинается с префикса, указывающего на тип, например, m_bRequireSave определяет идентификатор логического типа (b) на уровне модуля (m).

Комментарии

Комментарии в языке *Си* заключаются в символы, состоящие из пар знаков /* и */. Такой комментарий может быть *многострочным*. Вложенные комментарии не допускаются. Следующий комментарий будет правильным:

```
/* текст /* текст */,
```

а приведенный ниже — нет:

```
/* текст /* текст */ */.
```

Комментарий в стиле *Си* компилируется как *пробел*. Например:

```
int/* параметр 1 */s        /* компилируется в int s */
int/* */s                   /* компилируется в int s */
int/**/s                    /* компилируется в int s */
```

Компиляторы допускают комментарии в стиле *ANSI*, которые начинаются с двух подряд знаков *slash*: //. Эти комментарии не могут быть *многострочными* и продолжаются до конца текущей строки текста. Следовательно, комментарий в стиле *ANSI* нельзя вставить, например, в описание функции. Пример комментария в стиле *ANSI*:

```
int func(int i) { return i; } // комментарий в стиле ANSI
```

Ключевые слова

Некоторые идентификаторы недопустимы потому, что они уже используются компилятором или компоновщиком. К ним относятся прежде всего *ключевые слова*, предназначенные для построения конструкций языка. Ключевые слова должны использоваться в программе только в соответствии с их назначением. В редакторе интегрированной среды разработки эти слова выделяются цветом. Список ключевых слов компилятора *Microsoft C*:

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

Реализации языка могут использовать дополнительные *зарезервированные идентификаторы*. Например, компилятор *Microsoft C* использует зарезервированные слова:

asm	based	cdecl	__declspec	dllexport
dllimport	except	fastcall	__finally	inline
int8	int16	int32	__int64	leave
naked	stdcall	thread	__try	

Компиляторы используют также некоторые дополнительные идентификаторы (такие, как main, far, near, pragma), которые не включаются в число зарезервированных слов. Кроме того, некоторые идентификаторы используются для именования функций и переменных в стандартных библиотеках. Программа, использующая библиотеку, не должна переопределять её имена. Так, использование имени clrscr для функции, определенной программистом, исключает возможность использования функции очистки экрана, определенной в стандартной библиотеке conio.

Целочисленные литералы

Литерал — это непосредственное значение, его явная запись. Литералы поставляют *значение* без использования *памяти*. Их используют в выражениях, для присвоения значений переменным, в качестве параметров функций и в качестве константных выражений в операторе выбора switch.

Целочисленный литерал (*integer constant*) — это *десятичный*, *восьмеричный* или *шестнадцатеричный* литерал, задающий целое значение со знаком или без знака.

Десятичный литерал (*decimal constant*):

Первый знак: один из `1 2 3 4 5 7 8 9`

Следующие знаки: один из `0 1 2 3 4 5 7 8 9`

Например:

```
char decc = 123;
int  deci = 12345;
long decl = 123456;
```

Восьмеричный литерал (*octal constant*):

Первый знак: `0`

Следующие знаки: один из `1 2 3 4 5 7`

Например:

```
char octc = 0173;
int  octi = 030071;
long octl = 0361100;
```

Шестнадцатеричный литерал (*hexadecimal constant*):

Первые два знака: `0x` или `0X`

Следующие знаки: один из `0 1 2 3 4 5 7 8 9 a b c d e f A B C D E F`

Значения буквенных цифр:

`a` = `A` = 10; `b` = `B` = 11; `c` = `C` = 12; `d` = `D` = 13; `e` = `E` = 14; `f` = `F` = 15

Например:

```
char hexc = 0x7B;
int  hexi = 0x3039;
long hexl = 0x1e240;
```

Литералу, описывающему отрицательное значение, предшествует знак минус, который интерпретируется как *унарная операция*. Знак плюс для литерала, описывающего положительное значение, необязателен. Например:

```
int neg = -63;
int pos = +63;
```

Тип целочисленного литерала определяется по его *значению* и по наличию или отсутствию *суффикса*. Если значение превышает диапазон типа int, тип литерала long. Если значение превышает диапазон типа long, тип литерала unsigned long. Диапазоны значений целых типов приведены ниже. Суффикс *l* или L явно задает тип литерала long. Суффикс u или U явно задает беззнаковый тип литерала (unsigned). Примеры:

```
10              /* тип int                                */
10L             /* тип long                               */
10UL или 10LU   /* тип unsigned long                      */
32541           /* тип int                                */
32768U          /* тип unsigned int                       */
65536           /* тип long                               */
65536U          /* тип unsigned long                      */
4294967297      /* тип unsigned long, значение равно 1 */
```

Тип литерала имеет значение при вычислении выражений. В следующем примере переменная a принимает значение 0, а не 65536, как программист может ожидать (компилятор *Borland C++*):

```
long a = 256 * 256;         /* a == 0 */
```

При вычислении выражения используется максимальный тип значения, участвующего в выражении, который определяется как int по значению литералов. Для вычисления значения типа long хотя бы один из литералов должен иметь суффикс *l* или L:

```
long a = 256 * 256L;        /* a == 65536 */
```

Если значение литерала превышает диапазон значений типа переменной, происходит преобразование, при котором значение изменяется — из него вычитается максимальное беззнаковое значение типа:

```
int i = 65535;  /* i == -1 == 65535 - 65536 */
int i = 65537;  /* i ==  1 == 65537 - 65536 */
```

Предельные значения знаковых целых типов:

Тип	*char*	*int* (Borland)	*int* (Microsoft)	*long*
Минимальное значение	−128	−32 168	−2 147 483 648	−2 147 483 648
Максимальное значение	127	32 767	2 147 483 647	2 147 483 647

Вещественные литералы

Вещественный литерал (*floating-point constant*) задает значение действительного числа со знаком. Вещественный литерал записывается с явным указанием десятичной точки (не запятой) и состоит из целой (*integer portion*) и дробной (*fractional portion*) частей, разделенных (соединенных) точкой (*decimal point*). Литерал не должен содержать пробельных символов. Допускается не указывать целую или дробную часть числа. Примеры:

```
1.0     /* действительное число  1        */
.1      /* действительное число  0,1      */
-1.     /* действительное число -1        */
1       /* неправильно, это целое число 1 */
```

Для записи очень больших или очень маленьких действительных чисел удобнее использовать экспоненциальную (научную) форму записи литерала. Экспоненциальная форма включает в себя латинскую букву e или E с последующим числом со знаком — степенью экспоненты. Экспоненциальная форма допускает не указывать точку:

```
1.0E01  /* действительное число  10           */
1E1     /* то же                              */
.1e1    /* действительное число  1            */
-1E11   /* действительное число -100000000000 */
-.1E-1  /* действительное число -0,01         */
```

Если не указан никакой суффикс, вещественный литерал имеет двойную точность, т.е. задает тип double. Суффикс f или F задает тип float, суффикс *l* или L задает тип long double.

Тип литерала влияет на точность (число верных значащих цифр) и диапазон допустимых значений. Диапазоны значений разных типов с плавающей точкой и точность приведены в следующей таблице.

Тип	*float*	*double*	*long double*
Минимальное положительное значение	1.1e−38	2.2e−308	3.3e−4932
Максимальное положительное значение	3.4e+38	1.7e+308	1.1e+4932
Число верных десятичных значащих цифр	6	15	19

Для компилятора *Microsoft C* тип long double совпадает с long.

Знаковые литералы

Литералы типа char состоят из одного знака алфавита, заключенного в одиночные кавычки, например 'a'. Знаковый литерал не может быть *пустым*. Его можно использовать как значение типа int, равное коду знака в таблице ASCII. Некоторые знаки, не имеющие графического представления, а также знаки, используемые в качестве ограничителей, обозначаются при помощи *escape*-последовательности, начинающейся со знака \:

Наименование	***escape-последовательность***
звонок (*bell, alert*)	**\a**
шаг назад (*backspace*)	**\b**
перевод формата (*form feed*)	**\f**
новая строка (*newline*)	**\n**
возврат каретки (*carriage return*)	**\r**
горизонтальная табуляция (*horizontal tab*)	**\t**
вертикальная табуляция (*vertical tab*)	**\v**
одиночная кавычка (*single quotation mark*)	**\'**
двойная кавычка (*double quotation mark*)	**\"**
обратная косая черта (*backslash*)	****
вопросительный знак (*literal question mark*)	**\?**

Любой знак также может быть представлен в виде **'\ddd'**, где **ddd** — код символа в восьмеричной системе счисления. Например,

запись **'\012'** обозначает знак *newline*,
запись **'\101'** — букву **A** (лат.),
запись **'\0'** — нулевой знак.

При использовании шестнадцатеричного значения любой знак может быть записан в виде **'\xdd'**, где **dd** — одна или две шестнадцатеричные цифры. Например,

'\x0' — нулевой знак,
'\x41' — буква A (лат.),
'\xA' — символ новой строки.

Следующие примеры описывают один и тот же знаковый литерал:

```
char c = 'a';
char d = '\141';
char e = '\x61';
```

Строковые литералы

Строковые литералы формируют строковые значения заключением последовательности знаков в двойные кавычки. Эта последовательность *может* содержать нулевое количество знаков (пустая строка). Строки в *Си* интерпретируются как массивы *знакового типа*. Примеры:

```
char s1[] = "";    /* пустая строка                        */
char s2[] = " ";   /* строка, состоящая из пробела         */
char s3[] = "A";   /* строка, состоящая из буквы A (лат)   */
char s4[] = "\n";  /* строка, состоящая из знака newline   */
```

Знак **'\x0'** служит признаком конца строки и добавляется к строковому литералу *автоматически*. Поэтому, с учетом нулевого знака в конце строки, длина строкового литерала «Hello», например, равна 6.

Для задания строкового литерала на нескольких строчках используется знак \ (*backslash*). Следующий литерал задает строку «пример литерала, расположенного на двух строчках», расположенный в тексте на двух строчках:

```
char s[] = "пример литерала, расположенного на двух \
```

```
            строчках"
```

Строковый литерал может содержать знаки, заданные *escape*-последовательностями. В примере строковый литерал задает строку, фактически состоящую из двух строк, разделяемых знаком *newline* \n:

```
char s[] = "первая строка\nвторая строка".
```

Если строковый литерал должен содержать знак двойной кавычки, служащий для ограничения самого строкового литерала, этот знак также задается в виде *escape*-последовательности:

```
char s[] = "Фирма \"Аксон\""
```

Не следует путать литералы:

```
'a' – знаковый (символьный);
"a" – строковый.
```

Несмотря на кажущуюся схожесть, они задают значения разных типов.

Так как знак «обратный слеш», используемый в качестве разделителя имен в спецификации файла, обозначает начало *escape*-последовательности, вместо него следует использовать *escape*-последовательность \\. Пример показывает, как правильно записать спецификацию C:\WINDOWS\TEMP:

```
char spec[] = "C:\\WINDOWS\\TEMP";
```

Константные объекты

Константные объекты определяются при помощи ключевого слова const, которое конструирует *объект*, значение которого нельзя изменять, например:

```
const LEN = 81;
const double f = 1.1;
```

Константные объекты используются для инициализации значений переменных, в выражениях, в условиях условного оператора, циклов и оператора выбора.

Константы

В качестве констант в *Си* используются макроподстановки (подробнее см. «Препроцессор»). В следующем примере описаны константы LEN и F:

```
#define LEN 81
#define F  1.1
#define TEMP "C:\\WINDOWS\\TEMP"
```

Константой здесь называется литерал. Отличие константы от константного объекта заключается в использовании памяти. Для константного объекта в программе выделяется память, в которой хранится постоянное значение, изменить которое нельзя. Для константы память не выделяется, вместо этого компилятор «помнит» значение константы и подставляет его непосредственно в код при компиляции.

Типы и переменные

Тип — это множество значений и набор операций, применимых к этим значениям. Например, целый тип представляет некоторый диапазон целых чисел, над которыми могут быть выполнены арифметические и логические операции.

Типы делятся ***основные*** и ***производные***. Основные типы представляют элементарные, неделимые порции информации, а производные типы являются составными и описываются комбинациями типов.

К основным типам относятся ***целый*** (integer), ***вещественный*** (floating point), ***знаковый*** (character), ***перечисляемый*** (enumeration) и ***пустой*** (void).

Указанные типы, исключая тип *void*, называются также ***арифметическими***, так как они представляют числовые значения, над которыми допустимы арифметические операции. Типы *целый*, *перечисляемый* и *знаковый* называются также ***дискретными*** (*integral*).

Производные типы конструируются из основных и производных. К ним относятся массивы (*array)*, функции (*function)*, указатели (*pointer)*, структуры (*struct)* и объединения (*union)*.

Тип определяет некоторый класс значений, выделяя их из бесконечного множества возможных. С этой точки зрения тип *классифицирует* значения по сортам. Компилятор реализует тип в виде ***типа данных***, определяющего конкретные способы хранения того или иного сорта значения. Например, основной тип ***целый***, определяющий *бесконечное* множество целых чисел, реализуется компилятором конкретными типами данных short int, int, long int.

Тип данных определяет *размер памяти*, выделяемой для хранения одного элемента данных. Так, тип данных char хранит знаковое значение в виде одного байта (8 бит), ограничивая количество возможных значений (знаков) числом 2^8 = 256. Одновременно тип данных определяет *способ представления* значения, который задает *диапазон* значений. Например, тип char представляется в виде дополнительного кода и имеет диапазон от –128 до +127.

Типы описывают возможные значения, а держателями конкретных значений являются объекты [программы]. ***Объектом*** называется **область памяти**, выделенная для хранения значения. *Переменные*, *константные переменные* и *функции* являются примерами объектов. Литералы, константы, перечисления, описания типа объектами не являются и память для них не выделяется.

Переменной называется компонент программы, используемый для хранения ассоциированного с ним типа данных. С переменной связаны *идентификатор*, *тип данных* и *значение*. ***Константная* переменная** отличается тем, что ее значение не может быть изменено во время выполнения программы.

Идентификатор отличает один объект программы от другого и компилируется в *адрес младшего байта* объекта. Идентификатор существует и имеет смысл только на стадии компиляции и связывания программы и предназначен для удобства программирования.

Значение переменной зависит от способа интерпретации (подробнее см. конкретные типы). *Присваивание* переменной нового значения вызывает потерю ее старого значения. В языке *Си* переменная не может не содержать никакого значения.

Описанием (*declaration*) объекта называется языковая конструкция, указывающая на свойства объекта без выделения для него памяти. ***Определение*** (*definition*) объекта не только описывает объект, но и выделяет для него память. Например,

```
extern int i;     /* описание    */
int j = 0;        /* определение */
```

Явная инициализация значения переменной и описание функции, включающее фигурные скобки , являются определениями этих объектов.

Описание указывает на тип объекта и необходимо для того, чтобы иметь возможность ссылаться на него ниже в тексте программы. Определение объекта может находиться в другом модуле программы или ниже по тексту. Общее правило — объект должен быть *описан* до того, как впервые будет использован в модуле программы (и не обязательно определен в этом модуле).

Функция является объектом, потому что она, во-первых, занимает память, а во-вторых, является процедурой для вычисления *значения*. Поэтому функция имеет

тип, который возвращается ею после завершения процедуры вычисления. Частный случай функции возвращает пустой тип void (нет возвращаемого значения).

Основные типы

Знаковый тип

Значения, ассоциированные с типом char, представляют собой элементы множества знаков, определенных реализацией языка. В компиляторах *Borland C++* и *Microsoft C* это набор символов кода ASCII.

Значения знаков хранятся как знаковые байтовые целые числа, поэтому знаки можно рассматривать как целые числа и наоборот (*двойственность представления знакового типа*). Хранимое значение знакового типа совпадает с кодом ASCII соответствующего символа. Возвращаемое значение знакового типа зависит от контекста. Если возвращаемое значение присваивается переменной целого типа, возвращается дополнительный код. В примере возвращаемое в i значение отрицательно, так как код ASCII буквы Б равен 129 (кодировка MS-DOS). Если требуется получить хранимое значение, следует использовать явное преобразование к беззнаковому типу (присваивание j):

```
char c = 'Б';
int i = c;                  /* i == -127 */
int j = (unsigned char)c;   /* j ==  129 */
```

Следующий пример показывает различные способы задания одного и того же значения знаковой переменной. Несмотря на то, что присваиваемые переменным c и e значения не совпадают с диапазонами значений знакового и беззнакового типов char, они инициализируются правильно (за счет приведения типа):

```
void main(void) {
   /* хранимое значение во всех случаях 129 */
            char a =  'Б';    /* 'Б' == 129 */
            char b = -127;
            char c =  129;
   unsigned char d =  'Б';
   unsigned char e = -127;
   unsigned char f =  129;
             int i = c;       /* i == -127 */
             int j = e;       /* j ==  129 */
}
```

Двойственность представления знаков дает программисту некоторые дополнительные возможности. Если, например, некоторая функция должна возвращать в результате своего выполнения знаковое значение, то она может быть описана как возвращающая целое значение int, и при этом корректно работать, возвращая как знаки, так и другие числовые значения, например, значение –1. Примером может служить функция fgetc(). Она возвращает очередной знак из потока символов, а в случае достижения конца потока эта функция возвращает значение EOF (end_of_file — конец файла), равное –1.

В силу того, что знаки имеют целочисленное внутреннее представление, над ними могут производиться все операции, свойственные целым типам. В следующем примере над знаковой переменной производится побитовая операция, применяемая к целым типам. Результат операции — перевод знака в нижний регистр:

```
void main(void) {
   char c = 'T';
   c = c | 32;       /* c = 't' */
}
```

Целые типы

Целые типы чрезвычайно важны — они дают возможность считать и нумеровать и используются для обозначения объектов типа *счетчик*, *количество*, *вариант*, *цвет*, *форма* и т. п. Целые типы в *Си* — int, short int (или просто short) и long int (или просто long). Между целыми типами существует следующее соотношение по длине внутреннего представления:

```
sizeof(short int) ≤ sizeof(int) ≤ sizeof(long int).
```

Следующая таблица показывает длину внутреннего представления целых типов для разных компиляторов:

Тип	*Компилятор Borland C++*	*Компилятор Microsoft C*	*Дополнительные типы Microsoft C*
`short int`	16 (2 байта)	16 (2 байта)	`__int8`, `__int16`, `__int32`, `__int64` (1, 2, 4 и 8 байт)
`int`	16 (2 байта)	32 (4 байта)	
`long int`	32 (4 байта)	32 (4 байта)	

Если необходимо целое с длиной внутреннего представления 8 бит, следует использовать тип char.

Указанные типы интерпретируют целое значение как имеющее знак. Ключевое слово unsigned, предшествующее целому или знаковому типу данных, предписывает рассматривать значение как не имеющее знака, то есть использовать хранимое значение. В следующей таблице приведены диапазоны значений целых типов с разной длиной внутреннего представления:

Длина внутреннего представления, бит	*8*	*16*	*32*
знаковое значение	`–128..127`	`–32768..32767`	`–2147483648..2147483647`
хранимое значение	`0..255`	`0..65535`	`0..4294967295`

Учитывая, что для представления целых отрицательных чисел используется дополнительный код, можно говорить о двойственности представления целых чисел в смысле интерпретации хранимого в памяти значения. Представление о соотношении между хранимым и знаковым числовым значением дает следующая таблица:

хранимое значение типа char	*знаковое значение типа char*	*двоичное и шестнадцатеричное представление типа char*
`0`	`0`	`0000 0000 == 0x00`
`1`	`1`	`0000 0001 == 0x01`
. . .		
`126`	`126`	`0111 1110 == 0x7E`
`127`	`127`	`0111 1111 == 0x7F`
`128`	`–128`	`1000 0000 == 0x80`
`129`	`–127`	`1000 0001 == 0x81`
. . .		
`254`	`–2`	`1111 1110 == 0xFE`
`255`	`–1`	`1111 1111 == 0xFF`

Для дополнительного кода, таким образом, характерно:

- число –1 хранится в виде двоичного числа, все разряды которого равны 1;
- минимальное число имеет единицу только в старшем (знаковом) разряде;
- максимальное беззнаковое число является инверсией минимального.

Максимальное значение целого числа в дополнительном коде определяется по формуле $2^{n-1}-1$, а *минимальное* значение — по формуле -2^{n-1}. Здесь n — длина внутреннего представления числа в битах.

Перечисляемые типы

Перечисляемые типы позволяют в качестве значений использовать *идентификаторы*. Перечисление определяет тип как набор целочисленных констант. Применение перечисляемых типов улучшает читаемость программы, позволяя придать значениям выразительность через их имена.

Множество значений, ассоциированное с перечисляемым типом, задается перечислением идентификаторов. Перечисляемый тип ассоциируется с именем (идентификатором) при помощи оператора typedef:

```
typedef enum {red, yellow, green} traffic_light; /* светофор */
```

Переменная перечисляемого типа, описанного выше, объявляется так:

```
traffic_light signal;
```

Использование этого перечисляемого типа поясняет следующий пример:

```
switch(signal) {
   case red: стоп; break;
   case yellow: внимание; break;
   case green: движение разрешено; break;
}
```

Перечисляемый литерал не может быть ассоциирован с двумя различными типами, поэтому следующие два описания несовместимы, так как литералы red и green описаны дважды:

```
typedef enum {red, yellow, green} traffic_light;
typedef enum {red, green, blue} color;
```

Для определения переменных можно использовать *описание метки* перечисляемого типа:

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

Переменная `signal` в этом случае может быть определена так:

```
enum traffic_light signal;
```

Перечисление может быть *анонимным* (не вводить метки или нового имени). В этом случае перечисление используется для определения констант. Следующий пример использует анонимное перечисление и показывает, как можно задать значениям литералов произвольные значения. Литерал ASM имеет значение 0, а литерал BREAK — значение 2:

```
enum {ASM, BREAK = 2};
int i = ASM;
i = i + BREAK;
```

Литералы перечисления могут использоваться в программе как константы в выражениях и инициализаторах.

Логические значения

В языке *Си* вместо логических значений используются целые типа int. Любое ненулевое целое значение интерпретируется как *истинное*, а нулевое — как *ложное*. По соглашению, заранее определенные операторы и функции возвращают 1 в качестве истинного значения. Значение –1 в качестве истинного удобно тем, что оно является инверсией значения *ложь*.

При необходимости в программе могут быть определены константы для идентификации логических значений одним из следующих способов:

```
#define FALSE 0
#define TRUE  1
```

или

```
enum {FALSE, TRUE};
```

Типы с плавающей точкой

Типы с плавающей точкой представляют действительные числа со знаком и используются в программах для обозначения величин типа *скорость*, *площадь*, *объем*, *масса*, *значение производной* и т.п.

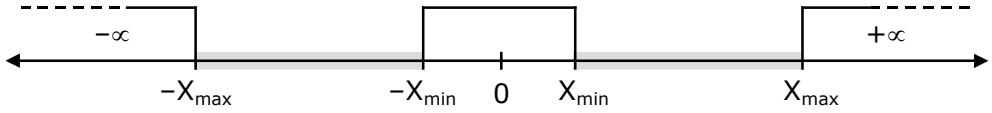
Компилятор *Microsoft C* определяет два вещественных типа — короткий float (4 байта) и длинный double (8 байт). Компилятор *Borland C++* дополнительно определяет тип long double (10 байт).

Вещественные типы, в отличие от целых, представляют числовые значения с некоторой степенью точности, которая определяется максимальным числом значащих цифр мантиссы внутреннего представления. Для числа одинарной точности мантисса состоит из 24 двоичных разрядов, что примерно соответствует 6 десятичным значащим цифрам числа. Число двойной точности имеет во внутреннем представлении 53 двоичных разряда, т.е. примерно 15 десятичных значащих цифр. В связи с неточностью представления возникает вопрос о мере точности. За *абсолютную* единицу точности *epsilon* принимается вес младшей значащей цифры двоичной мантиссы, т. е. $2^{-23} = 1{,}2 \cdot 10^{-7}$ для типа float, и $2^{-52} = 2.2 \cdot 10^{-16}$ для типа double. Любое число в диапазоне 1..2 представляется с точностью, не превышающей абсолютной единицы точности:

|точное_значение_числа — фактическое_значение_числа| <= epsilon.

Относительной мерой точности является отношение единицы точности к числу. Так как мантисса имеет минимальное значение, равное единице, максимальная относительная погрешность равна *epsilon*.

Диапазон представления вещественных чисел состоит из двух частей — положительной и отрицательной, расположенных симметрично относительно нуля. Само число 0 не входит в диапазон представления, хотя для обозначения нуля есть специальное значение. В отношении вещественных чисел имеют смысл такие понятия, как *машинный нуль* и *машинная бесконечность*.



Машинный нуль — это диапазон вещественных чисел, расположенный симметрично вокруг значения 0, который не может быть представлен никакими значениями. Любое число из этого диапазона интерпретируется как нулевое. Например, можно с уверенностью сказать, что число 10^{-10000} не может быть представлено в памяти, следовательно, это число может быть интерпретировано только как 0.

Машинная бесконечность, в отличие от машинного нуля — это два диапазона чисел, которые простираются от некоторых положительного и отрицательного значений до бесконечности (положительной или отрицательной). Числа этих диапазонов также не могут быть представлены в памяти, поэтому они должны быть интерпретированы каким-либо допустимым значением, например, соответствующим максимальному или минимальному числу, которые можно представить в заданном формате. Точные значения максимальных и минимальных значений вещественных чисел а также точное значение границы машинного нуля следует выяснять в конкретной реализации языка. На практике можно использовать значения, приведенные в следующей таблице.

Тип	*Длина, бит*	*Минимальное число*	*Максимальное число*
`float`	32	3.4e-38	3.4e38
`double`	64	1.7e-308	1.7e308
`long double`	80	3.4e-4932	1.1e4932

В связи с особенностями чисел с плавающей точкой следует следовать определенным правилам при сравнении двух таких чисел. Примеры сравнений:

```
        x  <  -epsilon       вместо   x <  0
    abs(x) <=  epsilon       вместо   x =  0
    abs(x) >   epsilon       вместо   x != 0
        x  >   epsilon       вместо   x >  0
abs(x - a) <=  epsilon       вместо   x =  a
abs(x - a) >   epsilon       вместо   x != a
        x  <   a - epsilon   вместо   x <  a
        x  >   a + epsilon   вместо   x >  a
```

Тип void (пустой)

Определяет пустое множество значений. Используется как тип функции, которая не возвращает значение, а также для указания на отсутствие параметров функции:

```
void main(void) {

}
```

Производные типы

Массивы

Массив — это объект, состоящий из компонентов (элементов) одного и того же типа. Тип элемента может быть одним из основных, типом другого массива, типом указателя, типом структуры (struct) или типом объединения (union).

Элементы массива отличаются от других объектов программы тем, что они имеют одно и то же имя (идентификатор), а различие между элементами проявляется при использовании операции индексирования [], параметр которой (порядковый номер элемента) позволяет вычислить адреса элемента:

```
char s[10];     /* массив из 10 знаковых значений       */
s[0] = '\x0';   /* обращение к первому элементу массива */
```

В языке *Си* индекс первого элемента массива всегда равен нулю. Для вычисления адреса конкретного элемента массива используется сложение адреса массива (то есть адреса первого элемента) с номером элемента. Поскольку эта операция выполняется над адресами, она дает правильный результат, так как *адресная единица* по величине равна размеру элемента.

Например, в программе объявлен массив из 5 целых значений:

```
int a[5];
```

Оператор

```
a[4] = 5;
```

использует операцию a + (4 * 2) для вычисления адреса пятого элемента, где 4 — индекс элемента, а 2 — размер элемента в байтах (адресная единица). Если бы индекс элемента исчислялся от единицы, пришлось бы дополнительно вычитать эту единицу из индекса элемента. На практике обращение к пятому элементу указанного массива можно осуществить при помощи оператора

```
4[a] = 5;
```

так как важным является лишь то, что указано имя массива, которое позволяет определить тип элемента массива и узнать, чему равна единица адреса, и индекс требуемого элемента, в данном случае 4.

Другой особенностью массивов в *Си* является отсутствие проверки границ массива к обращении к его элементам. Это может привести к ошибкам или неожиданным результатам.

Следующий пример выводит на экран значение 5. Дело в том, что память под элементы данных в компиляторе *Borland* C++ выделяется в порядке, обратном определению (так как локальные переменные располагаются в стеке). Поэтому в этой программе сначала выделяется память под массив, а затем под переменную b.

```
void main(void) {
   int b = 0, a[5];
   a[5] = 5;
   printf("%d\n", b);
}
```

Переменная b занимает в памяти место шестого элемент массива, если бы таковой был. Оператор a[5] = 5; устанавливает значение несуществующего шестого элемента, в результате изменяется значение переменной b.

Многомерные массивы могут быть описаны следующим способом:

```
int a[2][3][4];
```

Этот оператор описывает трехмерный массив с размерностью 2 массива, имеющих размерность 3 массива по 4 элемента типа int. Указывая не все индексы многомерного массива, можно ссылаться на его подмассивы, например, ссылка

```
a[1]
```

указывает на второй двумерный подмассив размерностью 3 массива по 4 элемента, а ссылка

```
a[1][1]
```

указывает на второй одномерный подмассив второго двумерного подмассива размерностью 4 элемента.

Структуры

Структура — это составной объект, элементы которого могут иметь разные типы. Эти типы могут быть любыми, за исключением функций. В отличие от массива, который является однородным, структура чаще всего неоднородна. Структуре языка *Си* в других языках программирования соответствуют записи и пользовательские типы. Анонимная структура указывается записью вида

```
struct {
   список_описаний
} список_переменных;
```

Например:

```
struct {
   double x, y;
} a, *b = &a, c[5];
```

С типом структуры может быть ассоциировано имя, которое задается описанием типа в форме

```
typedef struct {
   список_описаний
} имя_типа_структуры;
```

Например:

```
typedef struct {
   double x, y;
} point;
```

Переменные этого типа описываются обычным образом:

```
point a, *b = &a, c[5];
```

Для ассоциирования имени с типом структуры могут быть использованы метки структуры, так же, как метки перечисляемого типа для ассоциирования с типом перечисления. Метка структуры описывается следующим образом:

```
struct метка {
   список_описаний
};
```

Например:

```
struct point {
   double x, y;
};
```

Здесь метка является идентификатором.

Для описания переменных типа метки структуры следует использовать запись с указанием метки:

```
struct point a, *b = &a, c[5];
```

Для доступа к компонентам структуры используется операция доступа к компоненту структуры, которая для статической переменной является точкой, а для динамической — символом стрелка. Например, для переменной a доступ к компонентам x и y осуществляется так:

```
a.x = 1;
a.y = 9;
```

Переменная b имеет тип указателя на структуру point:

```
b->x = 1;
b->y = 9;
```

Для компонентов переменной c доступ к элементам структуры осуществляется так:

```
c[0].x = 1;
c[0].y = 9;
```

Метки структуры необходимы для описания рекурсивных структур. Структуры не могут быть прямо рекурсивными, т.е. структура типа s не может содержать компонент, являющийся структурой типа s. Однако структура типа s может содержать компонент, указывающий на структуру типа s.

```
struct node {
   double data;
   struct node *next;
};
```

Поля битов

Структуры используются также для доступа к отдельным битам слова, например, для доступа к битам регистра устройства. Поле описывается указанием имени и длины в битах. Тип поля — unsigned.

В приведенной ниже программе структура поля video_word описывает слово видеопамяти, состоящее из байта — кода ASCII и байта атрибутов. Байт атрибутов состоит из полей: 4 бита для цвета плана, 4 бита для цвета фона и 1 бит для признака мерцания. Программа выводит на экран шестнадцатеричное значение 7041, описывающее букву A (латинская) черным цветом на сером фоне без мерцания:

```
#include <stdio.h>
struct {
   unsigned ascii     : 8;
   unsigned forecolor : 4;
   unsigned backcolor : 3;
   unsigned blink     : 1;
} video_word;
void main(void) {
   video_word.ascii = 'A';
   video_word.forecolor = 0;
   video_word.backcolor = 7;
```

```
   video_word.blink = 0;
   printf("%X\n", video_word);
}
```

Если некоторые биты слова не используются, они могут быть обозначены как поля без имени, например, в следующей структуре не используется первый байт слова видеопамяти:

```
struct {
   unsigned           : 8;
   unsigned forecolor : 4;
   unsigned backcolor : 3;
   unsigned blink     : 1;
} video_word;
```

Объединения

Объединение похоже на структуру в смысле композиции разнородных компонентов, но в каждый момент времени оно представляет только один (любой) из его компонентов, так как все компоненты объединения используют (разделяют) одну и ту же область памяти.

Объединение можно использовать для экономии памяти. В этом случае одна и та же область памяти в разные моменты исполнения программы хранит любое из значений, указанных в описании. Например, если в программе описано объединение

```
union {
   int i;
   float f;
} a_union;
```

то переменная a_union.i хранит в объединении целое значение, а a_union.f — вещественное. Сначала объединение можно, например, использовать для хранения целого значения, а затем вещественного (или наоборот). Если в объединение записано вещественное значение, то и считывать имеет смысл только вещественное, так как целое и вещественное представления имеют различные внутренние форматы. При использовании объединения в этом случае важно не потерять смысл хранимого значения.

В некоторых случаях, однако, именно изменение смысла хранимого значения является целью объединения. Например, следующее объединение позволяет выделить в слове word его младший байт byte:

```
union {
   unsigned int word;
   unsigned char byte;
} wtob;
```

Для этого сначала в объединение записывают компонет-слово, а затем считывают компонент-байт. Поскольку представление слова в памяти начинается с младшего байта, компонент-байт разделяет область, занимаемую именно младшим байтом слова. Если требуется таким же образом получить доступ к старшей части слова, в качестве второго компонента следует использовать дополнительную структуру, состоящую из двух байтовых величин:

```
#include <stdio.h>
struct bytes {
   unsigned char low, high;
};
union {
   unsigned int word;
   struct bytes byte;
} w2b;
```

```
void main(void) {
   w2b.word = 256;
   printf("%d\n%d\n", w2b.byte.low, w2b.byte.high);
}
```

Эта программа выводит на экран значения 0 и 1.

Описание объединения производится аналогично описанию структуры.

Переменные структуры

Переменная структура является комбинацией структуры и объединения. В языке Паскаль переменной структуре соответствует запись с вариантами. Эти структуры данных используются тогда, когда хранимые значения можно разделить на общую для всех наборов данных часть и переменную часть, структура которой изменяется в зависимости от набора данных.

Например, для описания геометрической фигуры можно использовать структуру, в которой в качестве общей части используются площадь и периметр фигуры, а в качестве переменной — основные параметры, описывающие фигуру. Круг описывается радиусом, прямоугольник — двумя сторонами, треугольник — тремя сторонами. Кроме того, понадобится еще компонент, указывающий на тип фигуры, которую описывает конкретная переменная. В результате мы получим следующую переменную структуру:

```
typedef struct {
   double area, perimeter;  /* общая часть             */
   int type;                /* тип описываемой фигуры */
   union {
      double radius;        /* окружность              */
      double a[2];          /* прямоугольник           */
      double b[3];          /* треугольник             */
   } fig_data;
} figure;
```

При работе с этой структурой нужно задать тип хранимой информации, используя компонент type. При чтении структуры сначала проверяют значение type, а затем на основании типа считывают те или иные параметры.

Указатели

Указатель — это переменная (область памяти), предназначенная для хранения адреса. Указатели являются производными типами потому что они зависимы от типов указываемых объектов. Можно сказать, что указатели в *Си* типизированы. Две важных области применения указателей — это динамические объекты и адресная арифметика (вычисление адресов объектов).

Динамический объект отличается от статического тем, что не имеет имени, которому можно было бы присвоить адрес объекта. Поэтому адрес динамического объекта запоминается в переменной типа указатель. В следующем примере создается динамический объект типа int, который затем используется для хранения значения. Указатель хранит адрес этого объекта:

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

void main(void) {
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 33;
   printf("%d\n", *p);
}
```

Второе важное назначение указателей связано с массивами. Суть здесь та же — адрес конкретного элемента массива не связан ни с каким именем, потому что имя массива связывается с адресом первого элемента массива. Таким образом, имя массива — это указатель [на первый элемент массива]. Для обращения к другим элементам массива следует вычислить адрес элемента (получить указатель на него).

Для вычисления адреса элемента массива используется адресная арифметика, которая определяет операции над адресами. Предположим, что в программе объявлен массив из 5 целочисленных значений:

```
   int a[5];
```

Поскольку для размещения одного элемента типа int требуется 2 байта памяти, в качестве единицы адреса для указателя типа «указатель на int» используется число 2 (указатель на массив a типизируется как «указатель на int»). Теперь для вычисления адреса элемента массива достаточно взять индекс элемента и сложить его с указателем на первый элемент. Индекс будет преобразован к единице адреса 2, то есть индекс 1 будет заменен на 2, индекс 2 — на 4 и т.д.

Указатели отличаются наличием знака * (звездочка) перед именем указателя в его описании. Этот же символ используется для разыменования указателя, то есть для того, чтобы получить значение объекта вместо его адреса. Для указателей характерно так же и то, что они могут быть не инициализированы, то есть не указывать ни на какой объект (не являться ничьим адресом). Для взятия (определения) адреса объекта используется операция &.

Например, в программе есть переменная целого типа и указатель на целое:

```
int i = 33;          /* переменная типа int */
int *p;              /* указатель на int    */
```

Знак * перед именем p говорит о том, что p — это указатель. В этом объявлении указатель p не инициализирован. Для инициализации ему нужно присвоить значение адреса какого-либо объекта типа int, например, адрес объекта i:

```
p = &i;              /* p указывает на i */
```

После инициализации указатель p можно использовать для доступа к значению i посредством разыменования:

```
*p = *p + 22;        /* i получает значение 55 */
```

Как видно из этой строки, разыменованный указатель можно использовать и в правой и в левой части. То же самое можно записать при помощи следующего оператора:

```
*p += + 22;          /* i получает значение 55 */
```

Между указательными типами нет автоматических преобразований, т.е. нельзя указателю на int присвоить значение указателя на char или наоборот. Между тем, существует специальный тип указателя void*, который может получить значение любого указателя. Его следует понимать как «указатель на нечто». Любой другой указательный тип автоматически преобразуется к типу void* при присваивании или инициализации. В следующем примере с помощью типа void* значение адреса переменной типа int присваивается указателю на char:

```
int i = 33;
void *vp;
char *cp;
vp = &i;
cp = vp;
printf("%c\n", *cp);
```

Однако это является далеко не лучшей идеей.

Указатель типа void* преобразуется в указатель другого типа, если использовать явное преобразование. Например, для приведенного выше примера оператор

```
printf("%c\n", *vp);
```

будет ошибочным, а оператор (использующий явное преобразование)

```
printf("%c\n", *(char*)vp);
```

будет правильным.

В программах, работающих с аппаратурой компьютера напрямую, иногда необходимо получить доступ к конкретным ячейкам памяти или регистрам устройства. Хотя это полезное и удобное средство программирования, использовать его следует с осторожностью, так как любое прямое управление аппаратурой потенциально небезопасно. Доступ к ячейке памяти можно получить при помощи указателя и явного преобразования (к типу используемого значения):

```
char far *p = (char far *)0x00000449;
printf("%X\n", *p);
```

В этой программе определяется указатель на ячейку памяти с адресом 0x449, которая хранит номер текущего видеорежима MS-DOS.

Отличительной особенностью компилятора *Borland* C++ для компьютера на базе процессора *Intel* является различие между *ближним* и *дальним* указателями. Под ближним указателем понимается адрес объекта в пределах сегмента (т.е. смещение). Ближний (или короткий) указатель дополнительно может иметь обозначение near и занимает в памяти 2 байта. Дальний (или длинный) указатель обозначается при помощи ключевого слова far, занимает в памяти 4 байта и включает в себя сегмент и смещение. По умолчанию в среде *Borland* C++ указатели считаются ближними.

Пример описания дальнего указателя:

```
long far *p;
```

Массивы и указатели

Если в программе используется массив, доступ к элементам массива может быть осуществлен при помощи указателей и адресной арифметики. Массив — это последовательность расположенных подряд элементов одного типа. Разница между адресами соседних элементов массива равна размеру элемента (в байтах). Если размер элемента массива принять за единицу адреса, изменение адреса на 1 будет перемещать адрес от одного элемента массива к другому, соседнему.

Для операций с адресами используются стандартные операции инкремента, декремента, сложения и вычитания, перегруженные для конкретного типа указателя, который определяется типом, указанным в объявлении объекта. Рассмотрим пример, в котором для адресации элементов массива используется адресная арифметика:

```
int a[5], i;
for (i = 0; i < 5; i++) {
   *(a + i) = i;
}
```

Здесь элементам массива последовательно присваиваются значения, равные индексу элемента в массиве. Для указания на конкретный элемент используется конструкция *(a + i). Знак * означает, что это указатель, то есть адрес. Выражение в скобках вычисляет адрес элемента путем сложения адреса первого элемента a с индексом элемента i. Результат получается верным, несмотря на то, что переменная i принимает значения от 0 до 4. Тот факт, что переменная a имеет тип «указатель на целое», заставляет компилятор прибавлять к адресу первого элемента не значение i, а i, умноженное на размер элемента массива, в данном случае на 2. Ничего не нужно менять в программе, если массив a будет объявлен как знаковый или имеющий тип double. Адреса элементов массива все равно бу

дут правильно вычисляться, потому что операция + будет перегружена для указателя заданного типа.

Рассмотрим другой пример, в котором указатель на массив передается в функцию asum, которая вычисляет сумму элементов массива:

```
#include <stdio.h>

int asum(int *m, int n) {
   int *p = m, i = 0, s = 0;
   while (i++ < n) s += *p++;
   return s;
}

void main(void) {
   int a[5] = {1, 2, 3, 4, 5};
   printf("%d\n", asum(a, 5));
}
```

Внутри функции asum используется указатель на целое p, которому присваивается значение указателя на массив m. Элементы массива адресуются вычислением нового значения указателя p в каждой итерации цикла за счет операции инкремент (++). В первой итерации указатель p имеет значение адреса первого элемента массива, полученное им в результате оператора int *p = m. После того, как элемент массива будет считан, выполняется инкремент указателя ++. Так как указатель имеет тип int, инкремент увеличивает значение p не на 1, а на 2, обеспечивая правильное вычисление адреса следующего элемента массива.

Заметим, что вместо указателя p указателя можно использовать указатель m, что упрощает функцию asum:

```
int asum(int *m, int n) {
   int i = 0, s = 0;
   while (i++ < n) s += *m++;
   return s;
}
```

Чтобы еще лучше разобраться с указателями, стоит попробовать и другие варианты. Например, в следующем примере мы пробуем вычислить сумму элементов массива непосредственно в основной функции, используя указатель на массив a:

```
void main(void) {
   int a[5] = {1, 2, 3, 4, 5};
   int i = 0, s = 0;
   while (i++ < 5) s += *a++;
   printf("%d\n", s);
}
```

Увы, это не сработает, потому что операция a++ изменяет адрес объекта, что недопустимо.

Изменить можно указатель, например:

```
void main(void) {
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a, i = 0, s = 0;
   while (i++ < 5) s += *p++;
   printf("%d\n", s);
}
```

Внутрь функции asum в первоначальном примере была передана копия указателя на объект, поэтому мы могли изменять значение указателя m.

Оператор индексации [] дает более короткое выражение для операций с указателями. Индексное выражение a[i] эквивалентно *(a + i). Используя индексацию, мы получим следующую функцию asum:

```
int asum(int *m, int n) {
   int i = 0, s = 0;
   while (i < n) s += m[i++];
   return s;
}
```

Обратим внимание, что операция инкремента индекса элемента перенесена из условия внутрь цикла, в противном случае функция вычисляла бы неверное значение.

Оператор индексации может быть применен к любому указателю, а не только к имени массива. Чтобы лучше понять это, попробуйте следующий пример:

```
#include <stdio.h>

void main(void) {
   int i = 1;
   int j = 2;
   int k = 3;
   int *p = &j;
   p[-1] = 1;
   p[1] = 3;
   printf("%d\n%d\n%d\n", i, j, k);
}
```

Эта программа выводит на экран значения 3, 2 и 1. При анализе следует учитывать, что память под переменные выделяется компилятором в порядке, обратном объявлению.

Строки и указатели

Строки в языке *Си* — это массивы знаковых (байтовых) значений. Все, что сказано о массивах и указателях, справедливо и для строк. Единственное важное отличие строки от массива другого типа заключается в том, что нулевое значение какого-либо элемента массива является признаком конца строки.

Рассмотрим следующие объявления:

```
char *s_ptr, s_arr[9];
```

Первая из переменных — это знаковый указатель, вторая — знаковый массив. В силу сказанного о массивах выше, вторая переменная также может рассматриваться как знаковый указатель. С другой стороны, поскольку знаковые массивы используются для представления строк, обе переменных можно рассматривать как строковые. Память под первую строку следует выделить явно, для массива s_arr память уже выделена. При резервировании памяти под строку следует предусмотреть дополнительный элемент массива под завершающее строку нулевое значение. Следовательно, в массиве s_arr можно разместить строку длиной 8 знаков. Память для первой строки можно выделить оператором

```
s_ptr = (char *)malloc(9 * sizeof(char));
```

Для того, чтобы присвоить значение той или другой строке, можно попробовать применить оператор присваивания, как в случае с другими типами объектов

```
s_arr = "Hello";
```

однако этот оператор ошибочный, так как с левой стороны оператора присваивания находится указатель.

Строка как таковая не является объектом языка. Строки представляются как массивы знаков, поэтому присваивание значений строковым переменным должно происходить по правилам работы с массивами. Например, так:

```
s_arr[0] = "H";
s_arr[1] = "e";
s_arr[2] = "l";
s_arr[3] = "l";
s_arr[4] = "o";
s_arr[5] = 0;
```

На самом деле так никто обычно не поступает. Вместо этого следует использовать функции стандартной библиотеки string. В этом случае программа, инициализирующая строки значениями, может иметь вид:

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
void main(void) {
   char *s_ptr, s_arr[9];
   s_ptr = (char *)malloc(9 * sizeof(char));
   strcpy(s_arr, "Hello, ");
   strcpy(s_ptr, "World!\n");
   printf("%s%s\n", s_arr, s_ptr);
}
```

Эта программа выводит сообщение «Hello, World!».

Следует обратить внимание на то, что строка заканчивается знаком 0, и несмотря на то, что массивы строк имеют размер 9 элементов, в обоих массивах используется только 8 элементов. Функция strcpy() помещает нулевой знак в конец копируемой строки автоматически.

Программист должен сам позаботиться о том, чтобы для размещения строки было выделено достаточно места. Следующая программа выводит на экран сообщение «Hello, World!» в первой строке и «orld!» во второй:

```
#include <stdio.h>
#include <string.h>
void main(void) {
   char sa[8], sb[8];
   strcpy(sa, "1234567");
   strcpy(sb, "Hello, World!");
   printf("%s\n%s\n", sb, sa);
}
```

Так как для размещения строки sb места недостаточно, функция strcpy() остаток строки разместила в памяти последовательно, то есть поверх массива sa. В результате буква o слова World оказалась на месте первого элемента массива sa.

Особенностью компилятора является также то, что при выделении памяти под массив всегда выделяется четное количество байт, так что под знаковые массивы с объявленной длиной 7 и 8 байт будет выделено по 8 байт.

Описания типа (typedef)

Описание типа вводит новое имя типа (но не новый тип) для существующих типов или типов, которые могут быть образованы из существующих. Это новое имя является синонимом типа.

```
typedef спецификатор_типа новое_имя;
```

Спецификатор_типа — это любое допустимое описание объекта. Новое имя используется затем для описания других объектов. Например, описание

```
typedef double speed;
```

вводит новое имя типа speed, являющееся синонимом типа double. С помощью нового имени типа можно определять новые объекты так же, как с помощью стандартных типов:

```
speed a, b;
```

В следующем примере вводится новое имя pstr для типа «указатель на знак»:

```
typedef char *pstr;
char a[] = "Hello";
pstr s = a;
printf("%s\n", s);
```

В следующем примере новое имя str описывает тип знакового массива:

```
typedef char str[8];
str s = "Hello";
printf("%s\n", s);
```

Описание типа используется для ввода новых имен структур и объединений:

```
typedef struct {                    /* новая структура bytes */
   unsigned char low, high;
} bytes;
typedef union {                     /* новое объединение w2b */
   unsigned int word;
   bytes byte;
} w2b;
```

а также перечислений:

```
typedef enum {false, true} bool; /* новое перечисление bool */
bool func(void);                 /* и его использование     */
```

Описание типа используется также для сокращения описаний объектов введением новых имен для часто используемых типов. Например, можно встретить описания

```
typedef unsigned short ushort
typedef unsigned char uchar
```

которые сокращают объявление объектов этих типов:

```
ushort i, j;
uchar c, ch;
```

Другая причина использования имен состоит в необходимости обеспечения переносимости программ между различными платформами. Разные реализации языка имеют разные машинные представления для стандартных типов, например целочисленных. На одной машине тип int ассоциируется с 16-битовым представлением, а на другой — с 8-битовым. В результате при переносе программы с одной машины на другую возможны ошибки.

Чтобы свести к минимуму модификацию программы, следует определить новое имя для типа, например:

```
typedef int integer;
```

и использовать в программе машинно-независимое имя типа integer. При переходе к другой платформе достаточно изменить описание:

```
typedef char integer;
```

Использование описания типа полезно при конструировании сложного типа, такого, например, как массив указателей на функцию целого типа:

```
/* foo — функция целого типа */
typedef int  foo(void);
/* pfoo — указатель на функцию целого типа  */
typedef foo  *pfoo;
/* apfoo — массив указателей ... */
typedef pfoo apfoo[2];
```

Имя такого же типа можно определить и одним описанием типа, только сконструировать его сложнее:

```
typedef int (*apfun[2])(void);
```

Ниже приведен пример, в котором описывается функция, возвращающая указатель на знак и принимающая указатель на знак. В примере показано, как описать указатель на такую функцию и инициализировать его:

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
/* функция, возвращающая указатель на знак и принимающая аргумент-строку */
typedef char *pifoo(char *);
/* указатель на функцию ...  */
typedef pifoo *ppifoo;
/* то же */
typedef char *(*ppif)(char *);

/* функция возвращает строку, содержащую первый знак аргумента */
char * first(char * s) {
   char *a = (char *)malloc(2);  /* выделяет память 2 байта */
   a[0] = s[0];      /* первый байт - первый знак аргумента */
   a[1] = 0;         /* второй байт - конец строки          */
   return a;
}

void main(void) {
   char *s;
   ppifoo f1 = first;
   ppif f2 = first;
   s = f1("Hello");
   s = f2("World");
}
```

Операция косвенной ссылки * (звездочка) имеет меньший приоритет в сравнении с операциями () и []. Это означает, что оператор int *f() описывает функцию, возвращающую указатель на целое, а оператор int (*f)() описывает указатель на функцию, возвращающую целое. Во втором случае скобки вокруг идентификатора и звездочки предписывают сначала «прочитать» оператор «указатель», а затем оператор «функция».

Для разбора сложного описания типа следует следовать алгоритму:

1) если справа от идентификатора есть квадратные или круглые скобки, интерпретировать их;
2) если слева от идентификатора есть звездочка, интерпретировать ее;
3) если встретилась правая круглая скобка, вернуться назад и применить правила 1 и 2 для всего, что в скобках. После интерпретации скобок исключить их и продолжить с пункта 1.

Например, описание типа, приведенное ниже, интерпретируется в последовательности, обозначенной цифрами:

```
char *(*(*id)())[10];
  7  6 4 2 1  3   5
```

и означает:

1) идентификатор id — это
2) указатель на
3) функцию, возвращающую
4) указатель на
5) массив из 10 элементов, которые являются
6) указателями на
7) знак

Описание, определение и инициализация

Объект программы должен быть описан до первого обращения к нему в программе. Описание объекта является его определением, если оно выделяет память для хранения значения объекта. Стиль, используемый при описании объектов языка *Си*, отличается от описания объектов в других языках программирования. Сначала указывается класс и тип объекта, а затем его описатель. За каждым описателем объекта может следовать инициализатор, задающий начальное значение, ассоциированное с идентификатором, используемым в описателе:

класс_памяти тип_данных описатель = инициализатор;

В языке *Си* описания и определения переменных должны располагаться только в начале блока.

Классы памяти

Класс памяти определяет время жизни и область действия объекта. Существует четыре класса памяти:

auto

АВТОМАТИЧЕСКИЙ КЛАСС ПАМЯТИ — используется для локальных идентификаторов, память для которых выделяется при входе в блок {} и высвобождается при выходе из него;

static

СТАТИЧЕСКИЙ КЛАСС ПАМЯТИ — используется для локальных идентификаторов, память для которых выделяется один раз во время инициализации программы и освобождается при завершении ее работы;

extern

ВНЕШНИЙ КЛАСС ПАМЯТИ — используется для описания объектов, экспортируемых из других модулей программы. Память для этих объектов выделяется в месте их определения;

register

РЕГИСТРОВЫЙ КЛАСС ПАМЯТИ — используется для локальных идентификаторов, значения которых по возможности должны размещаться в регистрах процессора (для ускорения обработки).

Если класс памяти объекта не указан явно, он определяется расположением описания или определения объекта в тексте программы.

- Объект, описанный внутри блока, по умолчанию имеет класс auto.
- Объект, описанный вне любой функции, по умолчанию имеет класс extern.

Если объект объявлен вне любой функции с явным указанием класса extern, память под объект выделяется в другом модуле программы, объявление является описанием объекта и служит для проверки типа и правильной компиляции текущего модуля программы.

Если объект объявлен вне любой функции без указания класса, он считается принадлежащим классу extern, объявление является определением и память под объект выделяется в текущем модуле программы. Так как описание внешнего идентификатора может встретиться в нескольких модулях программы, только один из модулей должен содержать его определение.

- Область действия идентификатора, объявленного как внешний, распространяется на все модули, где встречается описание этого идентификатора.
- Область действия локального объекта (объявленного внутри блока) распространяется до конца блока.
- Область действия статического идентификатора, объявленного внутри блока, распространяется до конца блока, хотя объект существует до входа в блок и после выхода из него.

- Область действия статического идентификатора, объявленного вне любой функции, распространяется на текущий модуль программы.

Рассмотрим пример описаний и определений двух переменных, расположенных вне любых функций. Проект состоит из двух модулей. Первый из модулей содержит определения

```
int i;          /* определение */
static int j;   /* определение */
```

а второй модуль содержит описание и определение

```
extern int i;   /* описание    */
static int j;   /* определение */
```

Переменная i определена в первом модуле и здесь же для нее выделена память. Эта переменная может быть использована для связи (передачи значений) между функциями двух модулей.

Переменные j определены в каждом из модулей и каждый из модулей выделяет для них память. Область действия этих переменных распространяется только на тот модуль, внутри которого они определены — они, таким образом, локальные, но их локальная область равна одному модулю.

Типы данных

Для описания или определения объекта может быть использован один из следующих типов данных:

```
char
int
short int (или просто short)
long int (или просто long)
unsigned char
unsigned int (или просто unsigned)
unsigned long
float
double
long double
void
struct метка
union метка
enum метка
typedef-имя
```

Метка и *typedef*-имя должны быть предварительно описаны.

Если тип данных в объявлении не указан явно, объект считается имеющим тип данных int.

Описатели

Описатель содержит идентификатор (имя объекта), а также некоторые дополнительные указания:

- если идентификатору предшествует знак звездочка, объект является указателем на заданный тип данных;
- если после идентификатора следуют круглые скобки, объект является функцией, возвращающей значение заданного типа данных; внутри скобок могут располагаться описатели параметров, состоящие из типов параметров через запятую;
- если после идентификатора следуют квадратные скобки, объект является массивом элементов заданного типа данных. Внутри скобок может располагаться выражение с вычисляемым на стадии компиляции значением. Это выражение определяет количество элементов массива.

Ограничения на описатели:
- функция не может возвращать массив и функцию;
- элементами массива не могут быть функции;
- функции не могут быть компонентами структур или объединений.

Примеры описаний

Ниже приведены примеры наиболее часто используемых описаний:

```
int i, *pi, fi(void), *fpi(void), (*pfi)(void), *api[];
```

Описатели этих описаний имеют следующий смысл:
`i` — целое;
`pi` — указатель на целое;
`fi` — функция, возвращающая целое значение;
`fpi` — функция, возвращающая указатель на целое;
`pfi` — указатель на функцию, возвращающую целое;
`api` — массив указателей на целое.

Дополнительно об описаниях см. «Описание типа (typedef)».

Инициализация

Инициализатор определяет объект, так как значение должно быть размещено в памяти. Инициализация переменных скалярных типов тривиальна:

```
int i = 1;
double d = 1.0;
char c = 'A';
```

Инициализация массивов осуществляется перечислением значений элементов, заключенным в фигурные скобки:

```
int a[5] = {1, 2, 3, 4, 5};
```

Размерность массива можно не указывать (вычислит компилятор):

```
int a[] = {1, 2, 3, 4, 5};
```

В следующем определении заданы значения только первых трех элементов:

```
int a[5] = {1, 2, 3};
```

Для многомерного массива

```
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Инициализация знакового массива

```
char a[] = "Hello";
```

резервирует в памяти на один знак больше, чем число символов в литерале.

Следующее описание компилятор допускает, но оно ошибочное и может привести к логическому разрушению памяти, если использовать объект `a` в качестве строки знаков:

```
char a[6] = "Hello!";
```

Пример инициализации структуры:

```
typedef struct {
   int x, y;
} point;
point p = {3, 8};
```

Преобразования типов

Неявное преобразование типа

Выполняется компилятором автоматически для согласования аргументов оператора или функции с предполагаемыми значениями:

char	преобразуется к	int, short, long с расширением знакового разряда;
int	преобразуется к	short, long с расширением знакового разряда;
int	преобразуется к	char с усечением старших разрядов;
short	преобразуется к	int, long с расширением знакового разряда;
short	преобразуется к	char с усечением старших разрядов;
long	преобразуется к	char, int, short с усечением старших разрядов;
float	преобразуется к	double;
float	преобразуется к	int, short, long (если возможно);
double	преобразуется к	float с округлением.
double	преобразуется к	int, short, long (если возможно);

Арифметические преобразования

Выполняются при вычислении выражений по правилам:

- операнды типа char и short преобразуются к типу int; операнды типа float преобразуются к типу double.
- если хотя бы один из операндов имеет тип double, другой операнд преобразуется к типу double; результат имеет тип double.
- если хотя бы один из операндов имеет тип long, другой операнд преобразуется к типу long; результат имеет тип long.
- если хотя бы один из операндов имеет тип unsigned, другой операнд преобразуется к типу unsigned; результат имеет тип unsigned.
- если ни один из перечисленных случаев не имеет места, операнды и результат должны иметь тип int.

Явные преобразования типов

Выполняются по прямому указанию программиста посредством записи

(имя_типа)выражение

где имя_типа определяет тип данных, к которому требуется преобразовать значение выражения. Пример явного преобразования:

```
int i = 1;
double d;
d = (double)i;
```

Это преобразование выполняется автоматически, но явное указание улучшает читаемость программы и показывает источник возможной ошибки, потери точности и т.п.

Операции

Язык *Си* отличается потрясающим разнообразием операций (называемых также операторами), что обеспечивает ему большую гибкость при программировании. Каждая операция характеризуется уровнем приоритета и порядком, в котором операции исполняются в выражении. Если все операции имеют одинаковый уровень приоритета, выражение вычисляется слева направо, иначе сначала вычисляются операции, обладающие большим уровнем приоритета. Операции описаны в порядке убывания уровня приоритета.

Сводка операций

Приоритет	Группа	Операторы	Порядок
1		`() [] . ->`	слева направо
2	унарные	`* ? - ! ~ ++ -- sizeof`	*справа налево*
3	мультипликативные	`* / %`	слева направо
4	аддитивные	`+ -`	слева направо
5	сдвига	`<< >>`	слева направо
6	отношения	`< > <= >=`	слева направо
7	равенства	`== !=`	слева направо
8	побитовое AND	`&`	слева направо
9	побитовое XOR	`^`	слева направо
10	побитовое OR	`|`	слева направо
11	логическое AND	`&&`	слева направо
12	логическое OR	`||`	слева направо
13	условная	`?:`	*справа налево*
14	присваивания	`= <знак>=`	*справа налево*
15	запятая	`,`	*слева направо*

Операции 1 уровня

Вызов функции ()

Используется при вызове функции всегда, независимо от наличия параметров. Значения функций в выражениях вычисляются первыми.

Операция индексирования []

Предписывает использовать адресную арифметику для вычисления адреса значения. Использование:

адрес[целое_выражение] или целое_выражение[адрес]

Значением выражения является переменная, отстоящая от переменной, заданной параметром [адрес] на [целое_выражение] число переменных. Это значение эквивалентно значению выражения *(адрес + целое_выражение).

Пример:

```
int a[] = { 1, 2, 3, 4, 5 };
a[0] = 6; /* заменяет 1 на 6 */
1[a] = 7; /* заменяет 2 на 7 */
```

Операция выбора компонента структуры

`.` - для выбора компонента статической структуры, например, p.x.

`->` - для выбора компонента структуры через указатель. Если p — указатель на структуру, выбор компонента осуществляется одним из следующих способов: (*p).x или p->x.

Пример:

```
typedef struct {
   int x, y;
} point;        /* структура, описывающая точку */
point a;        /* переменная - точка           */
point *b = &a;  /* указатель на ту же точку     */
a.x = 1;        /* задаем значение компоненты x */
b->y = 2;       /* задаем значение компоненты y */
```

Унарные операции (2 уровень)

Для унарных операций требуется только один операнд. Эти операции либо префиксные, либо постфиксные, либо префиксные и постфиксные. Операция sizeof имеет два варианта: префиксная операция и унарная функция.
Порядок выполнения унарных операций — справа налево.

Косвенная ссылка *

Операнд — указатель на любой тип T, кроме void

Результат — тип T

Использование: *указатель, например:

```
int i = 33;
int *p;           /* описание указателя на целое      */
int j;
int *p = &i;      /* инициализация указателя на целое */
j = *p;           /* разыменовывание                  */
```

Использование указателя на функцию:

```
void fun(void);   /* описание функции                    */
void (*f)(void);  /* описание указателя на функцию       */
f = fun;          /* инициализация указателя на функцию */
(*f)();           /* вызов функции                       */
```

Получение адреса &

Операнд — переменная любого типа T, кроме void

Результат — указатель на тип T

Использование: &переменная, например

```
int i, *p;
p = &i;           /* &i возвращает адрес i */
```

Отрицание —

Операнд — арифметический.

Результат — int, unsigned, long, double

Логическое отрицание !

Операнд — арифметический или указатель.

Результат — int; если операнд равен 1, то результат равен 0 и наоборот.

```
int i = 5;        /* значение i не 0 */
int j = !i;       /* значение j == 0 */
i = !j;           /* значение i == 1 */
```

Дополнение до единицы ~

Операнд — целочисленный.

Результат — int, unsigned, long и равен инверсии целого_выражения.

Нули в битах операнда заменяются на единицы и наоборот (каждый разряд двоичного представления целого значения вычитается из единицы).

```
int i = 5;        /* i = 00000000 00000101 =  5 */
int j = ~i;       /* j = 11111111 11111010 = -6 */
i = ~j;           /* i = 00000000 00000101 =  5 */
```

Увеличение ++ (префиксное)

Операнд — арифметический или указатель.

Результат — int, unsigned, long, double, указатель.

Использование: ++выражение или ++указатель

Результирующим значением является значение выражения после увеличения на единицу.

Значение указателя увеличивается на размер указываемого объекта. Пример:

```
int i[] = { 0, 1, 2 }, *p = &i[1];  /* { 0, 1, 2 } */
i[0] = ++i[1];                      /* { 2, 2, 2 } */
*++p = 5;                           /* { 2, 2, 5 } */
```

Увеличение ++ (постфиксное)

Операнд — арифметический или указатель.

Результат — int, unsigned, long, double, указатель.

Использование: выражение++ или указатель++

Значение результата — значение выражения или указателя, после чего выражение увеличивается на единицу, а указатель увеличивается на размер указываемого объекта. Пример:

```
int i[] = { 0, 1, 2 }, *p = &i[1];  /* { 0, 1, 2 } */
i[0] = i[1]++;                      /* { 1, 2, 2 } */
*p++ = 5;                           /* { 1, 5, 2 } */
```

Уменьшение —— (префиксное)

Операнд — арифметический или указатель.

Результат — int, unsigned, long, double, указатель.

Использование: --выражение или --указатель.
Результирующим значением является значение выражения после уменьшения.

Значение указателя уменьшается на размер указываемого объекта. Пример:

```
int i[] = { 2, 1, 0 }, *p = &i[1];  /* { 2, 1, 0 } */
i[0] = --i[1];                      /* { 0, 0, 0 } */
*--p = 5;                           /* { 5, 0, 0 } */
```

Уменьшение —— (постфиксное)

Операнд — арифметический или указатель

Результат — int, unsigned, long, double, указатель.

Использование: выражение-- или указатель--

Значение результата — значение выражения или указателя, после чего выражение уменьшается на 1, а указатель уменьшается на размер указываемого объекта. Пример:

```
int i[] = { 2, 1, 0 }, *p = &i[1];  /* { 2, 1, 0 } */
i[0] = i[1]--;                      /* { 1, 0, 0 } */
*p-- = 5;                           /* { 1, 5, 0 } */
```

Операция sizeof

Возвращает требуемую для объекта память в байтах.

Операнд — значение любого типа или имени типа.

Результат — unsigned

Использование: sizeof(выражение) или sizeof(имя_типа) или sizeof(имя_объекта). Пример:

```
int i[] = { 0, 0, 0, 0 };          /* { 0, 0, 0, 0 } */
i[0] = sizeof(int);                /* { 2, 0, 0, 0 } */
i[1] = sizeof(i);                  /* { 2, 8, 0, 0 } */
                                   /* размер массива */
i[2] = sizeof(i) / sizeof(i[0]);   /* { 2, 8, 4, 0 } */
i[3] = sizeof(2 + i[0]);           /* { 2, 8, 4, 2 } */
```

Мультипликативные операции (3 уровень)

Порядок выполнения мультипликативных операций — слева направо.

Умножение *

Операнды — арифметические.

Результат — int, unsigned, long, double

Деление /

Операнды — арифметические.

Результат — int, unsigned, long, double

При целочисленном делении дробная часть отбрасывается независимо от знака. Пример:

```
int i[] = { 5, -5, 0, 0 };    /* { 5, -5, 0,  0 } */
i[2] = i[0] / 2;              /* { 5, -5, 2,  0 } */
i[3] = i[1] / 2;              /* { 5, -5, 2, -2 } */
```

Получение остатка %

Операнды — целочисленные.

Результат — int, unsigned, long

Знак результата равен знаку первого операнда.

Целочисленное деление и операция получения остатка связаны соотношением:

```
(a / b) * b + a % b эквивалентно a (если b не равно 0)
```

Пример:

```
int i[] = { 5, 0, 0, 0 };     /* { 5, 0, 0, 0 } */
i[1] = i[0] % 2;              /* { 5, 1, 0, 0 } */
i[2] = i[0] / 2;              /* { 5, 1, 2, 0 } */
i[3] = i[1] + i[2] * 2;       /* { 5, 1, 2, 5 } */
```

Аддитивные операции (4 уровень)

Порядок выполнения аддитивных операций — слева направо.

Сложение + и Вычитание —

Операнды — арифметические

Результат — int, unsigned, long, double

Адресные Сложение + и Вычитание —

Операнды — один указатель, другой целочисленный.

Результат — указатель.

Перед сложением целочисленный операнд умножается на размер указываемого элемента данных. Пример:

```
int a[5] = { 1, 2, 3, 4, 5 };
int *p = a;             /* p указывает на элемент 0    */
int i = *(p + 4);       /* i = 5 — значение элемента 4 */
printf("%d\n", i);      /* i = 5                       */
```

Вычитание адресов —

Операнды — указатели одного типа.

Результат — int (число объектов, отделенных двумя указателями).

Пример:

```
int a[5] = { 1, 2, 3, 4, 5 };
int *p0 = a;            /* p0 указывает на элемент 0    */
int *p4 = a + 4;        /* p4 указывает на элемент 4    */
int i = p4 - p0;        /* между указателями 4 элемента */
printf("%d\n", i);      /* i = 4                        */
```

Операции сдвига (5 уровень)

Порядок выполнения операций сдвига — слева направо.

Сдвиг влево <<

Операнд — целочисленный.

Результат — такой же, как у левого (сдвигаемого) операнда.

Правый операнд преобразуется к типу int; левый операнд сдвигается влево на число разрядов, равное значению правого операнда; новые разряды заполняются нулями. Сдвиг влево на один разряд умножает операнд на два и поэтому используется для быстрого умножения (операция целочисленного умножения выполняется примерно в 35 раз медленнее, чем сдвиг). Однако умножение на два произойдет только в случае, если есть, куда сдвигать старший разряд, например, в следующей программе результат первого сдвига 254, а второго — 252:

```
unsigned char i = 127;      /* 127 = 0111 1111 */
i = i << 1;                 /* 254 = 1111 1110 */
i = i << 1;                 /* 252 = 1111 1100 */
```

Сдвиг вправо >>

Операнд — целочисленный.

Результат — такой же, как у левого (сдвигаемого) операнда.

Правый операнд преобразуется к типу int; левый операнд сдвигается вправо на число разрядов, равное значению правого операнда; новые разряды заполняются нулями, если тип левого операнда unsigned, иначе значением знакового разряда:

```
char j, k;
unsigned char i = 128;      /*  128 = 1000 0000 */
i = i >> 1;                 /*   64 = 0100 0000 */
i = i >> 1;                 /*   32 = 0010 0000 */
j = -128;                   /* -128 = 1000 0000 */
j = j >> 1;                 /*  -64 = 1100 0000 */
j = j >> 1;                 /*  -32 = 1110 0000 */
k = 127;                    /*  127 = 0111 1111 */
k = k >> 1;                 /*  -64 = 0011 1111 */
k = k >> 1;                 /*  -32 = 0001 1111 */
```

Сдвиг вправо на один разряд выполняет быстрое целочисленное деление на два.

Операции отношения (6 уровень)

Порядок выполнения операций отношения — слева направо.

Меньше < и Больше >

Операнды — арифметические или указатели.

Результат — int

Пример — переменная k получает значение переменной j:

```
int i = 2, j = 1, k;
if (i < j) {
   k = i;
} else {
   k = j;
}
```

Меньше или равно <= и Больше или равно >=

Операнды — арифметические или указатели.

Результат — int

В случае сравнения указателей определяется относительное расположение объектов в памяти.

Пример — переменная k получает значение переменной i:

```
int i = 1, j = 1, k;
if (i <= j) {
   k = i;
} else {
   k = j;
}
```

Пример с использованием указателей — оба указателя приводятся к одному значению:

```
int i = 1, j = 2, *a = &i, *b = &j;
if (a < b) {
   a++;
} else {
   b--;
}
```

Операции равенства и неравенства (7 уровень)

Порядок выполнения операций равенства и неравенства — слева направо.

Равенство == и неравенство !=

Операнды — арифметические или указатели.

Результат — int

В случае сравнения указателей они сравниваются только с одним целочисленным значением — 0.

```
int i, j, k, m, n, a = 1, b = 2, *c = &a, *d = &b;
i = a == b;                 /* i == 0 */
j = a != b;                 /* j == 1 */
k = a != 0;                 /* k == 1 */
k = a != 1;                 /* k == 0 */
m = c == d;                 /* m == 0 */
n = c != d;                 /* n == 1 */
```

Примечание: оператор вида i = a == b; следует понимать так:

Вычислить значение выражения a == b и присвоить полученное значение переменной i. Так как в выражении используется операция отношения, результат имеет логическое значение (0 или 1).

Операции с битами (поразрядные)

Эта группа операций рассматривает операнды и результат как значения, разряды которых независимы друг от друга. Возвращаемое значение — измененные (или неизмененные) биты одного из операндов, которые определяются вторым операндом — маской. Маска содержит единицы только в тех разрядах, биты которых предполагается изменить (или проверить). Обратная маска является инверсией обычной маски — нули заменяются на единицы и наоборот. Биты в операндах нумеруются справа налево от нуля:

X	X	X	X	X	X	X	X
7	6	5	4	3	2	1	0

номер бита

Для вычисления результата используются таблицы истинности, которые следует применить к каждой паре разрядов обоих операндов для получения всех разрядов результирующего значения.

Операция & (поразрядное И, 8 уровень)

Порядок выполнения операции — слева направо.

Операнды — целочисленные

Результат — int, long, unsigned

Назначение: проверка бита и очистка бита. Для проверки бита номер n в операнде a следует использовать второй операнд — маску m, значение которой равно весу бита, т.е. 2^n. Для очистки бита следует использовать обратную маску (формула очистки операнд И НЕ).

Операция коммутативна (операнды можно менять местами).

Результат проверки битов:

- равен маске, если все проверяемые биты установлены;
- не равен нулю, если хотя бы один из проверяемых битов установлен;
- равен нулю, если ни один из проверяемых битов не установлен.

Таблица истинности:

&	0	1
0	0	0
1	0	1

Пример проверки битов 0 и 2:

```
char a = 8,       /* 0000 1000 */
     b = 12,      /* 0000 1100 */
     c = 13,      /* 0000 1101 */
     m = 5,       /* 0000 0101 */
     t;
t = a & m; /* равно нулю    - ни один из битов не установлен */
t = b & m; /* не равно нулю - какой-то из битов установлен   */
t = c & m; /* равно маске   - все биты установлены           */
```

Пример проверки бита 0 (проверка числа на четность):

```
char i = 2,       /* 0000 0010                        */
     j = 3,       /* 0000 0011                        */
     a, b;        /* 0000 0001 - маска нулевого бита */
a = i & 1;        /* результат 0 - число четное       */
b = j & 1;        /* результат 1 - число нечетное     */
```

Пример очистки бита 5 (перевод буквы в верхний регистр — операнд И НЕ 32):

```
char a = 'A',     /* a ==  0100 0001  == 'A'          */
     b = 'a';     /* b ==  0110 0001  == 'a'           */
                  /* маска 0010 0000  == 32 == пробел */
a &= ~32;         /* a ==  0100 0001  == 'A'          */
b &= ~32;         /* b ==  0100 0001  == 'A'          */
```

Операция ^ (поразрядное исключающее ИЛИ, 9 уровень)

Порядок выполнения операции — слева направо.

Операнды — целочисленные

Результат — int, long, unsigned

Назначение: переключение бита. Для переключения бита номер n в операнде a следует использовать второй операнд — маску m, значение которой равно весу бита, т.е. 2^n. Операция коммутативна (операнды можно менять местами) и идемпотентна — применение одной и той же маски дважды возвращает начальное значение.

Результат: биты, соответствующие единицам в маске, меняют значение на противоположное.

Таблица истинности:

^	0	1
0	0	1
1	1	0

Пример:

```
char a = 12,      /* 0000 1100 == 12 */
     m = 5;       /* 0000 0101 ==  5 */
a = a ^ m;        /* 0000 1001 ==  9 */
a = a ^ m;        /* 0000 1100 == 12 */
```

Операция | (поразрядное ИЛИ, 10 уровень)

Порядок выполнения операции — слева направо.

Операнды — целочисленные

Результат — int, long, unsigned

Назначение: установка бита. Для установки бита номер n в операнде a следует использовать второй операнд — маску m, значение которой равно весу бита, т.е. 2^n. Операция коммутативна.

Таблица истинности:

|	0	1
0	0	1
1	1	1

Пример установки бита 5 (перевод буквы в нижний регистр — ИЛИ 32):

```
char a = 'A',     /* a ==  0100 0001  == 'A'          */
     b = 'a';     /* b ==  0110 0001  == 'a'          */
                  /* маска 0010 0000  == 32 == пробел */
a |= 32;          /* a ==  0110 0001  == 'a'          */
b |= 32;          /* b ==  0110 0001  == 'a'          */
```

Логические операции

Эти операции вырабатывают значение, которое может быть использовано для принятия решения в операторах управления. Фактически возвращаемым значением является тип int, который рассматривается как логическое значение. Если

результат операции равен 0, значение считается ложным, иначе значение истинно. Для вычисления результата используются таблицы истинности, которые применяются один раз к операндам в целом.

Операция && (логическое И, 11 уровень)

Порядок выполнения операции — слева направо.

Операнды — арифметические или указатели.

Результат — int

Если первый операнд равен 0, то результат равен 0;

Если первый операнд не равен 0, то результат равен 0, если второй операнд 0, иначе равен 1.

Таблица истинности:

&&	0	не 0
0	0	0
не 0	0	1

Пример — цикл для поиска точки в строке знаков:

```
char s[] = "Hello.txt";
int i = 0;
while (s[i] != '.' && s[i]) i++;
```

Перед очередной итерацией вычисляются значения выражений s[i] != '.' и s[i] и полученные значения логически умножаются.

Для нулевого элемента массива s первое множимое в условии завершения цикла равно 1, поэтому проверяется второе множимое, которое не равно нулю (равно коду символа). Согласно таблице истинности, общий результат равен 1, и выполняется итерация.

Для пятого элемента массива s первое множимое равно 0, поэтому результат сразу принимается равным нулю и цикл завершается. Переменная i указывает на точку.

Если строка не содержит точки, цикл завершается при обнаружении нулевого знака — признака конца строки. В этом случае первое множимое равно 1, второе множимое равно 0, общий результат 0 и цикл завершается. Переменная i указывает на конец строки.

Операция || (логическое ИЛИ, 12 уровень)

Порядок выполнения операции — слева направо.

Операнды — арифметические или указатели.

Результат — int

Если первый операнд не равен 0, результат равен 1;

Если первый операнд равен 0, то результат равен 1, если второй операнд не 0, иначе равен 0.

Таблица истинности:

||	0	не 0
0	0	1
не 0	1	1

Пример — цикл для поиска точки в строке знаков:

```
char s[] = "Hello.txt";
int i = 0;
while (!(s[i] == '.' || s[i] == 0)) i++;
```

Перед очередной итерацией вычисляются значения выражений s[i] == '.' и s[i] == 0, полученные значения логически складываются и отрицаются.

Для нулевого элемента массива s первое слагаемое равно 0, второе слагаемое равно 0, результат сложения 0, результат отрицания 1, выполняется итерация.

Для пятого элемента массива s первое слагаемое равно 1, результат сложения принимается равным 1, после отрицания получаем 0 и цикл завершается. Переменная i указывает на точку.

Если строка не содержит точки, цикл завершается, когда переменная i указывает на конец строки и второе слагаемое становится равным 1, а результат отрицания 0.

Условная операция ?: (13 уровень)

Порядок выполнения операции — слева направо.

Операнды — арифметические; второй и третий операнды могут быть указателями, структурами, объединениями. Второй и третий операнды приводятся к одному и тому же типу.

Результат — int, long, unsigned, double, указатель, структура или объединение.

У этой операции 3 операнда. Использование: a ? b : c;
a, b, c — выражения. Если a не равно 0, результат равен b, иначе c.

Пример вычисляет максимальное значение из значений двух переменных:

```
max = (x > y) ? x : y;
```

Следующий пример вызывает одну из двух функций в зависимости от значений переменных x и y (функцию f1()). Результат, возвращаемый операцией — void:

```
void f1(void) {}
void f2(void) {}

void main(void) {
   int x = 1, y = 1;
   x == y ? f1() : f2();
}
```

Операции присваивания (14 уровень)

Порядок выполнения операции — справа налево.

Простое присваивание =

Операнды — арифметические, указатели, структуры или объединения.

Результат — если оба операнда арифметические, значение правого операнда преобразуется к типу левого. Операция присваивания в языке *Си* возвращает результат, поэтому имеет смысл оператор

```
a = b = c;
```

который вычисляется справа налево:

```
(a = (b = c));
```

Сложное присваивание <знак>=

Оператор

```
v <знак>= e;
```

где v — переменная, e — выражение, примерно эквивалентен оператору

```
v = v <знак> e;
```

Разница заключается в том, что в операторе v <знак>= e; операнд v вычисляется только один раз, а в операторе v = v <знак> e; — дважды.

Примеры сложных присваиваний:

```
int i = 0, j = 0;
i += 1;      /* i ==  1 */
j -= i;      /* j == -1 */
i <<= 1;     /* i ==  2 */
```

В качестве повода для размышлений рассмотрим оператор

```
a[i++] += n;
```

при выполнении которого мы получаем дополнительный эффект — увеличение i. Очевидно, что это присваивание не эквивалентно оператору

```
a[i++] = a[i++] + n;
```

в котором операция ++ выполняется дважды, а результат зависит от порядка вычислений конкретным компилятором.

Допустимые знаки в сложном присваивании — следующие десять:

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

Операция запятая , (15 уровень)

Порядок выполнения операции — слева направо.

Тип результата совпадает с типом правого операнда.

Объединяет два выражения в одно, значением которого является значение правого операнда; значение левого операнда используется для получения побочного эффекта. В контексте, где запятая используется для других целей, например для отделения аргументов функции, выражения с операцией запятая следует заключать в скобки. В примере вторая половина массива переносится в первую в обратном порядке:

```
void main(void) {
   int i, j, a[]= { 1, 2, 3, 4, 5 };
   for (i = 0, j = 4; i < j; i++, j--) {
      a[i] = a[j];
   }
}
```

Операторы (управления)

Основные операторы языка (операторы управления ходом вычислений) принципиально не отличаются от аналогичных операторов в других языках, таких, как Паскаль. Эти операторы соответствуют принципам структурного программирования.

Для облегчения построения циклов в языке *Си* есть операторы перехода continue и break, которые в других языках могут быть получены только с использованием оператора goto.

Выражения

Выражение — это конструкция, составленная с использованием операций, литералов, констант, переменных и вызовов функций. В отличие от других языков, в *Си* компилятор может переупорядочивать выражения даже при наличии скобок, поэтому для получения определенного порядка вычислений следует использовать отдельные операции присваивания.

Постоянные выражения

Выражение, полученное с использованием константных типов int, char и enum, операции sizeof, унарных операторов – и ~, бинарных операций

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

и тернарного оператора ?:, называется *постоянным* (константным).
Постоянные выражения используются в операторе switch, в инициализаторах границ массивов и операторе препроцессора #if (в котором к тому же не допускаются sizeof и перечисляемые константы).

Выражения и операторы

Любое выражение может быть преобразовано в оператор добавлением к нему точки с запятой. Запись вида

```
выражение;
```

является оператором. Значение выражения в этом случае игнорируется и используется для получения побочного эффекта.

Пустой оператор ;

Пустой оператор обозначается точкой с запятой:

```
;
```

Он используется в случае, если по синтаксису требуется оператор, но никаких действий по смыслу алгоритма выполнять не требуется. Например, пустой оператор составляет тело цикла while в следующем случае:

```
while (( c = getch()) != '\n' ) ;
```

Составной оператор {}

Называется также блоком. Используется:

- В качестве тела функции;
- Для группировки нескольких операторов в один в случаях, например, когда по синтаксису требуется один оператор, а по смыслу алгоритма — несколько;
- Для локализации действия описаний.

Составной оператор представляет собой фигурные скобки {}, внутри которых располагаются описания и определения и следующие за ними операторы:

```
{
   [определения и описания]
   [операторы]
}
```

В следующем примере блок используется для локализации вычислений:

```
void main(void) {
   int i = 1, j = 3;
   {
      int k = i;
      i = j;
      j = k;
   }
   printf("%d\n%d\n", i, j);
}
```

Переменная k не видна вне блока, а переменные i и j внутри блока видны. Блок меняет местами значения переменных i и j.

Оператор присваивания

Изменяет значение объекта, находящегося слева от знака присваивания равно, на значение выражения, находящегося справа от знака равно. Оператор присваивания имеет вид

```
переменная = выражение;
```

и получается из операции присваивания добавлением точки с запятой:

```
переменная = выражение
```

Отличительная особенность операции присваивания в языке *Си* — значением операции является значение выражения (операция возвращает значение). Поэтому в одном операторе присваивания можно объединять несколько операций присваивания, например:

```
i = j = k = 0;
```

Интересный случай возникает, если использовать оператор вида:

```
i = j = (k = 0) + 1;
```

Оператор if

Назначение условного оператора — ветвление алгоритма. При его выполнении сначала вычисляется выражение. Если выражение истинно, выполняется оператор, находящийся непосредственно после выражения. Если выражение ложно и оператор не содержит конструкции else, выполнение заканчивается. Если конструкция else есть, то выполняется оператор, непосредственно следующий за else.
Форма 1:

```
if (выражение)
   оператор1
```

Форма 2:

```
if (выражение)
   оператор1
else
   оператор2
```

Распространенная ошибка — попытка выполнить ветвь, состоящую из нескольких операторов, не заключенных в блок. По синтаксису ветвь оператора может содержать только один оператор. Если по смыслу алгоритма требуется несколько, следует использовать составной оператор (блок).
Использование оператора if без блоков может привести к неоднозначности («проблеме висящего else»). Например, оператор

```
if (выражение1) if (выражение2) оператор1 else оператор2
```

может быть интерпретирован двумя разными способами:

```
if (выражение1)
   if (выражение2)
      оператор1
   else
      оператор2
```

или:

```
if (выражение1)
   if (выражение2) оператор1
else
   оператор2
```

Для разрешения этой неоднозначности используется правило: часть else оператора всегда относится к самому правому оператору if, не имеющему else.

По этому правилу верна первая интерпретация. Чтобы избежать подобной неоднозначности, рекомендуется вместо одиночных операторов использовать составные:

```
if (выражение1) {
   if (выражение2) {
      операторы
   } else {
      операторы
   }
}
```

Следующая запись может оказаться более понятной:

```
if (выражение1) {
   if (выражение2) оператор1 else оператор2
}
```

Пример оператора:

```
if (a > b) {
   /* a больше */
   c = a;
} else {
   /* b больше */
   c = b;
}
```

Оператор switch

Используется для ветвления алгоритма по нескольким направлениям.

Оператор switch имеет следующую форму:

```
switch (выражение) {
   case (постоянное_выражение1) : оператор1 break;
   case (постоянное_выражение2) : оператор2 break;
   . . .
   case (постоянное_выражениеN) : операторN break;
   default: операторN+1
}
```

Выражение после switch должно быть целым или приводимым к целому.

Альтернативы case не должны повторяться.

Оператор switch выполняет ту альтернативу case, постоянное выражение которой совпадает со значением выражения switch. Если значение выражения switch не совпадает ни с одним из постоянных выражений альтернатив, выполняется альтернатива default, наличие которой не обязательно.
Число операторов альтернативы может быть больше одного или равным нулю.

Если действие двух или более альтернатив совпадает, их можно объединять:

```
switch (выражение) {
   case (пост_выр1) : case (пост_выр2) : оператор1 break;
   . . .
   default: операторN+1
}
```

Пример оператора:

```
switch (c) {
   case '+': add(a, b); break;
   case '-': sub(a, b); break;
   case '*': mul(a, b); break;
   case '/': div(a, b); break;
   default: puterror("Invalid operator.");
}
```

Оператор цикла for

Цикл с параметром. Используется в случаях, когда количество итераций известно (и может быть равным нулю). Параметр используется для счета итераций.

Синтаксис:

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

Выражение1 задает начальные условия выполнения цикла.

Выражение2 проверяет условие продолжения цикла.

Выражение3 модифицирует условия, заданные в выражение1.

Любое из этих выражений может быть опущено. Если опущено выражение2, вместо него подставляется значение «истина». Наличие двух знаков точка с запятой обязательно.

Основное правило использования параметрического цикла — нельзя модифицировать параметр операторами внутри тела цикла. С другой стороны, можно использовать параметр цикла как операнд в выражениях:

```
void main(void) {
   int i = 0;
   for (; i < 5; i++) {

      printf("%d\n", 1 + i);
   }
}
```

Эта программа выводит значения от 1 до 5.

Оператор цикла while

Цикл с предусловием. Используется в случаях, когда число итераций неизвестно и может быть равным нулю. Это универсальный цикл — с его помощью можно построить любой другой цикл.

Синтаксис:

```
while (выражение) оператор
```

Оператор выполняется до тех пор, пока значение выражения равно «истина».

Значение выражения вычисляется перед каждым выполнением оператора.

Пример оператора:

```
char s[] = "String literal";
int i = 0;
while (s[i]) {
   s[i++] |= 32;  /* перевод в нижний регистр */
}
```

Оператор цикла do

Цикл с постусловием. Имеет следующую форму:

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

Оператор выполняется до тех пор, пока выражение равно значению «истина». В отличие от цикла while, оператор в этом цикле будет исполнен минимум один раз. Пример оператора (вычисляется сумма элементов массива):

```
char a[] = { 1, 2, 3, 4, 5 };
int i = 0, s = 0;
do {
   s += a[i++];
}
while (i < sizeof(a) / sizeof(a[0]));
```

Оператор break

Имеет вид:

```
break;
```

Используется для выхода из цикла любого типа и из оператора switch. Управление передается на оператор, следующий за оператором, из которого осуществлен выход.

Пример оператора (вычисляется положение точки в строке):

```
char s[] = "file.txt";
int i = 0;
while (s[i]) {
   if (s[i++] == '.' break; /* если точка - выход */
}
```

Оператор continue

Имеет вид:

```
continue;
```

Вызывает переход к следующей итерации цикла (пропуск операторов, расположенных ниже оператора continue в теле цикла).

Пример оператора (вычисляется количество положительных элементов массива):

```
char a[] = { 1, -2, 3, -4, 5 };
int i = 0, n = 0;
do {
   /* если элемент массива меньше нуля - дальше */
   if (a[i++] < 0) continue;
   n++;
}
while (i < sizeof(a) / sizeof(a[0]));
```

Оператор return

Используется для завершения работы функции и выхода из нее. Если функция возвращает значение, это значение указывается в операторе return:

```
return выражение;
```

Если функция не возвращает значения, оператор return имеет вид:

```
return;
```

Функция может иметь несколько операторов возврата и не иметь их совсем, если возвращаемый тип функции объявлен как void.

Функции

Функция является одним из основных объектов языка *Си*, обеспечивающая абстракцию управления (независимость от реализации). Использование функций улучшает читаемость программы и помогает писать легко отлаживаемый и модифицируемый код.

Функции в *Си* могут быть рекурсивными (могут вызывать сами себя). Функции могут возвращать значение и могут не возвращать. Во втором случае они соответствуют процедурам в других языках программирования.

Функции могут иметь параметры (аргументы) и могут не иметь их.

Функции используются посредством операции вызова функции, представляющей собой имя функции и обязательные круглые скобки, внутри которых указываются фактические параметры, если определены формальные.

Функция, как и любой другой объект, должна быть описана до первого вызова в программе. Описание функции сообщает компилятору тип возвращаемого значения и типы аргументов, если таковые есть. Определение функции, кроме того, содержит код, который должен быть исполнен при вызове функции. Тело функции (код) заключается в фигурные скобки (составной оператор).

- Функция завершает свою работу в двух случаях: при достижении во время исполнения закрывающей скобки составного оператора, ограничивающего тело функции и при выполнении оператора return.
- В случае, если функция возвращает значение, она должна содержать как минимум один оператор return выражение.
- Функция может содержать любое количество операторов return.
- Функция не может возвращать в качестве своего результата массив или функцию, но может возвращать указатель на массив или на функцию.

Определение функции, возвращающей значение, имеет вид:

```
[static] тип имя_функции(список_формальных_параметров)
описания формальных параметров
{
   [определения и описания]
   [операторы]
   return выражение;
}
```

Для функции, не возвращающей значение, тип_результата должен быть void:

```
[static] void имя_функции(список_формальных_параметров)
описания формальных параметров
{
   [определения и описания]
   [операторы]
}
```

Класс памяти static ограничивает область видимости функции модулем, в котором функция определена. Во всех других случаях функция видна во всех модулях программы.

Пример функции, возвращающей максимум из двух параметров:

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

void main(void) {
   int i = 3, j = 5;
   printf("%d\n", max(i, j));
}
```

Современные компиляторы используют определение функции без описания формальных параметров отдельной строкой:

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

Описание функции отличается от определения отсутствием тела функции (включая скобки) и необязательностью указания имен формальных параметров. Описание функции называют также ее прототипом:

```
/* прототип функции max */
int max(int, int);
void main(void) {
   int i = 3, j = 5;
   printf("%d\n", max(i, j));
}
/* определение функции max */
int max(int a, int b) {
   return (a > b) ? a : b;
}
```

Типы параметров в описании (прототипе) функции по синтаксису языка *Си* также не нужны, однако компилятор *Borland* C++ воспринимает такое строгое следование правилам с неохотой. Единственный класс памяти, допустимый для формальных параметров функции — регистровый.

Параметры функции

Различают формальные и фактические параметры функций. Формальные параметры указываются в описании (определении) функции для того, чтобы можно было использовать внешние данные внутри функции через имена формальных параметров.

При вызове функции ей передаются фактические параметры посредством указания имен переменных, внешних по отношению к функции, например:

```
int max(int a, int b) {
   return (a > b) ? a : b;
}
void main(void) {
   int i = 3, j = 5;
   printf("%d\n", max(i, j));
}
```

В этом примере при вызове функции max ей передаются фактические параметры через переменные i и j. Формальный параметр a принимает значение фактического параметра i, а формальный параметр b — значение фактического параметра j.

В качестве фактических параметров могут быть использованы не только переменные, но и выражения, литералы и константы, например:

```
#define KON 12
. . .
printf("%d\n", max(i + 9, KON));
```

Параметры функции передаются по значению, что означает, что тело функции получает копии параметров. В других языках программирования параметры могут передаваться по ссылке. Передача по ссылке означает, что в тело функции передается сам объект (указатель на него), который может быть изменен операторами тела функции. В языке *Си* для изменения объекта внутри функции нужно использовать в качестве параметра не объект, а указатель на него.

Рассмотрим простую функцию, которая меняет местами значения двух объектов:

```
void swap(int a, int b) {
   int c = a; a = b; b = c;
}

void main(void) {
   int i = 3, j = 5;
   swap(i, j);
   printf("%d\n%d\n", i, j);
}
```

Поскольку внутрь функции передаются копии объектов, изменение копий никак не сказывается на оригиналах объектов, и функция не выполняет своего предназначения. Используем другой вариант:

```
void swap(int *a, int *b) {
   int c = *a; *a = *b; *b = c;
}

void main(void) {
   int i = 3, j = 5;
   swap(&i, &j);
   printf("%d\n%d\n", i, j);
}
```

Теперь в качестве фактических параметров передаются адреса переменных, которые внутри функции используются как указатели на оригиналы объектов и операторы тела функции изменяют объекты, расположенные вне функции, в соответствии с предназначением.

При передаче параметров производятся автоматические преобразования:

- фактические параметры типа float преобразуются к типу double;
- фактические параметры типов char и short int преобразуются к типу int;
- фактический параметр — имя массива копируется в формальный параметр — указатель на массив.

В описании формальных параметров — массивов может быть опущено количество элементов первого измерения массива, например:

```
int a[], int b[][4]
```

При передаче функции массива в качестве параметра теряется информация о его размере, поэтому внутри функции нельзя использовать операцию sizeof для определения числа элементов массива через выражение sizeof(*имя_массива*) / sizeof(*тип_элемента*).

Передача функций в качестве параметров

Для передачи функции в качестве параметра фактический параметр должен быть указателем на функцию, а формальный — описанием указателя на функцию. Например, в следующей программе используется функция для пузырьковой сортировки массива целых чисел. Первый параметр этой функции — количество элементов массива, второй — указатель на массив, третий — функция сравнения элементов:

```
int greater(int, int);
int smaller(int, int);
void sort(int n, int a[], int (*cmp)(int, int)) {
   int i, j;
   for (i = 0 ; i < n - 1 ; i++) {
      for (j = i + 1 ; j < n ; j++) {
         if ((*cmp)(a[i], a[j])) {
            int k = a[i]; a[i] = a[j]; a[j] = k;
         }
      }
   }
}
void main(void) {
   int i, n, m[] = { 2, 5, 4, 3, 1 };
   n = sizeof(m) / sizeof(int);
   sort(n, m, greater);
   for (i = 0 ; i < n ; i++) printf("%d\n", m[i]);
}
```

В программе должны быть также определены функции сравнения:

```
int greater(int a, int b) {
   return a > b;
}

int smaller(int a, int b) {
   return a < b;
}
```

Передача той или иной функции сравнения в функцию сортировки обеспечивает сортировку в возрастающем или убывающем порядке.

Указатель в качестве результата функции

Если функция возвращает указатель, он не должен указывать на объект, определенный внутри функции. Такой объект имеет автоматический класс памяти и уничтожается по завершении работы функции, так что ссылка на него может привести к непредсказуемым результатам. Следующий код неправилен:

```
int * f(void) {
    int i = 1;
    return &i;
}
```

Если функция должна возвращать указатель на массив, указатель следует выделить из динамической памяти или использовать передачу указателя через параметры функции.

Функция main

Функция с именем main используется в качестве главной функции программы. После исполнения кода инициализации (*startup code*) управление передается функции main, которая описывает весь предполагаемый алгоритм программы, используя вызовы других функций, определенных как в модулях программы, так и в стандартных или пользовательских библиотеках.

Функция main ничем не отличается от других функций за исключением того, что возвращаемое значение этой функции используется операционной системой как код возврата, а в качестве параметров функции могут выступать только аргументы командной строки, при помощи которой была запущена программа. Кроме того, эту функцию нельзя вызывать явно.

Если функция main возвращает значение, его тип должен быть int, а значение должно соответствовать принятым соглашениям о коде возврата. Так, возвращаемое значение `0` сообщает об успешном завершении работы программы, другие значения указывают на ту или иную ошибку, обнаруженную алгоритмом в ходе выполнения.

Параметры командной строки передаются функции main следующим образом. Первым параметром целого типа указывается количество аргументов командной строки с учетом того, что сама команда входит в число параметров. Следующий параметр является двойной ссылкой на знаковый тип или, иначе, указателем на массив строковых значений. Каждая строка в этом массиве представляет собой один отдельный аргумент — параметр командной строки, начиная с команды, индекс которой в массиве аргументов равен 0.

Например, если для запуска программы использовалась команда

```
test a.txt 2
```

массив аргументов будет состоять из строк (первая строка определяется расположением программы):

```
c:\bc\test.exe
a.txt
2
```

Преобразование третьего аргумента из строкового представления в числовое в этом случае является заботой программиста.

Пример основной функции, принимающей аргументы командной строки:

```
int main(int narg, char *arg[]) {
   if (narg == 1) {
      printf("Мало аргументов\n");
      return 1;
   } else {
      return main_process(arg);
   }
}
```

Неопределенное число параметров

Функции в языке *Си* могут принимать неопределенное число параметров. При описании такой функции список ее формальных параметров завершают многоточием, что означает «и еще какие-то параметры».

Для доступа к неспецифицированным параметрам следует использовать макросы, определенные в файле stdarg.h. Ниже приводится пример:

```
#include <stdio.h>
#include <stdarg.h>
void intsum(int *s, ...) {
   int arg;
   va_list list;
   va_start(list, s);
   while ((arg = va_arg(list, int)) != 0) *s += arg;
   va_end(list);
}

void main(void) {
   int sum = 0;
   intsum(&sum, 1, 2, 3, 4, 5, 0);
   printf("%d\n", sum);
}
```

Функция intsum подсчитывает сумму неопределенного числа аргументов в предположении, что они имеют тип int. Первый параметр этой функции — это возвращаемая сумма. Сначала определяется и инициализируется переменная list — массив аргументов. Макрос va_start использует последний известный параметр функции для отсчета последующих неизвестных. Макрос va_arg выбирает очередной параметр из массива, полагаясь на информацию о типе параметра, указываемую программистом. Нет никакого способа убедиться в том, что фактический параметр был передан при вызове функции, поэтому следует предусмотреть какой-либо механизм для определения количества параметров. В данном примере значение аргумента 0 является сигналом о том, что это последний аргумент.

Перед завершением функция вызывает макрос va_end, который аннулирует изменения стека корректным образом. Этот макрос должен вызываться в случае, если был использован макрос va_start.

Препроцессор

Каждый файл на языке *Си* перед компиляцией подвергается дополнительной обработке специальной программой, называемой препроцессором языка *Си*. В составе пакета *Borland* C++ препроцессором является файл cpp.exe.

Препроцессор выполняет три основные задачи:
1. макроподстановку;
2. включение текстов одних файлов в другие;
3. условную компиляцию.

Макроподстановка

Макроподстановка — это замена одних знаковых последовательностей другими. Основное назначение макроподстановки — определение констант.

Для выполнения макроподстановки ее следует определить при помощи директивы #define:

```
   #define идентификатор подстановка
```

Директива #define связывает имя идентификатор с последовательностью знаков подстановка. При обработке текста на языке *Си* все последовательности знаков, совпадающие с идентификатор, заменяются последовательностями подстановка.

Последовательность идентификатор не может содержать пробелы, а подстановка — может.

В качестве подстановки используются не только литералы, но и постоянные выражения, а также любые текстовые конструкции, которые могут быть интерпретированы правильным образом во время компиляции. Например, в следующем примере определяется идентификатор END_OF_OP, который заменяется знаком ; (конец оператора):

```
#define END_OF_OP ;
void main(void) {
    printf("Hello\n")END_OF_OP
}
```

Это, разумеется, экзотическая интерпретация, но пример демонстрирует суть подстановок. Программа абсолютно корректна, хотя читать ее не совсем привычно.

Макроподстановки могут принимать параметры для того, чтобы текст подстановки изменялся в соответствии с нуждами программиста:

```
#define AREA(a, b) (a) * (b)

void main(void) {
    printf("%d\n", AREA(2, 3));
}
```

В этом примере определена подстановка с двумя параметрами. Параметры идентификатора в тексте подстановки должны заключаться в скобки. Компилятор *Borland* C++ в большинстве случаев правильно выполнит подстановку и без скобок, но это особенность компилятора. Фактическая подстановка имеет вид (2) * (3). Если изменить текст программы следующим образом:

```
i = AREA(2 + 4, 3);
```

текст подстановки примет вид (2 + 4) * (3).

Часто в качестве примера макроподстановки с параметрами используется описание функции, вычисляющей максимум (или минимум):

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
void main(void) {
    int i;
    i = max(2, 3);
    printf("%d\n", i);
}
```

Текст подстановки ((2) > (3)) ? (2) : (3).

Включение файлов

Директива препроцессора #include предписывает включить в обрабатываемый текст другого файла, имя которого указывается как параметр директивы:

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

Если имя файла заключено в угловые скобки, файл должен быть расположен в каталоге, указанном в параметрах настройки среды *Borland* C++ как каталог включаемых файлов. В большинстве случаев включаемые файлы представляют собой заголовочные файлы, хотя это совсем на обязательно.

Если имя файла заключено в двойные кавычки, указанный файл ищется в текущем каталоге. Для ссылки на файл, расположенный не в текущем каталоге, вместо имени файла можно использовать его абсолютную или относительную спецификацию.

Например, директива

```
#include "myinc\myio.h"
```

предписывает включить файл myio.h, расположенный в каталоге myinc, который находится в текущем (относительная спецификация).

Текст включаемого файла заменяет собой строку директивы включения.

Чаще всего директива #include используется для включения в текст заголовочных файлов, описывающих стандартные и пользовательские библиотеки и определяющие константы, символы, имена типов.

Условная компиляция

Под условной понимается компиляция только той части текста модуля, которая указана при помощи директив условной компиляции. Это требуется для того, чтобы скомпилировать тот или иной код в зависимости от условий, в которых должна использоваться результирующая программа, например, чтобы получить код, который работает в той или иной операционной системе.

Для условной компиляции используется условный оператор препроцессора:

```
#if условие
#else
#endif
```

В качестве условия используются постоянное выражение, например:

```
#if ABC + 3
```

Если постоянное выражение не равно нулю (истина), то часть текста, расположенная между #else и #endif, игнорируется.

Другие варианты условного оператора проверяют наличие определенных идентификаторов. Идентификатор существует, если он определен при помощи директивы #define:

```
#ifdef идентификатор
#else
#endif
```

Например:

```
#ifdef ABC
```

Условие имеет значение истина, если идентификатор определен ранее директивой #define, иначе условие ложно.

```
#ifndef идентификатор
#else
#endif
```

Например:

```
#ifndef ABC
```

Условие имеет значение истина, если идентификатор не определен ранее директивой #define, иначе ложь.

Структура программы

Основным элементом структуры программы является функция.

Функция — это небольшая законченная часть программы, имеющая вполне определенное назначение и отличающаяся от остальных функций своим именем (в одной программе не может быть двух функций с одинаковыми именами).

Функция предназначена для достижения вполне конкретной цели. Цель при этом определяет программист, ее написавший. И только от него зависит, как и какую цель достигает конкретная функция, а в целом — как и какую цель достигает программа.

Функция является инструментом абстракции управления. Это означает, что выделяя часть кода (часть алгоритма) в функцию, мы в дальнейшем абстрагируемся от ее содержания, воспринимая функцию как *черный ящик*, имеющий вход и выход. Имеет значение только то, что требуется подать на вход и что при этом мы получим на выходе. Выделение функциональных компонентов программы облегчает ее понимание и, соответственно, отладку и модификацию.

Поскольку функция абстрагирует нас от своего кода, этот код становится самостоятельной частью и может рассматриваться как самостоятельная задача программирования, более простая, чем программа в целом. Написание программы в целом может быть разбито на отдельные фрагменты, которые могут быть выполнены различными программистами.

Другим структурным компонентом программы является модуль. Модуль представляет собой файл проекта, в который включены все указанные при помощи директивы препроцессора #include [другие] файлы. Модуль также служит цели разбиения программы на отдельные законченные фрагменты, обладающие меньшей сложностью, чем программа в целом. Модуль обычно объединяет несколько функций, имеющих одну направленность, решающих одну задачу. Кроме того, модуль служит для сокрытия реализации, так как код модуля может быть скрыт от посторонних за счет использования объектного кода или библиотеки, построенной на основе этого модуля (или нескольких модулей).

Разделение программы на модули позволяет выполнить их раздельную компиляцию. Отдельные модули программы могут иметь разную степень завершенности. Более завершенные модули не должны подвергаться повторной компиляции. В целом это позволяет быстрее компилировать и строить сложную программу. Средством взаимодействия модулей между собой являются заголовочные файлы, которые описывают общие для программы константы, переменные, имена типов и функции. Как правило, модулю кода на языке *Си* соответствует заголовочный

модуль. Использование модуля кода, имеющего заголовочный файл, предполагает включение этого заголовочного файла во все модули, использующие данный модуль кода. Поскольку при многократном включении одного и того же заголовочного файла его описания дублируются, заголовочные файлы имеют предохраняющие директивы условной компиляции. Например, следующий код в заголовочном файле препятствует его повторной компиляции:

```
#if !defined(__MYIO_H)
#define      __MYIO_H
    код модуля
#endif
```

При первом включении этого модуля символ __MYIO_H не определен, поэтому модуль компилируется и символ __MYIO_H определяется. При повторном включении символ __MYIO_H уже определен и код модуля пропускается.

В среде *Borland* C++ многомодульные программы разрабатываются на основе проекта. Для начала работы с проектом его нужно открыть, после чего включить в него файлы, составляющие программу. В проект могут быть включены файлы на языке *Си* (.c, .cpp), объектные модули (.obj) и библиотеки (.lib).

Литературные источники

Берри Р., Микинз Б. Язык Си: введение для программистов /Пер. с англ. и предисл. Д.Б. Подшивалова. — М.: Финансы и статистика, 1988. — 191 с.: ил.

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

Джехани Н. Программирование на языке Си: Пер. с англ.— М.: Радио и связь, 1988. — 272 с.: ил.

Уэйт М., Прата С., Мартин Д. Язык Си. Руководство для начинающих. Пер. с англ. — М.: Мир, 1988. — 512 с., ил.

Бруно Бабэ. Просто и ясно о Borland C++: Пер. с англ. — М.: БИНОМ, 1994. — 400 с.: ил.

Лукас Пол. C++ под рукой: Пер. с англ. — Киев: «ДиаСофт», 1993. — 176 с., ил.

Страуструп Б. Язык программирования C++: Пер. с англ. — М.: Радио и связь, 1991. — 352 с.: ил.

Стефан Дьюхарст, Кэти Сарк. Программирование на C++. Пер. с англ. — Киев: «ДиаСофт», 1993. — 272 с., ил.

MSDN Library — January 2001.

Приложение 1 — коды ASCII (кодировка MS-DOS)

Таблица кодов символов (вторая половина — кодировка *MS-DOS*)

10	16	Знак	10	16	Знак	10	16	Знак	10	16	Знак
128	80	А	160	A0	а	192	C0	└	224	E0	р
129	81	Б	161	A1	б	193	C1	┴	225	E1	с
130	82	В	162	A2	в	194	C2	┬	226	E2	т
131	83	Г	163	A3	г	195	C3	├	227	E3	у
132	84	Д	164	A4	д	196	C4	─	228	E4	ф
133	85	Е	165	A5	е	197	C5	┼	229	E5	х
134	86	Ж	166	A6	ж	198	C6	╞	230	E6	ц
135	87	З	167	A7	з	199	C7	╟	231	E7	ч
136	88	И	168	A8	и	200	C8	╚	232	E8	ш
137	89	Й	169	A9	й	201	C9	╔	233	E9	щ
138	8A	К	170	AA	к	202	CA	╩	234	EA	ъ
139	8B	Л	171	AB	л	203	CB	╦	235	EB	ы
140	8C	М	172	AC	м	204	CC	╠	236	EC	ь
141	8D	Н	173	AD	н	205	CD	═	237	ED	э
142	8E	О	174	AE	о	206	CE	╬	238	EE	ю
143	8F	П	175	AF	п	207	CF	╧	239	EF	я
144	90	Р	176	B0	░	208	D0	╨	240	F0	Ё
145	91	С	177	B1	▒	209	D1	╤	241	F1	ё
146	92	Т	178	B2	▓	210	D2	╥	242	F2	Є
147	93	У	179	B3	│	211	D3	╙	243	F3	є
148	94	Ф	180	B4	┤	212	D4	╘	244	F4	Ї
149	95	Х	181	B5	╡	213	D5	╒	245	F5	ї
150	96	Ц	182	B6	╢	214	D6	╓	246	F6	Ў
151	97	Ч	183	B7	╖	215	D7	╫	247	F7	ў
152	98	Ш	184	B8	╕	216	D8	╪	248	F8	°
153	99	Щ	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ъ	186	BA	║	218	DA	┌	250	FA	·
155	9B	Ы	187	BB	╗	219	DB	█	251	FB	√
156	9C	Ь	188	BC	╝	220	DC	▄	252	FC	№
157	9D	Э	189	BD	╜	221	DD	▌	253	FD	¤
158	9E	Ю	190	BE	╛	222	DE	▐	254	FE	■
159	9F	Я	191	BF	┐	223	DF	▀	255	FF	

Приложение 2 — коды ASCII (кодировка Windows)

Таблица кодов символов (вторая половина — кодировка *Windows*)

10	*16*	*Знак*	*10*	*16*	*Знак*	*10*	*16*	*Знак*	*10*	*16*	*Знак*
128	80	Ђ	160	A0		192	C0	А	224	E0	а
129	81	Ѓ	161	A1	Ў	193	C1	Б	225	E1	б
130	82	‚	162	A2	ў	194	C2	В	226	E2	в
131	83	ѓ	163	A3	Ј	195	C3	Г	227	E3	г
132	84	„	164	A4	¤	196	C4	Д	228	E4	д
133	85	…	165	A5	Ґ	197	C5	Е	229	E5	е
134	86	†	166	A6	¦	198	C6	Ж	230	E6	ж
135	87	‡	167	A7	§	199	C7	З	231	E7	з
136	88	€	168	A8	Ё	200	C8	И	232	E8	и
137	89	‰	169	A9	©	201	C9	Й	233	E9	й
138	8A	Љ	170	AA	Є	202	CA	К	234	EA	к
139	8B	‹	171	AB	«	203	CB	Л	235	EB	л
140	8C	Њ	172	AC	¬	204	CC	М	236	EC	м
141	8D	Ќ	173	AD		205	CD	Н	237	ED	н
142	8E	Ћ	174	AE	®	206	CE	О	238	EE	о
143	8F	Џ	175	AF	Ї	207	CF	П	239	EF	п
144	90	ђ	176	B0	°	208	D0	Р	240	F0	р
145	91	‘	177	B1	±	209	D1	С	241	F1	с
146	92	’	178	B2	І	210	D2	Т	242	F2	т
147	93	“	179	B3	і	211	D3	У	243	F3	у
148	94	”	180	B4	ґ	212	D4	Ф	244	F4	ф
149	95	•	181	B5	µ	213	D5	Х	245	F5	х
150	96	–	182	B6	¶	214	D6	Ц	246	F6	ц
151	97	—	183	B7	·	215	D7	Ч	247	F7	ч
152	98		184	B8	ё	216	D8	Ш	248	F8	ш
153	99	™	185	B9	№	217	D9	Щ	249	F9	щ
154	9A	љ	186	BA	є	218	DA	Ъ	250	FA	ъ
155	9B	›	187	BB	»	219	DB	Ы	251	FB	ы
156	9C	њ	188	BC	ј	220	DC	Ь	252	FC	ь
157	9D	ќ	189	BD	Ѕ	221	DD	Э	253	FD	э
158	9E	ћ	190	BE	ѕ	222	DE	Ю	254	FE	ю
159	9F	џ	191	BF	ї	223	DF	Я	255	FF	я