ОГЛАВЛЕНИЕ

1. Общая характеристика языка Си

Си – язык системного программирования; его принято относить к языкам *среднего уровня*, позволяющим выполнять как стандартные высокоуровневые подпрограммы, так и ассемблерно-ориентированный код.

Можно выделить следующие основные особенности Си:

- легкий доступ к аппаратным средствам компьютера, позволяющий писать высокоэффективные программы;
- высокая переносимость написанных на Си программ — как между компьютерами с различной архитектурой, так и между различными операционными средами;
- принцип построения "что пишем, то и получаем", т. е., в состав компилятора не включен код, который мог бы проверить корректность работы программы в процессе ее выполнения;
- в транслятор не включена информация о стандартных функциях, отсутствуют операции, имеющие дело непосредственно с составными объектами;
- компактный синтаксис, потенциально приводящий к трудноуловимым ошибкам.

2. Алфавит языка Си

Константа в Си может представлять собой число, символ или строку символов.

Целочисленные константы записываются, в зависимости от используемой системы счисления, в одной из следующих форм:

- десятичная: цифры от 0 до 9 со знаком "+", "-" или без знака. Примеры: `15`, `-305`.
- восьмеричная: лидирующий 0, далее цифры от 0 до 7. Примеры: `0777`, `0150`.
- шестнадцатеричная: лидирующий 0, далее символ "x" или "X", затем цифры от 0 до 9 и/или символы A-F или a-f. Примеры: `0x10`, `0XFF`.

Целочисленные константы могут иметь тип данных `int` (целочисленный) или `long` (длинный целый).

Программист может явно определить для любой целочисленной константы тип `long`, дописав символ `l` или `L` в конец константы. Реализован также *суффикс* `U` или `u`, означающий, что константа имеет тип `unsigned` (беззнаковый). Можно использовать одновременно оба суффикса — `L` и `U` — для одной и той же константы: `15L`, `1e6UL`, `0xFFFFul`.

Константы с плавающей точкой имеют следующую общую форму записи:

`[+ или -][цифры].[цифры][E][+ или -][цифры]`

Здесь E — признак экспоненциальной формы записи, задаваемый символом `E` или `e`. Либо целая, либо дробная часть константы могут быть опущены, но не обе сразу. Либо десятичная точка с дробной частью, либо экспонента могут быть опущены, но не обе сразу. Примеры: `-2.251e6`, `.45`, `1.E-03`, `1e-30`.

Символьная константа — это буква, цифра, знак пунктуации или специальный символ, заключенный в апострофы: `'c'`. Значение символьной константы равно ASCII-коду представляемого ею символа. Символ `c` может быть любым, за исключением апострофа `'` (записывается как `'\''`), обратного слеша `\` (`'\\'`) и новой строки (`'\n'`). Примеры символьных констант приведены в табл. 2.1.

Таблица 2.1 Примеры символьных констант

Константа	Значение
`'a'`	Малая латинская буква `a`
`'\007'`	Символ с кодом 7 ("звонок")
`'\b'`	Символ "забой" (BackSpace)
`'\x1B'`	Символ ESC в коде ASCII

Символьные константы имеют тип `char` или `int`. Младший байт хранит код символа, а старший байт, если он есть, — знаковое расширение младшего байта.

Множество символов языка Си включает символы ASCII-кода, при этом прописные и строчные буквы *различаются* компилятором в любом контексте.

Разделителями языка являются символы пробела, табуляции, перевода строки, возврата каретки, новой страницы, вертикальной табуляции и комментариев (см. табл. 2.2).

Специальные символы предназначены для представления пробельных и неграфических знаков в символьных константах и строках. Специальный символ состоит из обратного слэша, за которым следует либо буква, либо знаки пунктуации, либо комбинация цифр. Специальные символы языка Си перечислены в табл.2.2.

Таблица 2.2. Специальные символы языка Си

Специальный символ	16-ричная запись в коде ASCII	Наименование
`\n`	0A	Новая строка (перевод строки)
`\t`	09	Горизонтальная табуляция
`\v`	0B	Вертикальная табуляция
`\b`	08	Забой (backspace)
`\r`	0D	Возврат каретки
`\f`	0C	Новая страница
`\a`	07	Звуковой сигнал
`\'`	2C	Апостроф
`\"`	22	Двойная кавычка
`\\`	5C	Обратный слеш
`\ddd`		Байтовое восьмеричное значение
`\xdd`		Байтовое шестнадцатеричное значение

Стандартные библиотечные функции ввода и вывода текстовой информации обычно рассматривают пару символов `\r\n` как один символ.

Конструкция `\ddd` позволяет задать произвольное байтовое значение как последовательность от одной до трех восьмеричных цифр. Конструкция `\xdd` позволяет задать произвольное байтовое значение как последовательность от одной до двух шестнадцатеричных цифр

Нулевой код может быть записан как `\0` или `\x0`.

Символ <Ctrl>+<Z> (шестнадцатеричный код 1A) рассматривается как индикатор конца файла (символ `EOF`).

Комментарии компилятор Си также рассматривает как пробельные символы. Определены комментарии двух видов:

- `/*` многострочный `*/`. Комментируется весь текст после комбинации символов `/*` до первой встретившейся комбинации `*/`. Вложение многострочных комментариев опционально, т.е., зависит от настройки компилятора, поэтому не рекомендуется.
- `//` однострочный. Комментируется текст после комбинации символов `//` до конца строки.

Символьная строка — это последовательность символов, заключенная в двойные кавычки. В Си строка рассматривается как массив символов, каждый элемент которого представляет отдельный символ. Строка может содержать произвольное (в том числе нулевое) количество представимых символов. Вхождение в строку двойной кавычки ("), обратного слэша (\) или символа новой строки можно указать через специальные символы из табл 2.2. Примеры:

```
"Это символьная строка\n"
"Первый \\ Второй".
```

Для формирования символьных строк, занимающих несколько строк текста программы, используется комбинация символов "обратный слеш" и "новая строка":

```
printf ("\nHello,\
world");
```

Нулевой символ (`'\0'`) автоматически добавляется в качестве последнего байта символьной строки и служит признаком ее конца. Таким образом, строка из `N` символов занимает `N+1` байт памяти. В отличие от Паскаля, длина строки нигде не хранится. Каждая символьная строка в программе рассматривается как отдельный объект. Тип строки — массив элементов символьного типа данных `char`.

К *идентификаторам* относятся имена переменных, функций и меток в программе на Си.

Идентификатор Си — это последовательность из одной или более латинских букв, цифр и символов подчеркивания, которая начинается с буквы или символа подчеркивания. Допускается любое число символов в идентификаторе, однако только первые 32 символа рассматриваются компилятором как значащие.

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

Компилятор языка Си не допускает использования идентификаторов, совпадающих по написанию с ключевыми словами. Так, идентификатор `do` недопустим, однако `Do` или `DO` возможен.

Ключевые слова — это предопределенные идентификаторы, которые имеют специальное значение для компилятора Си. Их использование строго регламентировано. При необходимости можно с помощью *директивы препроцессора* `#define` определить для ключевых слов другие имена. В общем случае директива `#define` располагается на отдельной строке и имеет вид

```
#define НовоеКлючевоеСлово ИдентификаторСи
```

Примеры:

```
#define boolean int
#define begin {
#define word unsigned int
```

3. Операции и выражения

Операции — это комбинации символов, определяющие действия по преобразованию значений.

В Си определены 5 *арифметических* операций: сложение (знак операции "+"), вычитание ("-"), умножение ("*"), деление ("/") и взятие остатка от деления ("%"). Приоритеты и работа операций обычные: умножение, деление и взятие остатка от деления равноправны между собой и старше, чем сложение и вычитание. *Ассоциативность* (порядок выполнения) арифметических операций принята слева направо.

Операция % определена только над целыми операндами, а результат операции деления зависит от типа операндов. Деление целых в Си дает всегда целое число, если же хотя бы один из операндов вещественный, результат также будет вещественным:

```
3/2  //результат=1
3./2 //результат=1.5
```

В Си существует богатая коллекция разновидностей оператора присваивания. Обычное присваивание выполняется оператором "=":

```
x=y+z;
```

Возможно *объединение присваивания с другой операцией*, используемое как сокращенная форма записи для присваивания, изменяющего значение переменной:

```
x+=3; //эквивалентно x=x+3;
p*=s; //эквивалентно p=p*s;
```

Присваивание также может быть *составным*, при этом цепочка вычислений выполняется справа налево:

```
i=j=0;
c=1; a=b=c+1; //a=b=2;
```

Присваивание начального значения переменной (а также элементов массива) может быть выполнено непосредственно при описании:

```
int k=5;
```

Для распространенных операций *инкремента* (увеличения на 1) и *декремента* (уменьшения на 1) есть специальные обозначения ++ и -- соответственно:

```
i++; //эквивалентно i+=1; или i=i+1;
```

Операнд инкремента и декремента может иметь целый или вещественный тип или быть указателем.

Операции инкремента и декремента могут записываться как перед своим операндом (*префиксная* форма записи), так и после него (*постфиксная* запись). Для операции в префиксной форме операнд сначала изменяется, а затем его новое значение участвует в дальнейшем вычислении выражения. Для операции в постфиксной форме операнд изменяется после того, как его старое значение участвует в вычислении выражения:

```
int i=3;
printf ("\n%d",i++); //напечатает значение i=3
printf ("\n%d",++i); //напечатает значение i=4
```

Присваивание с приведением типа `(тип)(выражение)` использует заключенное в круглые скобки название типа, к которому нужно привести результат:

```
float f1=4,f2=3;
int a = (int)(f1/f2);  //a=1
f2=(float)a+1.5;       //f2=2.5
```

При этом разрешены преобразования типов, приводящие к потере точности, ответственность за это целиком лежит на программисте.

Логические операции в языке Си делятся на 2 класса. Во-первых, это *логические функции*, служащие для объединения условий, во-вторых, *поразрядные* логические операции, выполняемые *над отдельными битами* своих операндов. Операнды логических операций могут иметь целый, вещественный тип, либо быть указателями. Типы первого и второго операндов могут различаться. Сначала всегда вычисляется первый операнд; если его значения достаточно для определения результата операции, то второй операнд не вычисляется.

Логические операции не выполняют каких-либо преобразований по умолчанию. Вместо этого они вычисляют свои операнды и сравнивают их с нулем. Результатом логической операции является либо `0`, понимаемый как ложь,

либо ненулевое значение (обычно 1), трактуемый как истина. Существенно то, что в языке Си нет специального логического типа данных и тип результата логической операции — целочисленный.

Логическая функция "И" (соответствует and в Паскале) обозначается как &&, "ИЛИ" (or) как ||, унарная функция отрицания (not в Паскале) записывается как ! перед своим операндом:

```
if (x<y && y<z) min=x;
if (!(x>=a && x<=z))
  printf ("\nx не принадлежит [a,b]");
```

Приоритеты логических функций традиционны: операция ! старше чем &&, которая, в свою очередь, старше ||. При необходимости приоритеты могут быть изменены с помощью круглых скобок.

Как отмечено ранее, *поразрядные* логические операции выполняются над отдельными битами (разрядами) своих операндов. Имеется три бинарных и одна унарная поразрядная операция. Они описаны в табл. 3.1.

Таблица 3.1. Поразрядные логические операции

Операнд x	0	0	1	1	Описание
Операнд y	0	1	0	1	
x|y	0	1	1	1	Побитовое ИЛИ
x&y	0	0	0	1	Побитовое И
x^y	0	1	1	0	Побитовое исключающее ИЛИ
~x	1		0		Побитовое отрицание

Примеры:

```
char x=1,y=3; char z=x&y; //z=1
char x=0x00; x=x^0x01; //либо x^=1;
   //организует "флажок", переключающийся
   //между состояниями 0 и 1
```

Также побитовыми являются операции *сдвига* << и >>, которые сдвигают двоичное значение своего операнда влево или

вправо на число бит, определенное вторым операндом. При сдвиге влево освобождающиеся справа биты заполняются нулями. При сдвиге вправо результат зависит от того, какой тип данных получен после преобразования первого операнда. Если это беззнаковый тип, то свободные левые биты заполняются нулями. В противном случае они заполняются копией знакового бита. Если второй операнд отрицателен, то результат операции сдвига не определен. При выполнении сдвига потеря точности не контролируется. Если результат сдвига не может быть представлен типом первого операнда после преобразования типов, то информация теряется. Пример:

```
x<<=1; //соответствует x*=2;
```

Традиционные для любого языка операции отношения (сравнения) сравнивают первый операнд со вторым и вырабатывают целочисленное значение 1 (истина) или 0 (ложь). Операции отношения описаны в табл. 3.2.

Таблица 3.2. Операции отношения

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

Обратите внимание, что в отличие от присваивания, сравнение требует указания двойного знака "равно".

Операция *последовательного выполнения* по очереди вычисляет два своих операнда, сначала первый, затем второй. Оба операнда являются выражениями. Знак операции – символ "," (запятая):

```
i=0,j=0;
```

Результат операции имеет значение и тип второго операнда. Ограничения на типы операндов не накладываются, преобразования типов не выполняются. Операция обычно применяется для вычисления нескольких выражений там, где по

синтаксису допускается только одно выражение, например, в открывающей части цикла `for`.

Характерная для Си *условная операция* `? :` работает не с двумя, а с тремя операндами (является *тернарной*). Она имеет следующий формат: `операнд1 ? операнд2 : операнд3`.

Операнд1 вычисляется и сравнивается с нулем, при этом он может иметь целый, плавающий тип, либо быть указателем. Если *операнд1* не равен 0, вычисляется *операнд2* и результатом операции является его значение. В противном случае вычисляется *операнд3* и результатом является его значение. В любом случае вычисляется только один из операндов 2 или 3, но не оба. Примеры:

```
#define max(a,b) (a>b ? a : b)
y=(x>0?1:(x==0?0:-1)); //y = знаку числа x
```

Приоритет и *ассоциативность* операций языка Си влияют на порядок группирования операндов и вычисления выражения. В табл. 3.3 приведены операции языка Си в порядке убывания приоритета. Операции, расположенные в одной ячейке таблицы, имеют одинаковые приоритет и ассоциативность.

Табл. 3.3. Приоритет и ассоциативность операций

Знаки операций	Наименование	Ассоциативность
`() [] . ->`	Первичные	Слева направо
`+ - ~ ! * & ++ -- sizeof(тип) приведение типа`	Унарные	Справа налево
`* / %`	Мультипликативные	Слева направо
`+ -`	Аддитивные	Слева направо
`>> <<`	Сдвиг	Слева направо
`< > <= >=`	Отношение	Слева направо
`== !=`	Отношение	Слева направо
`&`	Поразрядное И	Слева направо
`^`	Поразрядное исключающее ИЛИ	Слева направо

`|`	Поразрядное включающее ИЛИ	Слева направо
`&&`	Логическое И	Слева направо
`||`	Логическое ИЛИ	Слева направо
`?:`	Условная	Справа налево
`= *= /= %= += -= <<= >>= &= |= ^=`	Простое и составное присваивание	Справа налево
`,`	Последовательное вычисление	Слева направо

Следует отметить, что приоритет некоторых операций явно неудачен, в частности, сдвига и поразрядных логических операций. Они имеют приоритет ниже, чем арифметические действия. Поэтому выражение `a=b&0xF0+1` вычисляется как `a= b&(0xF0+1)`, а `a+b>>1` как `(a+b)>>1`.

Преобразование типов в Си производится либо неявно, например, при преобразовании по умолчанию операнды преобразуются к более длинным типам, либо в процессе присваивания, либо явно, путем выполнения операции приведения типа:

```
переменная=(новый тип)выражение;
```

Во всех операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Преобразования при присваивании допускаются даже в тех случаях, когда они влекут за собой потерю информации:

```
int a=-100; float p=4.5; a=(int)p;
unsigned b = (unsigned)a;
```

4. Операторы

Операторы управляют процессом выполнения программы. Набор операторов Си содержит все типовые управляющие конструкции структурного программирования.

Программа на Си выполняется последовательно, оператор за оператором, за исключением случаев, когда какой-либо

оператор явно передает управление в другую часть программы, например при вызове функции или возврате из функции.

В теле некоторых операторов могут содержаться другие операторы. Оператор, находящийся в теле другого оператора, в свою очередь может содержать операторы.

Составной оператор ограничивается фигурными скобками `{ }`. Все другие операторы заканчиваются точкой с запятой (`;`). Точка с запятой в языке Си является признаком конца оператора, а не разделителем операторов, как в ряде других языков.

Перед любым оператором может быть записана метка, состоящая из имени и двоеточия. Метки распознаются только оператором `goto`.

Далее приведен полный список операторов Си.

Пустой оператор `;`. Применяется там, где по правилам синтаксиса требуется указать оператор, например, для создания пустого тела цикла или пустой ветви условного оператора. Выполнение пустого оператора не меняет состояния программы.

Составной оператор или *блок* `{ }`. Действие составного оператора заключается в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда какой-либо оператор явно передает управление в другое место программы. Типичное применение блока подобно другим языкам – ограничение тела функции, тела цикла, описания структурного типа данных или ветви условного оператора.

В начале составного оператора могут содержаться объявления переменных, локальных для данного блока, либо объявления, служащие для распространения на блок области действия глобальных объектов. Возможны объявления переменных и в любом другом месте блока (см. п. 5.1).

Оператор-выражение строится по правилам, описанным в п. 3.

Условный оператор `if` записывается в общем виде следующим образом:

```
if (УсловноеВыражение1) оператор1;
else if (УсловноеВыражение2) оператор2;
```

…
```
else if (УсловноеВыражениеN) операторN;
else оператор0;
```

Как и в других языках, выполняется только один из операторов `1,2,…,N`, в зависимости от того, какое из соответствующих условных выражений первым оценено как истинное. Если все условия ложны, выполняется `оператор0`. Любая из ветвей, кроме первой, необязательна. Круглые скобки и точки с запятой обязательны везде, где указаны. Еще раз акцентируем внимание на том, что в Си отсутствуют булевские выражения и в качестве условий применяются обычные выражения языка Си. Значение выражения считается истинным, если оно не равно нулю, и ложным, если равно нулю. Из этого следует, что условные выражения не обязательно должны содержать операции отношения. Условия могут быть записаны в обычном виде

```
if (a < 0) …
```

а могут выглядеть, например, так:

`if (a) …` или `if (a + b) …`.

Условия могут включать логические функции:

```
if (x>0 && y>0) ch=1;
```

Оператор пошагового цикла `for` имеет следующий общий вид:

```
for (НВ; УВ; ВП) Оператор;
```

Здесь `НВ` — начальное выражение, служащее для инициализации параметров цикла, `УВ` — условное выражение, прямо или косвенно определяющее число повторений цикла (цикл выполняется, пока `УВ` не станет ложным), `ВП` — выражение приращения, используемое для модификации параметра или параметров цикла. Любое из трех выражений может быть опущено, а также любые 2 выражения или все сразу.

Цикл работает следующим образом: сначала вычисляется НВ, если оно имеется. Затем вычисляется УВ и производится его оценка следующим образом:

- если УВ истинно (не равно нулю), то выполняется тело оператора. Затем вычисляется ВП, если оно есть, и процесс повторяется;
- если УВ опущено, то его значение принимается за истину и процесс выполнения продолжается, как описано выше. В этом случае цикл `for` бесконечен и может завершиться только при выполнении в его теле операторов `break`, `goto`, `return`;
- если УВ ложно, то выполнение цикла заканчивается и управление передается следующему за ним оператору программы. Цикл `for` может завершиться также при выполнении операторов `break`, `goto`, `return` в его теле.

Пример: показанный ниже цикл выполняется 10 раз.

```
for (x=1; x<11; x++) printf ("*");
```

Оператор *цикла с предусловием* `while`:

```
while (выражение) оператор;
```

Тело цикла `while` выполняется до тех пор, пока значение `выражения` не станет ложным (равным нулю). Сначала вычисляется `выражение`, если оно изначально ложно, то тело цикла не выполняется и управление передается на следующий за ним оператор программы. Если `выражение` истинно, то выполняется тело цикла. Перед каждым следующим выполнением цикла `выражение` вычисляется заново. Процесс повторяется до тех пор, пока `выражение` не станет ложным. Оператор `while` может завершиться досрочно при выполнении `break`, `goto` или `return` внутри своего тела.

Приведенный далее цикл `while` аналогичен показанному выше `for`:

```
x=1;
while (x<11) {
 printf ("*");
 x++;
}
```

Оператор *цикла с постусловием* `do` записывается в виде

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

Тело цикла do выполняется один или несколько раз до тех пор, пока значение выражения не станет ложным (равным нулю). Сначала выполняется тело цикла — оператор, затем вычисляется условие — выражение. Если выражение ложно, цикл завершается и управление передается следующему за телом цикла оператору программы. Если выражение истинно (не равно нулю), то тело цикла выполняется вновь, и выражение вычисляется повторно. Выполнение цикла повторяется до тех пор, пока выражение не станет ложным. Цикл do может завершиться досрочно при выполнении в своем теле операторов break, goto, return. Показанный далее цикл по действию совпадает с примерами на for и while:

```
x=1;
do {
 printf ("*"); x++;
} while (x<11);
```

Оператор-переключатель switch записывается в виде

```
switch (выражение) {
 объявление
 case КВ1: оператор1;
 …
 case КВN: операторN;
 default: оператор0;
}
```

Переключатель предназначен для выбора одного из нескольких альтернативных путей выполнения программы. Выполнение переключателя начинается с вычисления выражения, следующего за ключевым словом switch.

Конструкции case КВ:, синтаксически представляющие собой метки операторов, называются константными выражениями или *константами варианта*. Значение каждой КВ должно быть уникальным внутри тела оператора-переключателя..

Все КВ должны иметь порядковый тип. В ряде компиляторов КВ принудительно преобразуются к типу int.

Тип каждой КВ также приводится к типу выражения переключения.

После вычисления выражения управление передается тому из операторов 1,…,N тела переключателя, значение константы варианта которого совпадает с вычисленным значением.

Выполнение тела оператора-переключателя начинается с выбранного таким образом оператора и продолжается до конца тела или до тех пор, пока какой-либо оператор не передаст управление за пределы тела.

Оператор, следующий за ключевым словом default, выполняется, если ни одна из констант варианта не равна значению выражения переключения. Если ветвь default опущена, ни один оператор в теле переключателя не выполняется.

Синтаксически конструкции case и default являются метками, однако, на них нельзя передать управление по оператору goto. Метки case и default существенны только при начальной проверке, когда выбирается оператор для выполнения в теле переключателя. Все операторы тела переключателя, следующие за выбранным, выполняются последовательно, "не замечая" меток case и default, если только какой-либо оператор не передаст управление за пределы тела переключателя. Для выхода из тела переключателя после выполнения одной из ветвей, как правило, используется оператор разрыва break.

В заголовок составного оператора, формирующего тело switch, можно помещать объявления (см. п. 5), но инициализаторы, включенные в объявления, не будут выполнены. Приведенный ниже оператор-переключатель отслеживает коды нажатия некоторых клавиш:

```
char c=getch();
switch (c) {
 case ' ': printf ("\nПробел"); break;
 case 72: case 80:
  printf ("\nСтрелка вверх или вниз");
```

```
 break;
 default: printf ("\nДругая клавиша");
}
```

Оператор продолжения `continue` передает управление на следующую итерацию в циклах `do`, `for`, `while`. Он может появиться только в теле этих операторов. Остающиеся в теле цикла операторы при этом не выполняются. В циклах `do` и `while` следующая итерация начинается с вычисления условного выражения. В цикле `for` следующая итерация начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения.

Оператор разрыва `break` прерывает выполнение операторов `do`, `for`, `while` или `switch`. Он может содержаться только в теле этих операторов. Управление передается оператору программы, следующему за прерванным.

Оператор перехода `goto` имеет вид

```
goto метка;
```

и передает управление непосредственно на оператор, помеченный меткой:

```
метка: оператор;
```

Метка представляет собой обычный идентификатор. Область действия метки ограничивается функцией, в которой она определена. Каждая метка должна быть уникальна в пределах функции, где она указана. Нельзя передать управление по оператору `goto` в другую функцию. Метка оператора имеет смысл только для `goto`. Можно войти в блок, тело цикла, условный оператор, оператор-переключатель по метке. Нельзя с помощью `goto` передать управление на конструкции `case` и `default` в теле переключателя.

Оператор возврата `return выражение;` заканчивает выполнение функции, в которой он содержится, и возвращает управление в вызывающую функцию.

Управление передается в точку вызывающей функции, непосредственно следующую за оператором вызова функции. Значение `выражения`, если оно задано, вычисляется,

приводится к типу, объявленному для функции, содержащей оператор, и возвращается в вызывающую функцию. Если `выражение` опущено, то возвращаемое функцией значение не определено (является пустым).

5. Структура программы

Программа на Си включает следующие элементы:

- *директивы препроцессора* – определяют действия по преобразованию программы *перед* компиляцией, а также включают инструкции, которым компилятор следует *во время* компиляции;
- *объявления* – описания переменных, функций, структур, классов и типов данных;
- *определения* – тела выполняемых функций проекта.

5.1 Объявление переменной — задает имя и атрибуты переменной, приводит к выделению для нее памяти, а также может явно или неявно задавать начальное значение:

```
int x,y; float r=0;
```

Все переменные в языке Си должны быть явно объявлены перед использованием.

Объявления имеют следующий общий синтаксис:

```
<класс_памяти> <знаковость> <длина> тип список_переменных;
```

Все указания, перечисленные в треугольных скобках, могут быть опущены. Список состоит из одной переменной или имен переменных, перечисленных через запятую.

Класс памяти может принимать следующие значения:

- отсутствует или `auto` — переменная определена в том блоке `{ }`, в котором описана, и вложенных в него блоках. Определенная вне всех блоков переменная видима до конца файла. Принят по умолчанию в объявлении переменной на внутреннем уровне. Переменные класса `auto` автоматически не инициализируются. Память отводится в стеке. Как правило, ключевое слово `auto` опускается.

- `static` — переменная существует глобально независимо от того, на каком уровне блоков определена. Область действия – до конца файла, в котором она определена. По умолчанию имеет значение 0.
- `extern` — переменная или функция определена в другом файле, объявление представляет собой ссылку, действующую в рамках одного проекта;
- `register` — (только для типов данных char и int) — переменная *при возможности* хранится в регистре процессора.

Спецификации классов памяти `auto` и `register` не допускаются к явному указанию на внешнем уровне.

Знаковость может быть указана только для перечислимых (порядковых) типов, таких как `char`, `int`. Знаковость может принимать одно из двух значений:

- `signed` (по умолчанию) — переменная со знаком;
- `unsigned` — переменная без знака.

Длина определена для типов `int`, `float`, `double`:

- `short` — короткий вариант типа;
- отсутствует — вариант типа по умолчанию;
- `long` — длинный вариант типа.

Действие этих модификаторов зависит от компилятора и аппаратной платформы. Например, на IBM-PC совместимых компьютерах типы `short int` и `int` совпадают и занимают по 2 байта оперативной памяти, `long int` требует выделения 4 байт.

Базовыми *типами данных* являются:

- `char` — символьный;
- `int` — целочисленный;
- `float` — вещественный (плавающий) одинарной точности;
- `double` — вещественный (плавающий) двойной точности;

- `void` — "пустой" тип, имеет специальное назначение. Указание `void` в объявлении функции означает, что она не возвращает значений, а в списке аргументов объявления функции — что функция не принимает аргументов. Нельзя создавать переменные типа `void`, но можно указатели.

Переменные любых типов могут быть объявлены в любом месте проекта.

5.2. Объявление функции (*описание прототипа*) задает ее имя, тип возвращаемого значения и может задавать атрибуты ее формальных параметров. Общий вид объявления следующий:

```
ТипФункции имя (тип1 параметр1, …, типN параметрN);
```

Объявление необходимо для функций, которые описаны ниже по тексту, чем вызваны или описаны в другом файле проекта:

```
float f1(double t, double v) {
 return t+v;
 //Для этой функции прототип не указан
}
extern char f0 ();
 //f0 определена в другом файле проекта
f2(); //Прототип нужен т.к. тело функции
      //определено ниже ее вызова
void main () {
 f1 (1.,2.); f2 ();
}
int f2(void){//в прототипе не указан тип
             //функции, предполагается int
 return 0;
}
```

Функции могут быть объявлены в любом месте проекта. Для подключения функции из другого файла проекта можно использовать:

- модификатор `extern`;

- включение заголовочного файла внешнего модуля директивой препроцессора `#include <ИмяФайла.h>`

В примере ниже функция `f4()` определена во внешнем файле и должна быть доступна к моменту сборки приложения:

```
int f1 (int n) {
 extern int f4(int);
 return f4(n);
}
```

Обратите внимание, что если у функции опущен тип, то предполагается `int`:

```
main () {
 return 0;
}
```

но

```
void main () { /* ... */ }
```

В следующем примере в проект включены 2 файла, file1.cpp и file2.cpp. При этом прототип функции `f4()`, определенной в файле file2.cpp, включен в заголовочный файл file2.h и подключен к файлу file1.cpp с помощью директивы `#include`:

```
---------------------- листинг file1.cpp
#include <stdio.h>
#include <file2.h>

int f3(int n) {
 return f4(n);
}

void main () {
 printf ("\n%d",f3(3));
}
---------------------- листинг file2.cpp
int f4 (int k) {
 return ++k;
}
---------------------- листинг file2.h
int f4 (int);
```

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

Первый вид объявления позволяет определить *тег* (наименование типа) и ассоциированные с тегом элементы структуры, объединения или перечисления. После такого объявления имя типа может быть использовано в объявлениях переменных и функций для ссылки на него:

```
struct list {
 char name [20];
 long int phone;
}
struct list mylist [20];
```

Здесь объявлен структурный тип `list`, а затем описан массив структур типа `list` с именем `mylist`, состоящий из 20 элементов (см. п. 9).

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

```
typedef float real;
typedef long int integer;
typedef struct {
 float x,y;
} Point;
Point r;
```

Обратите внимание, что в отличие от директивы `#define`, имеющей синтаксис

```
#define новое_слово старое_слово
```

определение существующего типа через `typedef` имеет вид

```
typedef старый_тип новый_тип;
```

Типы могут быть объявлены в любом месте проекта, но для надежности следует делать их объявления глобальными.

Пример ниже показывает особенности, связанные с объявлением типов внутри блока:

```
#include <stdio.h>
typedef unsigned int word;
void f() {
 typedef unsigned long longint;
 #define byte unsigned char
 byte b;
}

void main () {
 word w=65535;
 byte b=65; //ошибки нет – #define
  //действует и вне блока, в котором указан
 longint l; //ошибка – тип longint
  //определен  только внутри функции f()
 printf ("\n%u,%c",w,b);
}
```

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

```
int f (int a, int b) {
 return (a+b)/2;
}
```

В определении функции допускается указание спецификации класса памяти `static` или `extern`, а также модификаторов типа функций `pascal`, `cdecl`, `interrupt`, `near`, `far` и `huge` (см. п. 11).

Тип возвращаемого значения, задаваемый в определении функции перед ее именем, должен соответствовать типу возвращаемого значения во всех объявлениях этой функции, если они имеются в программе.

При вызове функции ее выполнение начинается с первого оператора. Функция возвращает управление при выполнении оператора `return значение;`, либо когда выполнение доходит до конца тела функции.

В первом случае `значение` вычисляется, преобразуется к типу возвращаемого значения и возвращается в точку вызова функции. Если оператор `return` отсутствует или не содержит выражения, то возвращаемое значение функции не определено. Если в этом случае вызывающая функция ожидает возвращаемое значение, то поведение программы непредсказуемо.

Список объявлений формальных параметров функции содержит их описания через запятую. Тело функции (составной оператор) начинается непосредственно после списка. Список параметров может быть пустым, но и в этом случае он должен быть ограничен круглыми скобками. Если функция не имеет аргументов, рекомендуется указать это явно, записав в списке объявлений параметров ключевое слово `void`.

Формальные параметры могут иметь базовый тип, либо быть структурой, объединением, указателем или массивом. Указание первой (или единственной) размерности для массива не обязательно. Массив воспринимается как указатель на тип элементов массива.

Параметры могут иметь класс памяти `auto` (по умолчанию) или `register`. По умолчанию формальный параметр имеет тип int.

Идентификаторы формальных параметров не могут совпадать с именами переменных, объявляемых внутри тела функции, но возможно локальное переобъявление формальных параметров внутри вложенных блоков функции.

Тип каждого формального параметра должен соответствовать типу фактического аргумента и типу соответствующего аргумента в прототипе функции, если таковой имеется.

После преобразования все порядковые формальные параметры имеют тип `int`, а вещественные — тип `double`.

После последнего идентификатора в списке формальных параметров может быть записана запятая с многоточием (`,...`). Это означает, что число параметров функции переменно, однако не меньше, чем следует идентификаторов до многоточия.

Фактический аргумент может быть любым значением базового типа, структурой, объединением или указателем. По умолчанию фактические аргументы передаются по значению. Массивы и функции не могут передаваться как параметры, но могут передаваться *указатели* на эти объекты. Поэтому массивы и функции передаются *по ссылке*. Значения фактических аргументов копируются в соответствующие формальные параметры. Функция использует только эти копии, не изменяя сами переменные, с которых копия была сделана.

Возможность доступа из функции не к копиям значений, а к самим переменным обеспечивают указатели (см. п. 7) и параметры-ссылки (см. п. 5.5).

Стандарт языка не предполагает размещения тела одной функции внутри другой, хотя прототип функции может быть указан внутри другой функции. Т.е., определение функции возможно только *вне* всех блоков.

Программа на Си должна содержать хотя бы одно определение — тело функции с именем `main`**.** Функция `main` является единственной точкой входа в программу. На весь проект может быть только одна функция с именем `main`.

Текст программы может быть разделен на несколько исходных файлов. При компиляции программы каждый из исходных файлов должен быть скомпилирован отдельно, а затем связан с другими файлами компоновщиком. Исходные файлы можно объединять в один файл, компилируемый как единое целое, посредством директивы препроцессора `#include`. В тексте примера ниже файл file3.cpp включает file2.cpp, используя из него функцию `f4()`:

```
#include <stdio.h>
#include "file2.cpp"
void main () {
```

```
 printf ("\n%d",f4(0));
}
```

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

Исходный файл не обязательно содержит выполняемые операторы. Обычно удобно размещать объявления переменных, типов и функций в файлах типа *.h, а в других файлах использовать эти объекты путем их определения. Подключение заголовочного файла *.h выполняется директивой

```
#include "ИмяФайла.h"
```

предполагающей поиск заголовочного файла в текущей папке проекта. С другой стороны, директива

```
#include <ИмяФайла.h>
```

предназначена для подключения стандартных заголовочных файлов и ищет их в папках, указанных в настройке Include directories компилятора (см. п. 10.2).

Функция `main` также может иметь формальные параметры. Значения формальных параметров `main` могут быть получены извне — из командной строки при вызове программы и из таблицы контекста операционной системы. Таблица контекста заполняется системными командами SET и PATH:

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

Доступ к первому аргументу, переданному программе, можно осуществить с помощью выражения `argv[1]`, к последнему аргументу — `argv[argc-1]`. Аргумент `argv[0]` содержит строку вызова самого приложения.

Параметр `envp` представляет собой указатель на массив строк, определяющих системное окружение, т.е. среду выполнения программы. Стандартные библиотечные функции

getenv и putenv (библиотека stdlib.h) позволяют организовать удобный доступ к таблице окружения.

Существует еще один способ передачи аргументов функции main — при запуске программы как независимого подпроцесса из другой программы, также написанной на Си (функции семейства exec и spawn, библиотека process.h, см. п. 8.13).

Пример ниже представляет собой законченную программу на Си, состоящую из трех функций.

```
#include <stdio.h>
int long power(int,int); //прототип функции
//необходим, т.к. тело функции ниже вызова
void printLong (char *s, long int l) {
 //без прототипа
 printf ("%s%ld",s,l);
}
void main() {
 for (int i = 0; i <= 30; i++) {
  int long l1=power(2,i);
  printf("%d",i);
  printLong (" ",l1);
  printf("\n");
 }
}
int long power(int x, int n) {
 int i; long int p;
 for (i=1,p=1L; i <= n; ++i) p *= x;
 return (p);
}
```

Функции на Си могут быть рекурсивными:

```
int long power(int x, int n) {
 return (n>1 ?
 (long int)x*power(x,n-1) : x);
}
```

Компилятор не ограничивает число рекурсивных вызовов, но операционная среда может накладывать практические ограничения. Так как каждый рекурсивный вызов требует

дополнительной стековой памяти, то слишком большое их количество может привести к переполнению стека.

5.5 Передача параметров по значению и по ссылке. По умолчанию аргументы функций передаются *по значению*. При передаче по ссылке перед аргументом в прототипе и в заголовке указывается операция "адрес" (`&`):

```
void swap (int &,int &);
//...
void swap (int &a, int &b) {
 int c=a; a=b; b=c;
}
//...
swap (x1,x2);
```

Здесь функция `swap` поменяла местами фактические значения аргументов `x1` и `x2`.

Так как компилятор на первом этапе формирует внешние имена функций, в одном проекте возможны функции, имеющее одинаковые имена, но отличающиеся списком параметров:

```
int f(int,int);
float f(float,float);
```

но не

```
float f(int,int);
```

5.6. Время жизни и область действия объектов. Итак, описания переменных в языке Си не предполагают их объединения в отдельные разделы описаний. В этой связи огромную важность приобретают правила, касающиеся *времени жизни* и *области действия* объектов. Эти правила описаны в табл. 5.1.

Табл. 5.1. Время жизни и область действия

Объект	Время жизни	Область действия
функция	глобально для всех функций (все время	ниже по тексту от объявления или определения

	выполнения программы)	
переменная, определенная на внешнем уровне (вне всех блоков)	глобально	от точки программы, в которой объявлена, до конца исходного файла, включая все функции и вложенные блоки. При этом может быть вытеснена одноименной переменной, объявленной внутри блока. Если указан класс памяти static - только до конца исходного файла, содержащего объявление
переменная, определенная внутри блока	локально, пока выполняется этот и вложенные блоки (если не указан класс памяти static)	в этом и вложенных блоках, может быть переопределена во вложенных блоках

Пример ниже наглядно демонстрирует переопределение глобальных и локальных переменных одноименными переменными более низких уровней.

```
#include <stdio.h>
static int i=12;
  //глобальная статическая переменная
void f() {
 printf ("\n%d",i); //i=12
}
void main () {
 int i=5;
 printf ("\n%d\n",i); //i=5
 {//без этого блока компиляторы выведут
  //ошибку "множественная декларация
  //переменной", т.к. открывающая часть
```

```
  //цикла for выполняется до его тела
  for (int i=0; i<10; i++)
   printf ("%d ",i); //i=0,1,…,9
 }
 printf ("\n%d",i); //i=5
 f();
}
```

6. Статические массивы

Массив объявляется одним из следующих способов:

```
тип идентификатор [константное выражение];
тип  идентификатор [] = {список элементов};
```

Элементы списка перечисляются через запятую, в этом случае размерность массива определяется по фактически указанному количеству элементов. Примеры:

```
int x[10];
float a[]={3.5,4.5,5.5}; //размерность=3
char div[3]={' ','\n','\t'};
```

Элементы массивов в Си всегда нумеруются с нуля.

Синтаксис индексного выражения для обращения к элементу массива имеет следующий вид:

```
выражение1[выражение2]
```

Значение индексного выражения находится по адресу, который вычисляется как сумма значений `выражения1` и `выражения2`. `Выражение1` должно иметь тип указателя на некоторый тип, например, быть идентификатором массива, а `выражение2`, заключенное в квадратные скобки, должно иметь целый тип или преобразовываться к нему.

Индексное выражение может иметь более одного индекса, что соответствует многомерному массиву. Синтаксис такого выражения следующий:

```
выражение1[выражение2][выражение3]…
```

Такое индексное выражение интерпретируется слева направо. Сначала вычисляется самое левое индексное выражение — `выражение1[выражение2]`. С адресом,

полученным в результате сложения выражения1 и выражения2, складывается (по правилам сложения указателя и целого) выражение3 и т. д. Выражение2 и последующие выражения имеют целый тип.

Элементы многомерного массива запоминаются построчно. Примеры:

```
char a[2][3];
float matrix[10][15];
int b[3][3]={
 {1,2,3},
 {4,5,6},
 {7,8,9}
};
```

В следующем примере выполняется определение, обработка и печать статической матрицы.

```
#include <stdio.h>
void main () {
 int b[3][3]={
  {1,2,3},
  {1,2,3},
  {1,2,3}
 };
 b[0][0]=2;
 b[2][2]=b[0][0]*4;
 for (int i=0; i<3; i++) {
  printf ("\n");
  for (int j=0; j<3; j++)
   printf ("%d ",b[i][j]);
 }
}
```

Пример ниже выполняет подсчет количества разных цифр, пробелов и остальных символов, вводимых с клавиатуры. Для завершения программы служит ввода символа EOF (<Ctrl>+<Z>). Преобразование кода символа c в целочисленный индекс элемента массива ndigit выполняется конструкцией c-'0'.

```
#include <stdio.h>
void main() {
 int  c, i, nwhite, nother;
 int  ndigit[10];
 nwhite = nother = 0;
 for (i = 0; i < 10; ++i) ndigit[i] = 0;
 while ((c = getchar()) != EOF)
  if (c >= '0' && c <= '9') ++ndigit[c-'0'];
  else if (c==' ' || c=='\n' || c=='\t')
   ++nwhite;
  else ++nother;
 printf("digits =");
 for (i = 0; i < 10; i++)
  printf(" %d", ndigit[i]);
 printf("\nwhite space = %d, other = %d\n",
  nwhite, nother);
}
```

7. Указатели

Указателем называют переменную, содержащую адрес другой переменной.

Таким образом, именно указатели дают возможность *косвенного доступа* программы к объектам в памяти. Предположим, что `x` — переменная, например, типа `int`, а `px` — указатель. Они описываются следующим образом:

```
int x; int *px;
```

Из описания следует, что указатель может указывать только *на определенный тип* объектов.

Унарная операция `*` ("*взятие значения*") рассматривает свой операнд как *адрес* конечной цели и обращается по этому адресу, чтобы *извлечь содержимое*. Следовательно, если `y` также имеет тип `int`, то операция

```
y = *px;
```

присваивает `y` содержимое того объекта, на который *указывает* `px`. Так последовательность операций

```
px = &x;
y = *px;
```

присваивает `y` то же самое значение, что и оператор

```
y = x;
```

Указателю можно присваивать адрес объекта и непосредственно при описании:

```
int x;
int *px=&x;
```

Унарная операция `&` ("*взятие адреса*") уже упоминалась нами в п. 5.5. Здесь указатель `px` содержит *адрес* переменной `x`, ему *присвоен* ее адрес.

Над указателями определены операции сложения и вычитания с числом:

```
px++;
```

В этом примере унарная операция `++` увеличивает `px` так, что он указывает на следующий элемент набора объектов того типа, что задан при определении указателя. Именно подобное уменьшение или увеличение указателя дает возможность сканировать такие объекты, как строки и массивы.

```
px-=2;
```

Здесь операция `px+=i` увеличивает `px` так, чтобы он указывал на элемент, отстоящий на `i` элементов от текущего.

Сравнение указателей в общем случае некорректно! Это связано с тем, что одним и тем же физическим адресам памяти могут соответствовать различные пары значений "сегмент-смещение".

Указатели являются *переменными*, соответственно, их можно присваивать:

```
int *py=px;
```

или

```
int *py; py=px;
```

Теперь `py` указывает на то же, что `px`.

Указатели `px` и `py` *адресуют* одну и ту же переменную `x`, но сравнение `px==py` может быть некорректным в отличие от сравнения *значений* `*px==*py`.

Унарная операция `&` выдает *адрес* объекта, так что оператор

```
px = &x;
```

присваивает адрес x переменной px; говорят, что теперь px *указывает* на x. Операция & применима только к переменным и элементам массива, конструкции вида &(x-1) и &3 являются незаконными. Нельзя также получить адрес регистровой переменной.

Указатели могут входить в выражения. Например, если px указывает на целое x, то *px может появляться в любом контексте, где может встретиться x. Так, оператор

```
y = *px + 1;
```

присваивает y значение, на 1 большее значения x (получаем значение из указателя, затем прибавляем 1). Оператор `printf ("\n%d", *px);` печатает текущее значение x, а оператор `d = sqrt((double)*px);` получает в d квадратный корень из x, причем до передачи функции sqrt значение x преобразуется к типу double.

В выражениях вида

```
y = *px + 1;
```

унарные операции * и & связаны со своим операндом *более крепко*, чем арифметические операции (см. табл. 3.3), так что это выражение берет значение, на которое указывает px, прибавляет 1 и присваивает результат переменной y:

```
y = (*px) + 1;
```

Выражение

```
y = *(px + 1);
```

имеет совершенно иной смысл: записать в y значение, взятое из ячейки памяти, следующей за той, на которую указывает px. Адрес, на который указывает px, при этом не изменится.

Ссылки на указатели могут появляться в левой части операторов присваивания. Если px указывает на x, то

```
*px = 0;
```

записывает в x значение 0, а

```
*px += 1;
```

увеличивает значение x на единицу, как и выражение

```
(*px)++;
```

Круглые скобки в последнем примере *необходимы*; если их опустить, то поскольку унарные операции, подобные * и ++, выполняются справа налево, это выражение увеличит px, а не ту переменную, на которую указывает px.

Наконец, операция

```
*px++;
```

получает значение из указателя, затем сдвигает указатель на следующую ячейку памяти (поскольку использована постфиксная форма инкремента).

Рассмотрим подробнее различные аспекты использования указателей.

7.1 Указатели и аргументы функций.

Так как в Си передача аргументов функциям осуществляется по значению, вызванная подпрограмма не имеет непосредственной возможности изменить переменную из вызывающей подпрограммы.

Указатели в качестве аргументов используются в функциях, которые должны возвращать более одного значения.

Пример ниже иллюстрирует применение указателей в качестве аргументов функции — иными словами, *передачу параметров по адресу и прием по значению*.

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

Сравните этот подход с ранее применявшимися *передачей по значению и приемом по адресу*, использующими операцию &:

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

```
swap (a,b);
```

7.2. Указатели и массивы.

Как правило, указатель используется для последовательного доступа к элементам статического или динамического массива. Так, конструкция

```
int a[]={1,2,3};
int *p=a;
for (int i=0;i<3;i++)
 printf ("\t%d",*p++);
```

последовательно распечатает элементы массива `a`, доступ к которым осуществлялся через указатель `p`.

Присваивание указателю адреса нулевого элемента массива можно было записать и в виде

```
int *p=&a[0];
```

или

```
int *p=&(*a+0);
```

или

```
int *p=&*a;
```

Следующий пример иллюстрирует сканирование строки с помощью указателя.

```
int strlen(char *s){
 int n;
 for (n = 0; *s != '\0'; s++) n++;
 return(n);
}
char *s="Test";
int len=strlen (s);
```

Здесь с указателем на тип `char` сопоставляется статическая строка символов, подробнее этот прием будет раскрыт в п. 7.3.

Тело функции `strlen` можно было записать и короче:

```
int n=0;
while (*s++) n++;
return n;
```

Существуют важные различия между массивами и указателями.

- Указатель занимает одну ячейку памяти, предназначенную для хранения машинного адреса (в частности, адреса нулевого элемента массива). Массив занимает столько ячеек памяти, сколько элементов определено в нем при его объявлении. Только в выражении массив представляется своим адресом, который эквивалентен указателю.
- Адрес массива является постоянной величиной, поэтому, в отличие от идентификатора указателя, идентификатор массива не может составлять левую часть операции присваивания.

Для одномерного массива следующие 2 выражения эквивалентны, если a — массив или указатель, а b — целое:

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

Аналогично, для матрицы a с целочисленными индексами i и j эквивалентны выражения

```
a[i][j]    *(*(a+i)+j)
```

Так, следующий оператор выводит элемент матрицы a[1][2]:

```
int i=1,j=2;
printf ("%d",*(*(a+i)+j));
```

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

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

то можно присвоить указателю p адрес переменной i:

```
p = &i;
```

но изменить значение указателя без приведения типа нельзя:

```
p++;  /* недопустимо */
(int *)p++; /* допустимо */
```

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

7.3. Указатели и символьные данные.

Если описать указатель message в виде

```
char *message;
```

то в результате оператора

```
message = "Any string of text";
```

message будет указывать на фактический массив символов. Это не копирование строки, так как в операции участвует только указатель. Также важно то, что в Си не предусмотрены какие-либо операции для обработки всей строки символов как целого. Как и в других контекстах, присваивание значения переменной можно объединить с ее определением:

```
char *message = "Any string of text";
```

Указатели в сочетании с операцией инкремента естественным образом используются для сканирования строк. В таком контексте динамически меняющийся указатель на строку часто называют просто строкой.

Следующий пример реализует на Си функцию копирования строки с именем strcpy.

```
#include <stdio.h>
char *strcpy (char *s, char *t) {
 char *n=s; //запомнили, куда показывал s
 while (*t!='\0') { *s++=*t++; }
 return n; //s сдвинулся, вернули
           //его начальное значение
}
void main () {
 char *s1="none",*s2="test";
 printf ("\n%s",strcpy(s1,s2));
}
```

Функция получает 2 указателя на строки s (строка назначения) и t (строка-источник). Значением *t++ является

символ, на который указывал t до увеличения; постфиксная операция ++ не изменяет t, пока этот символ не будет извлечен. Точно так же этот символ помещается в старую позицию s, до того как s будет увеличено. Конечный результат заключается в том, что все символы копируются из t в s, исключая завершающий символ нуля.

Более кратко процесс копирования можно было бы описать в виде

```
while ((*s++ = *t++) != '\0');
```

Здесь увеличение s и t вынесено в проверочную часть цикла. Также за счет того, что сначала выполняется присваивание, а затем сравнение с нулевым байтом, код копирует и завершающий символ '\0'. Наконец, сравнение с нулем также можно было опустить:

```
while (*s++ = *t++);
```

Напишем функцию сравнения строк с использованием указателей. Она вернет число меньше 0, если строка s лексикографически (по кодам символов) предшествует t, вернет 0, если строки одинаковы и положительное значение, если s "больше" t по кодам символов.

```
int strcmp(char *s, char *t) {
 for ( ; *s == *t; s++, t++)
  if (*s == '\0') return(0);
 return(*s-*t);
}
```

Запись этой функции также можно сократить.

7.4. Указатели и динамическая память.

Подробнее тема использования динамической памяти рассматривается в п. 8.5. Здесь мы ограничимся двумя стандартными функциями, имеющимися в библиотеке stdlib.h:

- функция void *malloc(n) с целочисленным беззнаковым аргументом n возвращает в качестве своего значения нетипизированный указатель p, который указывает на первый из n выделенных байт памяти. Эта память может быть

использована программой для хранения данных; перед использованием указатель должен быть типизирован операцией приведения типа. Если выделить память не удалось, функция возвращает NULL. Операция приведения типа имеет вид sizeof(тип) и позволяет узнать размер переменной этого типа в байтах.

- функция `void free(p)` освобождает приобретенную таким образом память, так что ее в дальнейшем можно снова использовать. Обращения к `free` должны производиться в порядке, обратном тому, в котором производились обращения к `malloc`.

В приведенном далее примере указателю `p` сопоставляется динамическая память под строку из `n` символов, значение `n` вводится пользователем.

```
#include <stdlib.h>
#include <stdio.h>
//...
unsigned char *p;
unsigned n;
printf ("\nN="); fflush (stdin);
scanf ("%u",&n);
p=(unsigned char *)
 malloc(n*sizeof(unsigned char));
if (p==NULL) {
 //Здесь производится диагностика ошибки
}
```

В следующем примере функция `strsave` копирует свою строку-аргумент в динамически выделенную область памяти.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
 /*прототипы строковых функций*/
char *strsave(char *s) {
 char *p=NULL;
 p=(char *) malloc(strlen(s)+1);
 if (p != NULL) strcpy(p, s);
```

```
 return(p);
}
void main () {
 char *s1="hello", *s2;
 s2=strsave(s1);
 printf ("\n%s",s2);
}
```

7.5. Указатели и функции с переменным числом аргументов.

В конце списка формальных параметров функции могут быть записана запятая и многоточие. Это означает, что число аргументов функции переменно, но не меньше, чем число имен типов, заданных до многоточия.

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

В списке типов аргументов в качестве имени типа допускается также конструкция `void *`, которая специфицирует аргумент типа "указатель на любой тип". Для доступа к переменному списку параметров можно использовать указатели.

Приведенная ниже функция с переменным числом аргументов используется для записи кодов клавиш в собственный буфер клавиатуры.

```
#define KEY unsigned int
#define MAX_BUF 9
static KEY Buf [MAX_BUF];
KEY Start = 0;

void Set_Key( KEY kol, ... ) {
//Параметр kol задает
//количество остальных параметров
 KEY *ptr;   //Указываем на первый символ
 ptr = &kol; //в строке параметров
 for(; kol != 0; kol-- ) {
```

```
    Buf [Start++] = *++ptr;
    //Последовательно записываем символы
    Start %= MAX_BUF;
    //во внутренний буфер с контролем
    //его переполнения
   }
   return;
  }
```

Предполагается, что полные двухбайтовые коды клавиш определены в заголовочном файле:

```
#define ENTER   0x000D
#define END     0x4F00
#define RIGHT   0x4d00
```

Тогда функция может быть вызвана, например, так:

```
Set_Key (1,ENTER);
Set_Key (2,END,RIGHT);
```

В дальнейшем функция получения кодов символов может извлекать ранее записанные символы следующим образом:

```
#define MODE unsigned char
KEY Get_Key (void) {
 KEY sim;
 MODE scan,ascii;
 if (Start != End) {
  sim = Buf [End++];  End %= MAX_BUF;
 }
 else  {
  asm MOV AH,0x00;
  asm INT 16H;
  sim=_AX; //или вызов bioskey (0);
  scan=(MODE)((sim&0xff00)>>8);  //_AH
  ascii=(MODE)(sim&0x00ff);      //_AL
  if (ascii) scan=0;
  sim=(scan<<8)+ascii;
 }
 return (sim);
}
```

Здесь инструкция `asm` позволяет выполнить ассемблерный код непосредственно из программы на Си. Инструкция разрешает выполнять и блок ассемблерных команд: `asm { }`. Существует также библиотека stdarg.h для работы с переменными списками аргументов.

7.6. Указатели и прямой доступ к памяти.

Организацию прямого доступа к памяти с помощью указателей рассмотрим на примере обращения к памяти видеоадаптера в текстовом режиме монитора с разрешением экрана 80*25 позиций. Как известно, видеопамять при этом начинается с адреса B800:0000 и состоит из пар байт "символ-атрибут", описывающих экранные позиции слева направо и затем сверху вниз.

```
static unsigned char far *s =
 (unsigned char far *) 0xB8000000UL;
void putc (int x, int y, char c) {
 *(s+y*160+x*2)=c;
}
void main () {
 putc (0,0,'*'); putc (79,0,'*');
 putc (0,24,'*'); putc (79,24,'*');
 //вывели звездочки по краям
 //текстового экрана
}
```

Модификатор `far` определяет "длинный" 4-байтовый указатель (см. п. 11).

Поскольку одна строка экрана консоли состоит из 80 символов и требует 160 байт памяти, конструкция `*(s+y*160+x*2)`, где `x` – экранный столбец, а `y` – строка, адресует на экране позицию в y-строке и x-столбце.

Учитывая, что операции сдвига порождают более быстрый код, чем умножение, а $160=128+32=2^7+2^5$, в функции `putc` лучше использовать присваивание вида

```
*(s+ (y<<7) + (y<<5) + (x<<1)) = c;
```

8. Стандартная библиотека Си

Основные отличия стандартной библиотеки Си от других языков состоят в следующем:

- более сильная интеграция с языком. Так, в самом языке Си нет никаких средств ввода-вывода, поэтому ни одна программа не может быть написана без использования функций стандартной библиотеки;
- несмотря на то, что существуют стандарты языка Си, ряд функций уникален для той или иной системы программирования;
- многие функции имеют несколько "подфункций", решающих схожие задачи и обычно отличающихся одной буквой в названии, например, `abs`, `fabs` и `labs` для функции взятия модуля от разных типов данных, `printf`, `fprintf` и `sprintf` для стандартной функции вывода и т.д.

Далее рассматриваются основные разделы и функции стандартной библиотеки. Более полная информация может быть получена из рекомендуемых книг и справочной системы.

8.1. Определение класса символов и преобразование символов

Все функции, приведенные в табл. 8.1 имеют тип `int` и возвращают `int`. Возвращаемая величина равна 0, если условие проверки не выполняется. Все функции реализованы как макроопределения, заданные в файле ctype.h**.**

Таблица 8.1. Функции проверки и преобразования символов

Функция	Краткое описание
`isalnum`	проверка на латинскую букву или цифру
`isalpha`	проверка на латинскую букву
`isascii`	проверка на символ из набора кодировки ASCII
`iscntrl`	проверка на управляющий символ
`isdigit`	проверка на десятичную цифру
`isgraph`	проверка на печатный символ, исключая пробел

islower	проверка на малую латинскую букву
isprint	проверка на печатный символ
ispunct	проверка на знак пунктуации
isspace	проверка на пробельный символ
isupper	проверка на заглавную латинскую букву
isxdigit	проверка на шестнадцатеричную цифру
toascii	преобразование символа в код ASCII
tolower	проверка и преобразование в малую латинскую букву, если передана заглавная буква
toupper	проверка и преобразование малой латинскую буквы в заглавную
_tolower	преобразование латинскую буквы в малую (без проверки)
_toupper	преобразование латинскую буквы в заглавную (без проверки)

В примере ниже символ c преобразуется к верхнему регистру:

```
char c='x';
if (islower(c)) c=toupper (c);
```

Пример ниже подсчитывает относительные частоты букв кириллицы во вводимом тексте.

```
#include<stdio.h>
#include<conio.h>
void main(void) {
char lower_case[] =
 "абвгдежзийклмнопрстуфхцчшщьъыэюя";
char upper_case[] =
 "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЬЪЫЭЮЯ";
static float frequency[sizeof(lower_case)],
 total=0.;
char ch; int index;
 /* Завершение ввода - символ '$' */
puts("Печатайте текст\n");
while((ch = getchar()) != '$') {
 //Определяем порядковый номер буквы.
 for (index = 0; index < 32; index++)
```

```
 if((lower_case[index]==ch)||
    (upper_case[index]==ch)) {
  total++;
  frequency[index]++;
  break;
 }
}
puts("\nОтносительные частоты\n");
if (total) {
 for (index = 0; index < 32; index++) {
  frequency[index]=
  frequency[index]*100./total;
  printf("%c-%5.1f ",upper_case[index],
   frequency[index]);
 }
}
else
 puts("\aНи одна буква не введена");
}}
```

8.2. Работа с областями памяти и строками

Основными понятиями являются *буфер* и *строка*:

Буфер — это область памяти, рассматриваемая как последовательность байт. Размер буфера всегда задается явно.

Строка — это последовательность байт, завершаемая байтом с кодом `'\0'`. В языке Си длина строки, в отличие от буфера, нигде не хранится и может быть получена в цикле сканирования строки слева направо. Нулевой байт также считается принадлежащим строке, поэтому строка из `n` символов требует `(n+1)*sizeof(char)` байт памяти.

В библиотеке mem.h для работы с буферами предназначен ряд функций, основные из них перечислены в табл. 8.2. В компиляторах от Borland прототипы указанных функций содержит также библиотека string.h.

Таблица 8.2. Функции для работы с буферами

Функция	Краткое описание
memccpy	void *memcpy (void *s1, const void *s2, int c , size_t n); копирует n символов из буфера s2 в s1 с указанием дополнительного стоп-символа c
memchr	void *memchr (const void *s, int c, size_t n); возвращает указатель на первое вхождение символа c в буфер s длиной n символов
memcmp	int memcmp(const void *s1, const void *s2, size_t n); сравнивает n символов из двух буферов как беззнаковые. memicmp – то же с игнорированием регистра латинских символов
memcpy	void *memcpy(void *dest, const void *src, size_t n); копирует n символов из буфера src в dest
memset	void *memset(void *s, int c, size_t n); инициализирует значением c указанные n байт в буфере s
memmove	void *memmove(void *dest, const void *src, size_t n); работает как memcpy, но корректно обрабатывает перекрывающиеся области памяти

Прототипы функций обработки строк содержатся в файле string.h. Все функции работают со строками, завершающимися нулевым байтом '\0'. Для функций, возвращающих указатели, в случае ошибки возвращается NULL.

Типизированный модификатор size_t определен в ряде стандартных библиотек следующим образом:

```
typedef unsigned size_t;
```

Основные библиотечные функции описаны в табл. 8.3.

Таблица 8.3. Функции для работы со строками

Функция	Краткое описание

strcat	char *strcat(char *dest, const char *src); конкатенация (склеивание) строки src с dest
strncat	char *strncat(char *dest, const char *src, size_t maxlen); добавить не более n символов из src в dest
strchr	char *strchr(const char *s, int c); найти первое вхождение символа c в строку s и вернуть указатель на найденное
strrchr	char *strrchr(const char *s, int c); найти последнее вхождение символа c в строку s и вернуть указатель на найденное
strcmp	int strcmp(const char *s1, const char *s2); сравнить две строки. Возвращаемое значение меньше 0, если s1 лексикографически (алфавитно) предшествует s2, ноль, если s1=s2, больше нуля, если s1 больше s2
strncmp	int strncmp(const char *s1, const char *s2, size_t n); сравнить две строки, учитывая не более n символов
stricmp	int stricmp(const char *s1, const char *s2); сравнить две строки, считая латинские символы нижнего и верхнего регистров эквивалентными
strnicmp	int strnicmp(const char *s1, const char *s2, size_t n); то же, что stricmp и strncmp совместно
strcpy	char *strcpy(char *dest, const char *src); копировать строку src в dest, включая завершающий нулевой байт
strncpy	char *strncpy(char *dest, const char *src, size_t n); копировать не более n символов из src в dest
strdup	char *strdup(const char *s); дублирование строки с выделением памяти,

	например
	`char *dup_str, *string = "abcde";`
	`dup_str = strdup(string);`

`strlen`	`size_t strlen(const char *s);` возвращает длину строки в символах
`strlwr`	`char *strlwr(char *s);` преобразовать строку в нижний регистр (строчные буквы)
`strupr`	`char *strupr(char *s);` преобразовать строку в верхний регистр (заглавные буквы)
`strrev`	`char *strrev(char *s);` инвертировать (перевернуть) строку
`strnset`	`char *strnset(char *s, int ch, size_t n);` установить `n` символов строки `s` в заданное значение `ch`
`strset`	`char *strset(char *s, int ch);` установить все символы строки `s` в заданное значение `ch`

`strspn`	`size_t strspn(const char *s1, const char *s2);` ищет начальный сегмент `s1`, целиком состоящий из символов `s2`. Вернет номер позиции, с которой строки различаются, например:
	`char *string1 = "1234567890";`
	`char *string2 = "abc123";`
	`int l = strspn(string1, string2);`
	`printf("Position=%d\n", l); //l=3`

`strcspn`	`size_t strspn(const char *s1, const char *s2);` ищет начальный сегмент `s1`, целиком состоящий из символов, не входящих в `s2`. Вернет номер позиции, с которой строки различаются, например
	`char *string1 = "123abc7890";`
	`char *string2 = "abc";`
	`int l = strcspn(string1, string2);`
	`printf("Position=%d\n", l); //l=3`

`strstr`	`char *strstr(const char *s1, const`

	`char *s2);` найти первую подстановку строки `s2` в `s1` и вернуть указатель на найденное
`strtok`	`char *strtok(char *s1, const char *s2);` найти следующий разделитель из набора `s2` в строке `s1`
`strerror`	`char *strerror(int errnum);` сформировать в строке сообщение об ошибке, состоящее из двух частей: системной диагностики и необязательного добавочного пользовательского сообщения

Пример иллюстрирует действие функции `strtok`, позволяющей разбивать строку на лексемы:

```
  char input[16] = "abc,d";
  char *p;
/* strtok помещает нулевой байт  вместо
 разделителя лексем, если поиск был успешен */
  p = strtok(input, ",");
  if (p)   printf("%s\n", p); //"abc"
/* второй вызов использует NULL как
   первый параметр и возвращает
   указатель на следующую лексему */
  p = strtok(NULL, ",");
  if (p)   printf("%s\n", p); //"d"
```

Разумеется, число выделяемых лексем и набор разделителей могут быть любыми. В коде, приведенном ниже, лексемы могут разделяться пробелом, запятой или горизонтальной табуляцией, а прием и синтаксический разбор строк завершается при вводе пустой строки.

```
   #include <string.h>
   #include <stdio.h>
   #define BLANK_STRING ""
   void main(void){
   char *token, buf[81],*separators = "\t,. ";
   int i;
   puts("Вводите строки\n\
         Завершение - нажатие ENTER.\n");
```

```
while(strcmp(gets(buf),BLANK_STRING)!=0) {
 i = 0;
 token = strtok(buf, separators);
 while(token != NULL) {
  printf("Лексема %d - %s\n", i, token);
  token = strtok(NULL, separators);
  i++;
 }
}}
```

Следующий пример показывает простейшее использование обработчика ошибок `strerror`:

```
#include <stdio.h>
#include <errno.h>
//...
char *buffer;
buffer = strerror(errno);
printf("Error: %s\n", buffer);
```

8.3. Функции преобразования типов

Описанные в табл. 8.4 функции объявлены в стандартной библиотеке stdlib.h. Прототип функции `atof` содержится, кроме того, в файле math.h.

Таблица 8.4. Функции преобразования типов

Функция	Краткое описание
`atof`	`double atof(const char *s);` преобразование строки, в представляемое ей число типа `double`. На переполнение возвращает плюс или минус `HUGE_VAL` (константа из библиотеки)
`atoi`	`int atoi(const char *s);` преобразование строки в число типа `int` (целое). При неудачном преобразовании вернет `0`
`atol`	`long atol(const char *s);` преобразование строки в число типа `long` (длинное целое)
`ecvt`	`char *ecvt(double value, int ndig, int *dec, int *sign);` преобразование числа типа

	double в строку. ndig – требуемая длина строки, dec возвращает положение десятичной точки от 1-й цифры числа, sign возвращает знак
fcvt	char *fcvt(double value, int ndig, int *dec, int *sign); преобразование числа типа double в строку. В отличие от ecvt, dec возвращает количество цифр после десятичной точки. Если это количество превышает ndig, происходит округление до ndig знаков.
gcvt	char *gcvt(double value, int ndig, char *buf); преобразование числа типа double в строку buf. Параметр ndig определяет требуемое число цифр в записи числа.
itoa	char *itoa (int value, char *string, int radix); преобразование числа типа int в строку, записанную в системе счисления с основанием radix (от 2 до 36 включительно)
ltoa	char *ltoa (long value, char *string, int radix); преобразование числа типа long в строку
ultoa	char *ultoa (unsigned long value, char *string, int radix); преобразование числа типа unsigned long в строку

Некоторые системы программирования предоставляют также функции, перечисленные в табл. 8.5.

Таблица 8.5. Дополнительные функции преобразования типов

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

В приведенном далее коде число n преобразуется в строку s, представляющую собой запись числа в системе счисления с основанием r.

```
int n;
printf ("\nN="); scanf ("%d", &n);
char s[25];
int r;
do {
 printf ("\nRadix[2-36]: ");
 fflush (stdin); scanf ("%d",&r);
} while (r<2 || r>36);
s=itoa (n,s,r);
printf ("\n%d in %d notation is %s",n,r,s);
```

8.4. Математические функции

Прототипы математических функций содержатся в файле math.h, за исключением прототипов _clear87, _control87, _fpreset, status87, определенных в файле float.h.

Вещественные функции, как правило, работают с двойной точностью (тип double).

Многие функции имеют версии, работающие с учетверенной точностью (тип long double). Имена таких функций имеют суффикс "l" в конце (atan и atanl, fmod и fmodl и т. д.). Действие модификатора long в применении к типу double зависит от архитектуры ЭВМ.

Таблица 8.6. Математические функции

Функция	Краткое описание
abs	нахождение абсолютного значения выражения типа int
acos	вычисление арккосинуса. Аргументы этой и других тригонометрических функций задаются в радианах
asin	вычисление арксинуса
atan	вычисление арктангенса x

atan2	вычисление арктангенса от y/x
cabs	нахождение абсолютного значения комплексного числа
ceil	нахождение наименьшего целого, большего или равного x
_clear87	получение значения и инициализация слова состояния сопроцессора и библиотеки арифметики с плавающей точкой
_control87	получение старого значения слова состояния для функций арифметики с плавающей точкой и установка нового состояния
cos	вычисление косинуса
cosh	вычисление гиперболического косинуса
exp	вычисление экспоненты
fabs	нахождение абсолютного значения типа double
floor	нахождение наибольшего целого, меньшего или равного x
fmod	нахождение остатка от деления x/y
_fpreset	повторная инициализация пакета плавающей арифметики
frexp	вычисляет для x вещественную мантиссу m и целое n так, что $x=m*2^n$
hypot	вычисление гипотенузы
labs	нахождение абсолютного значения типа long
ldexp	вычисление $x*2^e$
log	вычисление натурального логарифма
log10	вычисление логарифма по основанию 10
matherr	управление реакцией на ошибки при выполнении функций математической библиотеки
modf	разложение x на дробную и целую часть
pow	вычисление x в степени y
sin	вычисление синуса
sinh	вычисление гиперболического синуса

sqrt	нахождение квадратного корня
_status87	получение значения слова состояния с плавающей точкой
tan	вычисление тангенса
tanh	вычисление гиперболического тангенса

В библиотеке определен также ряд констант, таких как M_PI (число π), M_E (основание натурального логарифма e) и др.

Функция matherr, которую пользователь может определить в своей программе, вызывается любой библиотечной математической функцией при возникновении ошибки. Эта функция определена в библиотеке, но может быть переопределена для установки различных процедур обработки ошибок.

```
int matherr (struct exception *a) {
 if (a->type == DOMAIN)
  if (!strcmp(a->name,"sqrt")) {
   a->retval = sqrt (-(a->arg1));
   return 1;
  }
 return 0;
}
double x = -2.0, y;
y = sqrt(x);
printf("Matherr corrected value: %lf\n",y);
```

8.5. Динамическое распределение памяти

Стандартная библиотека предоставляет механизм распределения динамической памяти (heap). Этот механизм позволяет динамически запрашивать из программы дополнительные области оперативной памяти.

Работа функций динамического распределения памяти различается для различных *моделей памяти*, поддерживаемых системой программирования (см. п. 11).

В малых моделях памяти (tiny**,** small**,** medium) доступно для использования все пространство между концом сегмента

статических данных программы и вершиной программного стека, за исключением 256-байтной буферной зоны непосредственно около вершины стека.

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

Как правило, функция выделения памяти возвращает нетипизированный указатель на начало выделенной области памяти. Для динамического распределения памяти используются функции, описанные в табл. 8.7. Их прототипы содержатся в файлах alloc.h и stdlib.h.

Таблица 8.7. Функции динамического распределения памяти

Функция	Краткое описание
`calloc`	`void *calloc(size_t nitems, size_t size);` выделяет память под `nitems` элементов по `size` байт и инициализирует ее нулями
`malloc`	`void *malloc(size_t size);` выделяет память объемом `size` байт
`realloc`	`void *realloc (void *block, size_t size);` пытается переразместить ранее выделенный блок памяти, изменив его размер на `size`
`free`	`void free(void *block);` пытается освободить блок, полученный посредством функции `calloc`, `malloc` или `realloc`

Приведем примеры использования функций из табл. 8.7.

```
char *str = NULL;
str = (char *) calloc(10, sizeof(char));
/* . . . */
char *str;
if ((str = (char *) malloc(10)) == NULL) {
 printf(
 "Not enough memory to allocate buffer\n");
```

```
 exit(1); }
/* . . . */
char *str;
str = (char *) malloc(20);
printf("String is %s\nAddress is %p\n",
 str, str);
str = (char *) realloc(str, 40);
printf("String is %s\nAddress is %p\n",
 str, str);
free (str);
```

Функции `calloc` и `malloc` выделяют блоки памяти, `malloc` при этом просто выделяет заданное число байт, а `calloc` выделяет и инициализирует нулями массив элементов заданного размера.

Ряд систем программирования предоставляют множество альтернативных функций, не входящих в ядро языка Си. Они кратко описаны в табл. 8.8.

Таблица 8.8. Другие функции распределения памяти

Функция	Краткое описание
`alloca`	выделение блока памяти из программного стека
`_expand`	изменение размера блока памяти, не меняя местоположения блока
`_ffree`	освобождение блока, выделенного посредством функции `_fmalloc`
`_fmalloc`	выделение блока памяти вне данного сегмента
`_freect`	определить примерное число областей заданного размера, которые можно выделить
`_fmsize`	возвращает размер блока памяти, на который указывает дальний (`far`) указатель
`halloc`	выделить память для большого массива (объемом более 64 Кб)
`hfree`	освободить блок памяти, выделенный посредством функции `halloc`
`_memavl`	определить примерный размер в байтах

	памяти, доступной для выделения
_msize	определить размер блока, выделенного посредством функций calloc, malloc, realloc
_nfree	освобождает блок, выделенный посредством _nmalloc
_nmalloc	выделить блок памяти в заданном сегменте
_nmsize	определить размер блока, на который указывает близкий (near) указатель
stackavail	определить объем памяти, доступной для выделения посредством функции alloca
brk	переустановить адрес первого байта оперативной памяти, недоступного программе (начала области памяти вне досягаемости программы)
allocmem	низкоуровневая функция выделения памяти
freemem	низкоуровневая функция возврата памяти операционной системе
coreleft	узнать, сколько осталось памяти для выделения в данном сегменте
farcalloc	выделить блок памяти вне данного сегмента
farcoreleft	определить, сколько памяти для размещения осталось вне данного сегмента
farmalloc	выделить блок памяти вне данного сегмента
farrealloc	изменить размер блока, ранее выделенного функцией farmalloc или farcalloc
farfree	освободить блок, ранее выделенный функцией farmalloc или farcalloc

Эти функции могут быть полезны для выделения памяти под данные объемом более сегмента (64 Кб), а также для определения объема свободной памяти.

Приведем более подробный пример, иллюстрирующий динамическое выделение памяти для вектора и последующую работу с этим вектором.

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

```
#include <alloc.h>
#include <stdlib.h>
void wait (void) {
 fflush (stdin); getchar(); }
void error (int n) {
 switch (n) {
  case 0: printf ("\n OK"); break;
  case 1: printf ("\n Bad size of array!");
  break;
  case 2: printf
   ("\n Can't allocate memory!"); break;
 }
 wait(); exit (n);
}
int *allocate (long n) {
 return (int *)malloc(n*sizeof(int));
}
void main(void) {
 long n=0;
 printf ("\n Enter number of items:");
 scanf ("%ld",&n);
 if (n<2) error (1);
 int *p=allocate(n);
 if (p==NULL) error (2);
 for (long i=0; i<n; i++) {
  *(p+i)=random(n);
  printf ("%6ld",*(p+i));
 }
 wait ();
 free (p);
}
```

При работе с динамической матрицей следует сначала выделить память под массив указателей на строки, а затем каждый из этих указателей связать с динамически выделенной областью, куда запишутся элементы строк. Приведем только измененные фрагменты предыдущего примера.

```
int **allocatematrix (int n, int m) {
```

```
 //выделяет память для матрицы n*m
 int **p=NULL;
 p=(int **)malloc(n*sizeof(int *));
 if (p==NULL) error (2);
 for (int i=0; i<n; i++) {
  p[i]=(int *)malloc(m*sizeof(int));
  if (p[i]==NULL) error (2);
 }
 return p;
}
/* . . . */
int n=0,m=0;
printf ("\n Enter number of items:");
scanf ("%d %d",&n,&m);
if (n<2 || m<2) error (1);
int **p=allocatematrix(n,m);
for (int i=0; i<n; i++) {
 printf ("\n ");
 for (int j=0; j<m; j++) {
  p[i][j]=i+j; printf ("%4d ",p[i][j]);
}}
```

Правильный порядок освобождения занятой памяти для динамической матрицы будет таков:

```
for (long i=n-1; i>-1; i--) free (p[i]);
free (p);
```

Ввод значений многомерных вещественных массивов с клавиатуры также требует осторожности. Код вида

```
scanf ("%f",&p[i][j]);
```

следует заменять на

```
float f;  scanf ("%f",&f);  p[i][j]=f;
```

8.6. Функции стандартного ввода и вывода

Все функции форматированного ввода и вывода используют в качестве первого параметра строку формата, управляющую способом преобразования данных. В строке формата символы после % и до первого разделителя рассматриваются как *спецификация преобразования значений*

выводимой переменной (или просто *форматная спецификация*). Количество спецификаций в строке формата должно совпадать с количеством вводимых или выводимых значений, передаваемых, начиная со второго параметра функции.

Сначала рассмотрим подробнее структуру форматной спецификации. Она имеет следующий общий вид:

```
%<флаги><ширина><.точность><модификатор>тип
```

В символы <> заключены необязательные элементы спецификации, таким образом, единственным обязательным элементом является тип. Символы управления форматированием, составляющие различные элементы спецификации, описаны в табл. 8.9.

Таблица 8.9. Символы управления форматированием

Элемент спецификации	Действие
флаги	
-	выравнивание числа влево в пределах выделенного поля, справа дополняется пробелами. По умолчанию устанавливается выравнивание влево
+	выводится знак числа символом "-" или "+"
пробел	перед положительным числом выводится пробел, для отрицательных всегда выводится знак "-"
#	для целых чисел выводится идентификатор системы счисления: 0 перед числом для вывода в 8-ричной с.с. 0x или 0X перед числом для вывода в 16-ричной с.с. ничего для чисел, выводимых в 10-ной с.с. Выводится десятичная точка для чисел типа float
ширина – воздействует только на вывод	
n	целое n определяет минимальную ширину поля в n символов. Если этой ширины

	недостаточно, выводится столько символов, сколько есть. Незаполненные позиции дополняются пробелами.
0n	то же, что n, но позиции слева для целого числа дополняются нулями
*	следующий аргумент из списка аргументов задает ширину
точность – воздействует только на вывод	
ничего	точность по умолчанию
.0	для d,i,o,u,x точность по умолчанию. Для e,E,f десятичная точка отсутствует
.n	Для e,E,f не более n знаков после точки
*	следующий аргумент из списка аргументов задает точность
модификатор – действует там, где применим	
h	для d, i, o, u, x, X аргумент является short int
l	для d, i, o, u, x, X аргумент является long int для e, E, f, F, g, G аргумент является double (обычно работает только для scanf)
тип переменной	
c	Тип char При вводе читается и передается 1 байт. При выводе переменная преобразуется к типу char. В файл передается 1 байт
 d i o u x X	Тип int десятичное целое со знаком десятичное целое со знаком 8-ричное целое без знака 10-ное целое без знака 16-ричное целое без знака. При выводе использует a…f 16-ричное целое без знака. При выводе использует A…F При вводе действие x и X не различается
 f	Тип float число со знаком в формате <->dddd.ddd

e E g G	число со знаком в формате <->dddd.ddde<+ или ->ddd число со знаком в формате <->dddd.dddE<+ или ->ddd При вводе e и E не различаются число со знаком в формате e или f в зависимости от указанной точности число со знаком в формате E или F в зависимости от указанной точности При вводе действие g и G не различается
s p	Тип char * При вводе принимает символы без преобразования, пока не встретится разделитель '\n' или не достигнута указанная точность. В программу передаются символы до '\n' или пробела. При выводе выдает в поток все символы, пока не встретится '\0' или не достигнута указанная точность Выводит аргумент как адрес, формат зависит от модели памяти, в общем случае, включает 16-ричные сегмент и смещение

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

Функции ввода/вывода делятся на три класса:

1. *Ввод/вывод верхнего уровня* (с использованием понятия "поток"). Для этих функций характерно, что они обеспечивают *буферизацию* работы с файлами. Это означает, что при чтении или записи информации обмен данными осуществляется не между программой и указанным файлом, а между программой и промежуточным буфером, расположенным в оперативной памяти.

При записи в файл информация из буфера записывается при его заполнении или при закрытии файла. При чтении данных программой информация берется из буфера, а в буфер она считывается при открытии файла и впоследствии каждый раз при опустошении буфера. Буферизация ввода/вывода выполняется автоматически, она позволяет ускорить выполнение программы за счет уменьшения количества обращений к сравнительно медленно работающим внешним устройствам.

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

Когда поток открывается для ввода/вывода, он связывается со структурой типа `FILE`, определенной с помощью `typedef` в файле stdio.h. Эта структура содержит необходимую информацию о файле. При открытии файла с помощью стандартной функции `fopen` возвращается указатель на структуру типа `FILE`. Этот указатель, называемый *указателем потока*, используется для последующих операций с файлом. Его значение передается всем библиотечным функциям, используемым для ввода/вывода через данный поток.

Функции верхнего уровня одинаково работают в различных операционных средах, с их помощью можно писать переносимые программы.

2. *Ввод/вывод для консольного терминала* путем непосредственного обращения к нему. Функции для консоли и порта распространяют возможности функций ввода/вывода верхнего уровня на соответствующий класс устройств, добавляя также новые возможности.

Функции этого класса позволяют читать или записывать на консоль (терминал) или в порт ввода/вывода (например, порт принтера). Они обрабатывают данные побайтно. Для ввода или

вывода с консоли устанавливаются некоторые дополнительные режимы, например, ввод с эхо-печатью символов или без нее, установка окна вывода, цветов текста и фона. Функции для консоли и порта являются уникальными для компьютеров, совместимых с IBM-PC.

3. *Ввод/вывод низкого уровня* (с использованием понятия "дескриптор"). Функции низкого уровня не выполняют буферизацию и форматирование данных, они позволяют непосредственно пользоваться средствами ввода/вывода операционной системы.

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

Функции низкого уровня из стандартной библиотеки обычно применяются при разработке собственных подсистем ввода/вывода. Большинство функций этого уровня переносимы в рамках некоторых систем программирования на Си, в частности, относящихся к операционной системе Unix и совместимым с ней.

8.7. Функции ввода/вывода высокого уровня

Прототипы всех функций, описанных в табл. 8.10, содержатся в файле stdio.h.

Таблица 8.10. Функции в/в высокого уровня

Функция	Краткое описание
`clearerr`	`void clearerr(FILE *stream);` очистка флага ошибки для потока `stream`
`fclose`	`int fclose(FILE *stream);` закрытие потока. Возвращает `0` в случае успеха, `EOF` если обнаружена ошибка
`fcloseall`	`int fcloseall(void);` закрытие всех открытых (на верхнем уровне) файлов.

	Возвращаемые значения: успех — общее число закрытых потоков, ошибка — EOF
feof	int feof(FILE *stream); проверка на конец потока (не 0, если достигнут)
ferror	int ferror(FILE *stream); проверка флажка ошибок потока (не 0 – ошибка)
fflush	int fflush(FILE *stream); сброс буфера потока на связанное с ним внешнее устройство. Успех – вернет 0, ошибка – вернет EOF
fgetc	int fgetc(FILE *stream); чтение символа из потока. На конец файла или ошибку возвращает EOF
fgets	char *fgets(char *s, int n, FILE *stream); чтение строки из потока (до перевода строки или достижения n-1 символов). Сохраняет символ новой строки ее последним байтом. Добавляет после него нулевой байт. В случае успеха возвращает указатель на прочитанную строку, в случае ошибки или конца файла – NULL
flushall	int flushall(void); сброс буферов всех потоков. Возвращает целое число открытых потоков ввода и вывода
fopen	FILE *fopen(const char *filename, const char *mode); открытие потока (открыть файл и связать его с потоком). Возвращает указатель на структуру FILE или NULL в случае ошибки. Строковая константа mode может иметь следующие значения: r открыть для чтения; w открыть для записи, существующий файл, если он есть, обнуляется; a открыть для записи в конец существующего файла. Если файл не существует, работает как

	"w";
	r+ открыть существующий файл на чтение и запись;
	w+ создать новый файл для чтения и записи. Если файл с таким именем уже существует, он будет обнулен;
	a+ открыть для добавления или создать файл, если он не существует.
	Для указания на текстовый тип файла после любого из этих значений может быть добавлен символ "t", для указания на бинарный файл "b". В последнем случае следует работать с файлом методами fread и fwrite, а не использовать функции чтения/записи строк. Пример:
	`FILE *f=fopen ("file.txt","r+t");`
	`if (f==NULL) { //обработка ошибки`
	`}`

fprintf	`int fprintf(FILE *stream, const char *format <, argument, ...>);` Запись данных в поток по формату. Число аргументов, начиная с третьего, зависит от числа элементов спецификации в строке формата и должно совпадать с ним. В случае успеха вернет число записанных байт, в случае ошибки – EOF

fputc	`int fputc(int c, FILE *stream);` запись символа с в поток. Пример:
	`char msg[] = "Hello world";`
	`int i = 0;`
	`while (msg[i]) {`
	`fputc(msg[i], stdout); i++;`
	`}`

fputs	`int fputs(const char *s, FILE *stream);` запись строки, завершающейся нулевым байтом, в поток. Не добавляет символ новой строки и не копирует нулевой байт. В случае успеха возвращает последний записанный

	символ, в случае ошибки вернет EOF. Пример: `fputs("Hello world\n", stdout);`
fread	`size_t fread(void *ptr, size_t size, size_t n, FILE *stream);` неформатированное чтение данных из потока. Читает из потока stream n элементов данных по size байт каждый в память, на которую указывает ptr. Общее число прочитанных байт будет равно n*size. В случае успеха возвращает число прочитанных элементов (не байт!), в случае ошибки – 0 (точно не определено в спецификации)
freopen	`FILE *freopen(const char *filename, const char *mode, FILE *stream);` повторное открытие потока в новом режиме mode. Вернет указатель на поток или NULL в случае ошибки
fscanf	`int fscanf(FILE *stream, const char *format<, address, ...>);` чтение из потока по формату. Требования к аргументам совпадают с fprintf, все аргументы, начиная с третьего, должны быть адресами. В случае успеха вернет число успешно прочитанных, преобразованных и сохраненных полей ввода, вернет 0, если не удалось сохранить ни одно поле, вернет EOF при попытке чтения за пределами конца файла или строки (для sscanf)
fseek	`int fseek(FILE *stream, long offset, int whence);` Перемещение указателя файла в заданную позицию offset. Для текстового режима offset=0 или позиции, возвращаемой ftell. Для бинарного режима возможны и отрицательные значения. Константа whence определяет, откуда отсчитывать смещение: значение SEEK_SET – от начала

	файла; SEEK_CUR – от текущей позиции файлового указателя; SEEK_END – от конца файла. Вернет 0, если указатель успешно перемещен, в противном случае – ненулевую величину. Аналог: int fsetpos (FILE *stream. long offset); всегда считает смещение с начала файла. Успех – 0, иначе не-ноль.
ftell	long int ftell(FILE *stream); получение текущей позиции указателя файла. Если файл бинарный, позиция отсчитывается от начала файла, начиная с 0. В случае ошибки вернет -1. Аналог: int fgetpos (FILE *stream. fpos_t *pos); запишет текущую позицию файлового указателя в pos. Тип fpos_t здесь и далее определен как long. в случае успеха вернет 0, иначе не-ноль.
fwrite	size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream); неформатированная запись данных в поток. Добавляет в поток stream n элементов данных по size байт каждый. Общее число записываемых байт = n*size. В случае успеха вернет число элементов (не байт!), записанных в поток, в случае ошибки — меньшее, чем n значение.
getc	int getc(FILE *stream); чтение символа из потока (реализуется через макроопределение). В случае успеха вернет прочитанный символ после преобразования к int без знака. В случае ошибки или достижения конца файла вернет EOF
getchar	int getchar(void); чтение символа из потока stdin (версия макро). Работает как getc

	для потока stdin
gets	char *gets(char *s); чтение строки из потока stdin. Заменяет символ конца строки нулевым байтом. Все символы, включая перевод строки, пишутся в строку s и возвращается указатель на нее. На конец файла или ошибку возвращает NULL
printf	int printf(const char *format<, argument, ...>); запись данных в поток stdout по формату. Требования – см. fprintf. В случае успеха вернет число записанных байт, в случае ошибки – EOF
putc	int putc(int c, FILE *stream); запись символа в поток stream (версия макро). Возвращает записанный символ c. В случае ошибки вернет EOF
putchar	int putchar(int c); запись символа в поток stdout (версия макро). Возвращает записанный символ c.В случае ошибки вернет EOF
puts	int puts(const char *s); запись строки s в поток stdout. Добавляет символ перевода строки. В случае успеха вернет неотрицательное число, на ошибку вернет EOF
scanf	int scanf(const char *format<, address, ...>); чтение данных из потока stdin по формату. Требования те же, что для fscanf
setbuf	void setbuf(FILE *stream, char *buf); управление буферизацией потока (назначить буфер потоку). Пример: char outbuf[BUFSIZ]; setbuf (stdout, outbuf);
sprintf	int sprintf(char *buffer, const

	char *format<, argument, ...>); запись данных в строку buffer по формату. Требования те же, что к fprintf. В случае успеха вернет число записанных байт, в случае ошибки – EOF. Не включает нулевой байт в число записанных символов.
sscanf	int sscanf (const char *buffer, const char *format<, address, ...>); чтение данных из строки buffer по формату. Требования те же, что к fscanf.
ungetc	int ungetc(int c, FILE *stream); вернуть символ c в буфер потока. Поток должен быть доступен для чтения. Символ будет прочитан из потока при следующем вызове getc или fread для этого потока. Возвращение одного символа безопасно, повторный вызов без обращения к функциям чтения из потока приведет к потере предыдущего символа. Вернет возвращенный символ (успех) или EOF (ошибка).

В случае ошибки ряд функций устанавливает значение глобальной переменной errno из библиотеки errno.h.

Некоторые константы, определенные в stdio.h, могут быть полезны в программе. Они описаны в табл. 8.11.

Таблица 8.11. Константы библиотеки stdio.h

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

Для каждой выполняемой программы автоматически открываются пять потоков: стандартный ввод (`stdin`), стандартный вывод (`stdout`), стандартный вывод для сообщений об ошибках (`stderr`), стандартный последовательный порт (`stdaux`) и стандартное устройство печати (`stdprn`). По умолчанию первые три потока связаны с консольным терминалом, стандартными портами обычно служат последовательным порт и принтер. В файле stdio.h определены соответствующие указатели стандартных потоков:

```
extern FILE * stdin;
```

Ввод и вывод программы могут быть переопределены, например, средствами консоли Windows (символы перенаправления в/в <, > или >>), можно переопределять стандартные потоки также с помощью функции `freopen`.

В отличие от стандартных потоков, открытые файлы, для которых осуществляется высокоуровневый ввод/вывод, буферизуются по умолчанию. Для управления буферизацией служат функции `setbuf` и `setvbuf`. Их использованием можно сделать поток небуферизованным или связать буфер с небуферизованным до этого потоком. Буферы, размещенные в системе, недоступны пользователю, кроме тех, что получены с помощью `setbuf` или `setvbuf`. Буферы должны иметь постоянный размер, равный константе BUFSIZ из stdio.h.

Буферы можно сбросить в произвольный момент времени, используя функции `fflush` и `flushall`. Например, это имеет смысл делать перед каждым вводом с консоли с помощью функции `scanf`.

Потоки, с которыми завершена работа, следует закрыть функцией `fclose` или `fcloseall`. Стандартные потоки при этом не закрываются. Потоки, открытые программой, автоматически закрываются при ее завершении, однако, это может привести к потере данных при записи в поток, а также превышению предельного количества одновременно открытых потоков.

Работа с потоками происходит следующим образом. Функция открытия потока возвращает указатель на тип `FILE`, который используется при дальнейших обращениях к потоку.

Операции чтения и записи начинаются с текущей позиции в потоке, определяемой как *указатель файла* (англ. file pointer). Указатель файла изменяется после каждой операции чтения или записи. Например, при чтении символа из потока указатель продвигается на 1 байт, так что следующая операция начнется с первого несчитанного символа. Если поток открыт для добавления, указатель файла автоматически устанавливается на конец файла перед каждой операцией записи. Если поток не имеет указателя файла (например, консоль), функции позиционирования в файле (`fsetpos`, `fseek`) имеют неопределенный результат.

При ошибке в операции с потоком для него устанавливается в ненулевое значение флаг ошибки. Для определения того, произошла ли ошибка можно использовать макроопределение `ferror`. Флаг ошибки остается установленным до тех пор, пока не будет сброшен вызовом функции `clearerr` или `rewind`.

Перейдем к примерам. Следующий листинг иллюстрирует открытие файла для чтения, проверку ошибок системными средствами и посимвольное чтение текстовых данных из файла.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(void){
 FILE *fp;
 char ch,*s;
 fp = fopen("c:\\test.txt", "rt");
 if (ferror(fp)) { //тест потока на ошибку
  sprintf (s,"Error %d",errno);
  //встроенная переменная errno – номер ошибки
  perror (s); //вызов системной печати
              //сообщения об ошибке
  fflush (stdin); getchar();
  clearerr (fp); //сброс errno
```

```
  exit (errno);
 }
 while (!feof(fp)) {
  ch = fgetc(fp);
  printf("%c",ch);
 }
 fflush (stdin); getchar();
 fclose(fp);
 return 0;
}
```

Во втором примере выполняется двоичная запись в файл, а проверка ошибок упрощена так, как чаще всего делается на практике — для определения того, удалось ли открыть файл, файловая переменная `f` сравнивается со значением `NULL`.

```
#include<stdio.h>
void error (int n) {
 /* метод для обработки ошибок */
}
void main(void) {
 FILE *f;
 int k,i[]={1,2,3,4,5};
 f=fopen("data.txt","r+b");
 if (f==NULL) error (1);
  //"Не могу открыть файл"
 fseek (f,0L,SEEK_END);
 long n;
 fgetpos (f,&n);
 if (n>0) error (2); //Файл не пуст
 for(k=0;k<5;k++)
  fwrite(&i[k],sizeof(int),1,f);
 fclose(f);
}
```

8.8. Функции ввода/вывода нижнего уровня

Функции нижнего уровня не требуют включения файла stdio.h. Тем не менее, несколько общих констант, определенных в этом файле, могут оказаться полезными (например, признак

конца файла EOF). Прототипы функций нижнего уровня содержатся в файле io.h.

В настоящее время низкоуровневый доступ к файлам имеет ограниченное применение, такое как написание системных функций и драйверов для устройств, программируемых на Си.

В табл. 8.12 кратко перечислены функции для низкоуровневого доступа к файлам.

Таблица 8.12. Низкоуровневый доступ к файлам

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

8.9. Функции ввода/вывода с консольного терминала

Функции ввода/вывода для консоли используют специфические особенности IBM-совместимого компьютера, такие как наличие специального видеоадаптера, и не являются переносимыми на другие типы компьютеров. Прототипы функций содержатся в файле conio.h. Консольные функции позволяют читать и записывать строки (cgets и cputs), форматированные данные (cscanf и cprintf) и отдельные символы. Функция kbhit определяет, было ли нажатие клавиши и позволяет определить наличие символов для ввода с клавиатуры до попытки чтения.

Во-первых, существуют функции для работы с окном консоли, аналогичные библиотеке `Crt` Паскаля. В табл. 8.13 кратко перечислены основные из них.

Таблица 8.13. Функции для работы с окном консоли

Функция	Краткое описание
`window`	`void window(int left, int top, int right, int bottom);` Устанавливает текущее окно консоли по указанным координатам
`clrscr`	`void clrscr(void);` очищает текущее окно
`clreol`	`void clreol(void);` очищает текущую строку окна от позиции курсора до конца
`delline`	`void delline(void);` удаляет строку окна, в которой установлен курсор
`insline`	`void insline(void);` вставляет пустую строку в позиции курсора
`gotoxy`	`void gotoxy(int x, int y);` перемещает курсор в указанные столбец (`x`) и строку (`y`) окна
`textbackground`	`void textbackground(int newcolor);` устанавливает указанный фоновый цвет окна. Цвета указываются номерами 0-15 или названиями, определенными в conio.h (`BLUE`, `GREEN` и т. д.)
`textcolor`	`void textcolor(int newcolor);` устанавливает указанный цвет вывода текста в окне. Цвета указываются аналогично функции `textbackground`
`wherex`	`int wherex(void);` возвращает

	номер столбца окна, в котором находится курсор
`wherey`	`int wherey(void);` возвращает номер строки окна, в которой находится курсор

Во-вторых, в файле conio.h описаны прототипы ряда специфичных для Си функций (см. табл. 8.14).

Таблица 8.14. Функции ввода/вывода для консоли

Функция	Краткое описание
`cgets`	`char *cgets(char *str);` чтение строки с консоли до комбинации CR/LF или достижения максимально возможного числа символов. Если cgets читает комбинацию CR/LF, она заменяет ее на нулевой байт перед сохранением строки. Перед вызовом функции `str[0]` должен быть установлен в максимально возможное число символов для чтения. В случае успеха `str[1]` содержит реально прочитанное число символов, возвращается указатель на `str[2]`. При чтении комбинации CR/LF она заменяется нулевым байтом. Нулевой байт добавляется в конец строки. Таким образом, длина буфера должна быть не меньше `str[0]+2` байт.
`cprintf`	`int cprintf(const char *format[, argument, ...]);` запись данных на консольный терминал по формату. Требования к строке формата и последующим аргументам аналогичны функции `fprintf`. Не переводит '`\n`' (символ LF) в пару символов '`\r\n`' (CR/LF). В случае успеха вернет число выведенных символов, в случае ошибки – `EOF`.
`cputs`	`int cputs(const char *str);` вывод строки в текущее окно консольного терминала,

	определенное по умолчанию или функцией window. Не добавляет символов новой строки.
getch	`int getch(void);` чтение символа с консоли. Символ не отображается на экране (нет эхо-печати). Вернет код символа.
getche	`int getche(void);` чтение символа с консоли с эхо-печатью. Вернет код символа.

kbhit	`int kbhit(void);` проверка нажатия клавиши на консоли. Пример:
	`while (!kbhit())`
	`  /*до нажатия клавиши */ ;`
	`cprintf("\r\nНажата клавиша...");`

putch	`int putch(int c);` Вывод символа на консольный терминал. Не переводит '\n' (символ LF) в пару символов '\r\n' (CR/LF). В случае успеха вернет символ, в случае ошибки – EOF
ungetch	`int ungetch(int ch);` возврат последнего прочитанного символа с консольного символа обратно с тем, чтобы он стал следующим символом для чтения. Ограничения те же, что для ungetc.

Приведенный ниже пример демонстрирует чтение строки с помощью функции cgets.

```
char buffer[83];
char *p;
/* Место для 80 символов + нулевого байта */
buffer[0] = 81;
printf("Введите строку:");
p = cgets(buffer);
printf("\ncgets прочитала %d \
          символов:\"%s\"\n", buffer[1], p);
printf("Возвращен указатель %p, \
      buffer[0] содержит %p\n", p, &buffer);
```

В следующем примере строка текста печатается в центре стандартного окна консоли.

```
#include <conio.h>
int main(void){
    clrscr();
    gotoxy(35, 12);
    textcolor (RED);
    cprintf("Hello world");
    getch();
    return 0;
}
```

Наконец, приведенный далее код демонстрирует обработку нажатий клавиш для консольного терминала.

```
#include <stdio.h>
#include <conio.h>
int main(void) {
 int ch;
 do {
  ch=getch();
  if (ch=='\0') { //расширенный код
   ch=getch();
   switch (ch) {
    case 72: printf ("\nUp");    break;
    case 80: printf ("\nDown");  break;
    case 75: printf ("\nLeft");  break;
    case 77: printf ("\nRight"); break;
    default: printf
      ("\nExtended key %2d",ch); break;
   }
  }
  else {
   switch (ch) {
    case 13: printf ("\nEnter"); break;
    case 27: printf ("\nEsc"); break;
    default: printf ("\nKey %4d",ch);
    break;
   }
  }
 } while (ch!=27); }
```

8.10. Работа с каталогами файловой системы

Прототипы функций, описанных в табл. 8.15, содержатся в стандартном заголовочном файле dir.h. Не все функции описаны в стандарте языка Си, но обычно все они предоставляются комилятором.

Таблица 8.15. Функции для работы с каталогами

Функция	Краткое описание
`chdir`	`int chdir(const char *path);` изменение текущего каталога, заданного полным или относительным путем `path`. Возвращает 0 в случае успеха, -1 в случае ошибки
`getcwd`	`char *getcwd(char *buf, int buflen);` получить имя текущего рабочего каталога. В случае успеха `buf!=NULL`. Ограничения, такие как `MAXPATH` (макс. длина пути к каталогу) определены в <dir.h>. Аналог — `int getcurdir (int drive, char *directory);` Диск по умолчанию кодируется как 0, 1 обозначает диск `A:` и т. д.
`mkdir`	`int mkdir(const char *path);` пытается создать новый каталог, заданный полным или относительным путем `path`. В случае успеха вернет 0, иначе -1
`rmdir`	`int rmdir(const char *path);` удаление каталога, заданного путем `path`. В случае успеха вернет 0, иначе -1
`findfirst`	`int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);` начинает поиск файла по шаблону имени `pathname`. для работы с поиском определен структурный тип `ffblk`: `struct ffblk {`

	long ff_reserved;
	long ff_fsize; //размер файла
	unsigned long ff_attrib;//атрибуты
	unsigned short ff_ftime; //время
	unsigned short ff_fdate; // дата
	char ff_name[256]; //найденное имя
	};
	Атрибуты определены в библиотеке dos.h:
	FA_RDONLY Read-only
	FA_HIDDEN Hidden
	FA_SYSTEM System
	FA_LABEL Volume label
	FA_DIREC Directory
	FA_ARCH Archive
	Поле ff_ftime определяет время следующим образом: биты от 0 до 4 обозначают секунды, деленные на 2, биты 5-10 – минуты, биты 11-15 – часы.
	Поле ff_fdate определяет дату следующим образом: биты 0-4 обозначают день месяца, биты 5-8 – номер месяца, биты 9-15 — год с 1980 (например, 9 означает 1989). В случае успеха вернет 0, иначе -1

findnext	int findnext(struct ffblk *ffblk); продолжает поиск файла по ранее указанному для findfirst шаблону имени
fnmerge	void fnmerge(char *path, const char *drive, const char *dir, const char *name, const char *ext); создает полное имя файла из отдельных компонент
fnsplit	int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext); разбивает полное имя файла на

	отдельные компоненты. Вернет целое число, биты которого показывают наличие отдельных компонент
`getdisk`	`int getdisk(void);` возвращает номер текущего диска. 0 обозначает диск `A:`, 1 — `B:` и т. д.
`searchpath`	`char *searchpath(const char *file);` выполняет поиск файла в каталогах, перечисленных в системной переменной PATH. Если файл найден, вернет полный путь к нему в статическом буфере (обновляется при каждом вызове функции)
`setdisk`	`int setdisk(int drive);` задает текущее дисковое устройство. Диски обозначаются как для `getdisk`. Возвращает общее число дисков, доступных в системе

Приведенный далее пример формирует листинг из всех файлов текущего каталога.

```
#include <stdio.h>
#include <dos.h>
#include <dir.h>
int main(void) {
   struct ffblk ffblk;
   int done;
   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,
    FA_ARCH|FA_DIREC);
   while (!done) {
    unsigned t=ffblk.ff_ftime;
    char s=(t&0x001F)<<1, m=(t&0x07E0)>>5,
     h=(t&0xF800)>>11;
    t=ffblk.ff_fdate;
    char d=(t&0x001F), mon=(t&0x01E0)>>5;
    int y=(t&0xFE00)>>9;
     printf("%s %15ld \
```

```
%02d/%02d/%4d,%02d:%02d:%02d\n",
        ffblk.ff_name,ffblk.ff_fsize,
        d,mon,y+1980,h,m,s);
      done = findnext(&ffblk);
     }
   getchar(); return 0;
}
```

В следующем примере иллюстрируется работа с функцией `fnsplit`.

```
#include <stdlib.h>
#include <stdio.h>
#include <dir.h>
    char *s;
    char drive[MAXDRIVE];
    char dir[MAXDIR];
    char file[MAXFILE];
    char ext[MAXEXT];
    int flags;
    s=getenv("COMSPEC");
    flags=fnsplit(s,drive,dir,file,ext);
    printf("Command processor info:\n");
    if(flags & DRIVE)
       printf("\tdrive: %s\n",drive);
    if(flags & DIRECTORY)
       printf("\tdirectory: %s\n",dir);
    if(flags & FILENAME)
       printf("\tfile: %s\n",file);
    if(flags & EXTENSION)
       printf("\textension: %s\n",ext);
```

8.11. Операции над файлами

В табл. 8.16 описаны основные стандартные функции, служащие для работы с файлами. В левом столбце таблицы под именем функции указаны имена библиотечных файлов, содержащих прототипы.

Таблица 8.16. Функции для работы с файлами

Функция	Краткое описание
access <io.h>	int access(const char *filename, int amode); определение прав доступа к файлу. Допустимы значений amode=06 (проверка на чтение и запись), 04 (чтение), 02 (запись), 01 (выполнение), 00 (проверка на существование файла)
chmod <io.h>	int chmod(const char *path, int amode); изменение прав доступа к файлу. Допустимые значения amode определены в sys\stat.h: S_IWRITE (разрешение на запись), S_IREAD (на чтение), S_IREAD | S_IWRITE (то и другое). Вернет 0 в случае успеха, -1 при ошибке

mktemp <dir.h>	char *_mktemp(char *template); генерация уникального имени файла. Пример:
	char *fname = "TXXXXXX", *ptr;
	ptr = mktemp(fname);

remove <stdio.h>	int remove(const char *filename); удаление файла. Вернет 0 в случае успеха, -1 при ошибке
rename <stdio.h>	int rename(const char *oldname, const char *newname); переименование файла из oldname в newname. Вернет 0 в случае успеха, -1 при ошибке

8.12. Использование вызовов операционной системы

Для доступа к функциям операционных систем, совместимых с MS-DOS предназначена библиотека dos.h. Можно выделить следующие основные группы функций этой библиотеки:

- Обработка прерываний (int86, int86x, getinterrupt, disable, enable, getvect, setvect);
- Работа с сегментами памяти (peek, poke, MK_FP);
- Доступ к секторам диска (absread, abswrite);

- Доступ к файловым таблицам FAT (`getfat`, `getfatd`, `getdfree`);
- Работа с портами (`inport`, `outport`).

Компилятор может предоставлять также возможности для обращения к базовой подсистеме ввода/вывода операционной системы BIOS. Основные функции кратко перечислены в табл. 8.17. Их прототипы содержатся в файле bios.h.

Таблица 8.17. Основные функции для работы с BIOS

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

8.13. Управление процессами

Функции этой группы позволяют приложению выполнять дочерние процессы, программно выполнять команды операционной системы, а также вырабатывать код завершения. Прототипы функций управления процессами объявлены в файле process.h. Основные функции описаны в табл. 8.18.

Таблица 8.18. Функции для управления процессами

Функция	Краткое описание
`abort`	`void abort(void);` прерывает текущий процесс (прототип содержится также в stdlib.h)
`exit`	`void exit(int status);` завершает процесс с кодом `errorlevel=status` (прототип есть также в stdlib.h)
`execl`	`int execl(char *path, char *arg0, *arg1, ..., *argn, NULL);` позволяет

	выполнить порождаемый процесс со списком аргументов. Существуют разновидности функции для передачи процессу параметров командной строки, использования PATH, системного окружения и т. п.
`spawnl`	`int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);` позволяет выполнить порождаемый процесс со списком аргументов. Величина mode определяет режим запуска дочернего процесса
`system`	`int system(const char *command);` выполнение команды ОС

8.14 Поиск и сортировка

Си содержит ряд стандартных функций, позволяющих упростить задачи сортировки и поиска данных. Их прототипы описаны в библиотеке stdlib.h. Библиотечные функции, предназначенные для поиска и сортировки в массиве, перечислены в табл. 8.19.

Таблица 8.19. Функции поиска и сортировки

Функция	Краткое описание
`bsearch`	Выполняет бинарный поиск в массиве
`lfind`	выполняет линейный поиск для заданного значения
`lsearch`	выполняет линейный поиск для заданного значения, которое добавляется в массив, если не найдено
`qsort`	выполняет быструю сортировку

8.15. Функции для работы с датой и временем

Прототипы этой группы функций описаны в библиотеках dos.h и time.h. Функции описаны в табл. 8.20.

Таблица 8.20.. Функции для работы с датой и временем

Функция	Краткое описание

getdate	void getdate(struct date *datep); получает системную дату как структуру типа date со следующими полями:
	struct date{
	int da_year; //год
	char da_day; //день месяца
	char da_mon; //месяц (1= январь)
	};

gettime	void gettime(struct time *timep); получает системное время как структуру типа time со следующими полями:
	struct time {
	unsigned char ti_min; //минуты
	unsigned char ti_hour; //часы
	unsigned char ti_hund;
	//сотые доли секунды
	unsigned char ti_sec; //секунды
	};

setdate	void setdate(struct date *datep); устанавливает системную дату
settime	void settime(struct time *timep); устанавливает системное время
time	long time(long *timer); получает системное время в "тиках" — секундах с начала "эры Unix" 1 января 1970 г. в полночь по Гринвичу

9. Составные типы данных

В этом разделе рассматриваются модифицируемые, перечислимые и структурные типы данных языка Си.

9.1. Имена с модификаторами

Использование модификаторов при описании данных позволяет придавать объявлениям специальный смысл.

Информация, которую несут модификаторы, используется компилятором при генерации кода.

Модификаторы `cdecl`, `pascal`, `interrupt` воздействуют на идентификатор и могут быть записаны непосредственно рядом с ним:

```
<модификатор> тип список;
тип <модификатор> список;
```

В ряде случаев допускаются оба варианта.

Модификаторы `const`, `volatile`, `near`, `far`, `huge` воздействуют либо на идентификатор, либо на указатель, расположенный непосредственно справа. Если справа расположен идентификатор, модифицируется тип именуемого объекта. Если справа расположен признак указателя "`*`", то преобразуется *тип объекта*, адресуемого указателем на модифицированный тип. Таким образом, конструкция "`модификатор *`" обозначает указатель на модифицированный тип.

Например, `int const *t;` - это указатель на `const int`, который можно переназначать, тогда как `int * const t;` — это `const` указатель на `int` и переназначить его нельзя.

Указание `const` не допускает действий по изменению значения переменной. Приведем пример.

```
char *const str = "Строка текста";
str = "Другая строка";
```

Последний оператор недопустим, т. к. пытается переназначить `const` указатель. Тем не менее, допустим оператор

```
strcpy(str,"Строка");
```

Последнее было бы невозможно для указателя вида

```
char const *str = "Строка текста";
```

т. к. он *указывает на неизменяемую строку*.

Модификатор `volatile` несет противоположный смысл. Он подчеркивает, что значение переменной может быть изменено не только исполняемой программой, но и некоторым внешним воздействием, например, программой обработки

прерываний или обменом с внешним устройством. Фактически, объявление volatile "предупреждает" компилятор, что модифицируемая переменная может измениться в любой момент и не следует делать предположений относительно стабильности значения соответствующего объекта в памяти. Для выражений, содержащих объекты volatile, не будут применяться методы оптимизации кода, такие объекты не будет загружаться в регистры процессора.

Модификатор pascal в применении к идентификатору означает, что он преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. В применении к имени функции модификатор оказывает влияние также на передачу аргументов. В этом случае аргументы пересылаются в стек *в прямом порядке* , т. е., первым отправляется первый аргумент. По умолчанию в Си принято пересылать аргументы в обратном порядке. Существует также настройка компиляции, которая присваивает всем функциям и указателям на функции тип pascal. Такие функции могут вызываться из программы на Паскале. При этом может потребоваться, чтобы некоторые функции и указатели на функции применяли вызывающую последовательность, принятую в Си, а их имена имели принятый для идентификаторов Си вид. В этом случае соответствующие объявления делаются с модификатором cdecl. Все функции в стандартных включаемых файлах объявлены с модификатором cdecl.

Модификатор interrupt предназначен для объявления функций-обработчиков прерываний. Для такой функции генерируется дополнительный код для сохранения и восстановления состояния регистров процессора. Модификатор не может использоваться совместно с near, far или huge. Приведем пример использования volatile и interrupt.

```
volatile int t;
void interrupt time() { t++; }
wait(int i) {
 t=0;
 while ( t<i);
```

```
}
```

Без объявления `t` как `volatile` оптимизирующий компилятор мог бы вынести за пределы цикла сравнение `t` и `i`, поскольку обе они не меняются в теле цикла. Это привело бы к зацикливанию.

Модификаторы `near`, `far`, `huge` оказывают действие на работу с адресами объектов. Формат указателей, принятый в данный момент по умолчанию, определяется используемой *моделью памяти* (см. п. 11). Однако можно объявить указатель с форматом, отличным от действующего по умолчанию. Это делается с помощью явного указания ключевого слова `near`, `far` или `huge`:

```
char far *p = 0xB8000000ul;
 //"длинный" указатель сегмент+смещение
```

Подробнее действие этих модификаторов раскрыто в п. 11.

Допускается более одного модификатора для одного объекта, например

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

9.2. Перечислимый тип данных

Объявление переменной *перечислимого типа* задает имя переменной и определяет список именованных констант, значения которых она может принимать. Каждому элементу списка перечисления ставится в соответствие целое число. Переменная перечислимого типа может принимать только значения из своего списка перечисления.

Практически в любом контексте перечислимый тип интерпретируется как `int`; первый элемент списка перечисления по умолчанию равен 0.

В примере ниже описывается тип `day` для перечисления дней недели. При описании переменной типа сразу же создается переменная `workday`.

```
enum day { SATURDAY, SUNDAY=0, MONDAY,
TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
workday;
workday = WEDNESDAY; //SATURDAY также =0!
```

В следующем примере значения переменной перечислимого типа последовательно распечатываются как целые.

```
typedef enum season {
 spring, summer, autumn, winter };
season now=spring;
for (int i=0; i<4; i++)
 printf ("\n%d", now++); //0,1,2,3
```

В последнем примере значение `item3` установлено равным 0, однако при увеличении переменной `myitem` в цикле она последовательно примет значения от 0 до 3.

```
#include <stdio.h>
typedef enum item {
 item1, item2, item3=0, item4 };
void main () {
 item myitem=item1;
 for (int i=0; i<4; i++)
  printf ("\n%d",myitem++); //0,1,2,3
 myitem=item1; printf ("\n%d",myitem); //0
 myitem=item2; printf ("\n%d",myitem); //1
 myitem=item3; printf ("\n%d",myitem); //0
 myitem=item4; printf ("\n%d",myitem); //1
}
```

9.3. Структуры

Структура — основной составной тип данных. В отличие от массива, она объединяет в одном объекте совокупность значений, которые могут иметь различные типы.

В Си реализован ограниченный набор операций над структурами как единым целым: передача структуры в качестве аргумента функции, возврат структуры из функции, получение ее адреса и создание указателя на структуру. Можно присваивать одну структуру другой, если они имеют одинаковый *тег* (структурный тип).

Общий синтаксис описания структуры следующий:

```
struct тег {
```

```
 список объявлений элементов;
} список описателей;
```

Например, классический способ определения структурного типа может выглядеть так:

```
struct book {
 char title [80];
 char author [40];
 float cost;
};
struct book mybook;
```

Рекомендуется в описании структуры использовать ключевое слово `typedef`:

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

В этом случае можно не писать ключевое слово `struct` в описании структур вновь созданного типа или при передаче их функциям. Обращение к полю структуры выполняется с помощью первичной операции "`.`" (точка).

```
void main () {
 point a,b;
 a.x=a.y=0;
 b=a;
 printf ("\na=(%lf,%lf)",a.x,a.y);
 printf ("\nb=(%lf,%lf)",b.x,b.y);
}
```

Альтернативный вид программы мог бы быть следующим:

```
struct point {
 //'typedef' здесь был бы ошибкой!
 double x,y;
} a,b;
void main () {
 a.x=a.y=0;
 b=a;
 //...
}
```

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

Идентификаторы элементов структуры должны различаться между собой. Идентификаторы элементов разных структур могут совпадать.

Элементы структуры запоминаются в памяти последовательно в том порядке, в котором они объявляются: первому элементу соответствует меньший адрес памяти, а последнему — больший.

Каждый элемент в памяти может быть выровнен на границу слова, соответствующую его типу. Выравниванием управляет опция Word Alignment в компиляторах. Для процессоров на базе Intel 8086/8088 выравнивание означает, что любой тип выравнивается на четную границу адресации при размере машинного слова 2 байта.

В качестве примера приведем код, работающий с текстовым экраном через структуры. При записи в видеопамять используется тот факт, что текстовый режим консоли требует на одну экранную позицию 2 последовательно расположенных байта памяти. Первый из них описывает выводимый на экран символ, а второй — его атрибут, задающий цвет и фон.

```
#include <stdio.h>
typedef struct texel_struct {
 unsigned char ch;
 unsigned char attr;
} texel;
typedef texel screen_array [25][80];
screen_array far *screen_ptr =
 (screen_array far *)0xB8000000L;
#define screen (*screen_ptr)
void fillscr
 (int l,int t,int r,int b,int c,int a) {
 int i,j;
 for (i=t; i<=b; i++)
```

```
 for (j=l; j<=r; j++) {
  screen[i][j].ch=c; screen[i][j].attr=a;
 }
}
void main () {
 fillscr(0,0,79,24,'*',128);
 getchar();
}
```

Для доступа к полям структуры через указатель вместо конструкции

```
(*ptr).field
```

необходимой из-за того, что приоритет операции "`.`" выше, чем у унарной "`*`", используется сокращенная запись

```
ptr->field
```

Оба способа обращения компилируются одинаково, но второй более компактен и удобен.

Работу с указателем на структуру проиллюстрируем следующим примером, в котором формируется и выводится массив из 3 переменных структурного типа.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <stdlib.h>
struct test {
 char *name; //Фамилия или имя
 int ball; //Средний балл
};
struct test *tests;
 //Указатель на массив структур
void check (struct test *);
void main () {
 const int n=3;
 tests = (struct test *) malloc
  (n * sizeof (struct test));
  //Выделение памяти под массив структур
 char buf[80];
 for (int i=0; i<n; i++) {
```

```
    puts ("Имя? "); fgets (buf,78,stdin);
    tests[i].name =
     (char *) malloc (strlen(buf));
     //Выделение памяти под одну фамилию
    buf[strlen(buf)-1]='\0';
     //Удаляем '\n', который fgets
     //оставит в конце строк
    strcpy (tests[i].name, buf);
    puts ("Балл? "); fgets (buf,5,stdin);
    tests[i].ball = atoi (buf);
    check (&tests[i]);
   }
   for (i=0; i<n; i++)
    printf ("\n%s,%d",
     tests[i].name,tests[i].ball);
}
void check (struct test *p) {
 if (p->ball < 0 ) p->ball = 0;
 if (strlen (p->name)<1) {
  p->name = (char *) malloc (6);
  strcpy (p->name,"NoName");
 }
}
```

Использование указателей на структурные типы в сочетании с указателями на функции обеспечивает мощь и гибкость языка Си в инкапсуляции данных даже без применения появившихся в Си++ *классов* (см. п. 13). Следующий пример представляет собой "заготовку" для несложной системы динамических меню.

```
#include <conio.h>
#include <stdlib.h>
typedef unsigned char byte;
typedef void (*FUN)(void);
 //Указатель на функцию обработки
 //пункта меню
struct MENU {
 int x,y; // Позиция на экране пункта меню
```

```
 byte *str; // Строка текста меню
 FUN sf; // Указатель на функцию
         // обработки пункта
};
void Exit () { exit (0); }
void Start () { /* код функции
 обработки пункта меню */ }
void DrawMenu (MENU *m) {
 gotoxy(m->x,m->y);
 cprintf ("%s",m->str);
}
#define ITEMS 2
void main () {
 MENU Menu[ITEMS]={
  { 1, 1, "Начать", Start },
  {10, 1, "Выход", Exit }
 };
 clrscr ();
 for (int i=0; i<ITEMS; i++)
  DrawMenu (&Menu[i]);
}
```

9.4. Битовые поля структур используются обычно в двух целях:

- для экономии памяти, поскольку позволяют плотно упаковать значения не по границам байтов;
- для организации удобного доступа к регистрам внешних устройств, в которых различные биты могут иметь самостоятельное функциональное назначение, например, при доступе к элементам файловых таблиц FAT.

Объявление битового поля имеет следующий синтаксис:

```
тип идентификатор: константное_выражение;
```

Целочисленное константное выражение, записанное после двоеточия, определяет число бит, выделяемых под переменную.

Идентификатор необязателен. Неименованное битовое поле означает пропуск указанного числа битов перед размещением

следующего элемента структуры. Неименованное битовое поле, для которого указан размер `0`, имеет специальное назначение: оно гарантирует, что память для следующей переменной в этой структуре будет начинаться на границе машинного слова (`int`). Это относится и к следующему битовому полю.

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

В примере ниже показана структура, описывающая текстовый экран консоли с возможностью доступа к отдельным битам байта-атрибута через поля.

```
struct {
 unsigned background: 8;
 unsigned color: 4;
 unsigned underline: 1;
 unsigned blink: 1;
} screen [25][80];
```

9.5. Объединение позволяет в разные моменты времени хранить в одном объекте значения различного типа. При объявлении объединения для него описывается набор типов значений, которые могут с ним ассоциироваться. В каждый момент времени объединение интерпретируется как значение только одного типа из набора. Контроль над тем, какого типа значение хранится в данный момент в объединении, возлагается на программиста. Синтаксис объявления объединения имеет следующий вид:

```
union тег {
 список_объявлений_элементов;
} список_описателей;
```

Память, которая выделяется для объединения, определяется размером наиболее длинного из его элементов. Все элементы

объединения размещаются с одного и того же адреса памяти. Значение текущего элемента объединения теряется, когда другому элементу присваивается значение. В качестве примера приведем объединение, которое можно интерпретировать как знаковое или беззнаковое целое число.

```
union sign {
 int svar;
 unsigned uvar;
} number;
sign number2;
```

Во втором примере показано объединение, которое можно использовать для получения либо полного двухбайтового кода нажатой клавиши, либо отдельно скан- и ASCII-кодов.

```
union key {
 char k[2];
 unsigned kod;
};
```

Элементами объединений могут быть и указатели. Например, в приведенном далее коде с помощью 4-байтового указателя "сегмент-смещение" отслеживается нажатие клавиш Ctrl, Shift или Alt через BIOS.

```
#include <stdio.h>
#include <conio.h>
struct FAR_PTR {
 unsigned off, seg;
};
union MK_FAR {
 unsigned char far *ptr;
 struct FAR_PTR mk;
} work;
void main () {
 work.mk.off = 0x17;
 work.mk.seg = 0x40;
  //сегмент и смещение в BIOS,
  //где хранится флаг нажатия клавиш
 while (!kbhit ()) {
  int shift = *(work.ptr);
```

```
    cprintf ("\r%02d",shift);
   }
  }
```

10. Директивы препроцессора

Компиляция программы на Си — многопроходная. На ее нулевой фазе для обработки исходного файла используется *препроцессор*. Компилятор вызывает препроцессор автоматически, однако последний может быть вызван и независимо. Директивы препроцессора — это инструкции, записанные непосредственно в исходном тексте программы.

Директивы нужны для того, чтобы облегчить написание и модификацию программ, а также сделать их более независимыми от аппаратных платформ и операционных систем. Директивы препроцессора позволяют заменять лексемы в тексте программы, вставить в файл содержимое других файлов, запрещать компиляцию части файла или делать ее зависимой от некоторых условий и т. д.

Можно выделить следующие основные виды директив:

- определение макрокоманд;
- вставка файлов;
- условная компиляция программы.

Директивы препроцессора Си перечислены в табл. 10.1.

Таблица 10.1. Директивы препроцессора Си

`#define`	`#else`	`#if`	`#ifndef`	`#line`
`#elif`	`#endif`	`#ifdef`	`#include`	`#undef`

При указании любой директивы первым значащим символом в строке должен быть символ "#".

Директивы могут быть записаны в любом месте исходного файла. Их действие распространяется от точки программы, в которой они записаны до конца исходного файла. Часть директив могут содержать аргументы.

10.1. Определение макрокоманд

Используя конструкцию

```
#define идентификатор текст
```

программист может определить символическое имя или символическую константу, все появления которых, не заключенные в кавычки, будут заменены конкретной строкой символов. Эта операция называется *макроподстановкой*. После того, как макроподстановка выполнена, полученная строка вновь просматривается для поиска новых макроопределений. При этом ранее произведенные макроподстановки не принимается во внимание. Таким образом, директива вида

```
#define x x
```

не приведет к зацикливанию препроцессора. Это же позволяет делать макроопределения "многоступенчатыми" и зависящими от ранее сделанных подстановок. Обычное назначение директивы `#define` — введение удобных символических имен для различного рода констант и ключевых слов:

```
#define  null 0
#define begin {
```

Существенно то, что макроопределениям можно передавать параметры:

```
#define max((a),(b)) ((a)>(b)?(a):(b))
```

Формальные параметры макроопределения должны отличаться друг от друга. Их область действия ограничена определением, в котором они заданы. Список должен быть заключен в круглые скобки. При обращении к директиве с любыми фактическими аргументами они будут подставлены на место формальных параметров `a` и `b`. При этом не происходит вызова какой-либо функции, но выполняется код условной функции, подставленный препроцессором в каждое место вызова макроопределения. Эта подстановка носит чисто текстовый характер. Никаких вычислений или преобразований типа при этом не производится.

В примере с макроопределением max его формальные аргументы для надежности заключены в круглые скобки. Однако это не гарантирует отсутствия побочных эффектов. Так,

при макровызове `max(i, a[i++])` он преобразуется к виду `((i)>(a[i++])?(i):(a[i++]))`, что, очевидно, может привести к лишнему вычислению инкремента.

Директива `#define` позволяет "склеивать" лексемы как строки. Для этого достаточно разделить их знаками `##`. Препроцессор объединит такие лексемы в одну, например, определение

```
#define name(i, j) i##j
```

при вызове `name(a,1)` образует идентификатор `a1`.

Одиночный символ `#`, помещаемый перед аргументом макроопределения, указывает на то, что аргумент должен быть преобразован в символьную строку, то есть, конструкция вида `#формальный_параметр` будет заменена на конструкцию `"фактический_параметр"`. Например, сделав "отладочное" макроопределение `debug`

```
#define debug(a) printf (#a "=%d\n",a)
```

мы можем печатать значения переменных в формате `имя=значение`. Фрагмент программы

```
int i1=10; debug (i1);
```

после обработки препроцессором превратится в

```
int i1=10; printf("i1" " = %d\n", i1);
```

Наконец, возможно определение вида

```
#define some
```

При этом все экземпляры идентификатора `some` будут удалены из текста программы. Сам идентификатор `some` считается определенным и дает значение `1` при проверке директивой `#if`.

Директива `#undef идентификатор` отменяет текущее определение идентификатора. Только когда определение отменено, именованной константе может быть сопоставлено другое значение. Однако многократное повторение определения с одним и тем же значением не считается ошибкой.

10.2. Включение файлов

Общий вид директивы записывается в одном из двух форматов:

```
#include "имя_пути"
#include <имя_пути>
```

Директива включает содержимое исходного файла, для которого задано `имя_пути`, в текущий компилируемый файл. Например, общие для нескольких файлов проекта макроопределения и описания именованных констант могут быть собраны в одном файле и включены директивой `#include` во все исходные файлы. Включаемые файлы используются также для хранения объявлений внешних переменных и абстрактных типов данных. Препроцессор обрабатывает включенный файл так, как если бы он целиком входил в состав исходного файла в точке вставки директивы.

Директива `#include` может быть вложенной, то есть, встретиться в файле, включенном другой директивой `#include`. Препроцессор использует понятие стандартных каталогов для поиска включаемых файлов. Стандартные каталоги в DOS и Windows задаются командой path операционной системы.

Угловые скобки сообщают препроцессору, что файл ищется в каталоге, указанном в командной строке компиляции, а затем в стандартных каталогах (для их настройки во многих компиляторах служит опция Include directories):

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

Если полное или относительное имя включаемого файла задано однозначно и заключено в двойные кавычки, то препроцессор ищет файл только в каталоге, указанном в имени пути, а стандартные каталоги игнорирует:

```
#include "my.h"
```

10.3. Директива условной компиляции

Эта директива позволяет компилятору исключить из обработки какие-либо части исходного файла посредством проверки условий (константных выражений). Общий синтаксис директивы следующий:

```
#if ОКВ текст
<#elif ОКВ текст>
<#elif ОКВ текст>
<…>
<#else текст>
#endif
```

Здесь `ОКВ` — обозначение *ограниченного константного выражения*, специфика его будет раскрыта далее. Треугольные скобки, как обычно, обозначают необязательные элементы спецификации.

Директива управляет компиляцией частей исходного файла. Разрешается вложение условных директив. Каждой записи `#if` в том же исходном файле должна соответствовать завершающая запись `#endif`. Между ними допускается любое число директив `#elif` и не более одной директивы `#else`. Если ветвь `#else` присутствует, то между ней и `#endif` на данном уровне вложенности не должно быть других записей `#elif`.

Препроцессор выбирает один и только один участок текста для обработки на основе вычисления ограниченного константного выражения, следующего за каждой директивой условия. Выбирается весь текст, следующий за ограниченным константным выражением с ненулевым значением, вплоть до ближайшей записи `#elif`, `#else`, или `#endif` на данном уровне вложенности.

Текст может занимать более одной строки. Он может представлять собой фрагмент программного кода, но может использоваться и для обработки произвольного текста. Если текст содержит другие директивы препроцессора, они выполняются. Обработанный препроцессором текст передается на компиляцию. Все участки текста, не выбранные препроцессором, игнорируются и не порождают компилируемого кода.

Если все выражения, следующие за `#if`, `#elif` на данном уровне вложенности ложны (равны нулю), выбирается текст, следующий за `#else`. Если при этом ветвь `#else` отсутствует, никакой текст не выбирается.

Ограниченное константное выражение не может содержать операций приведения типа, операций `sizeof` (возможна в некоторых компиляторах), констант перечисления и вещественных констант, но может содержать специальную препроцессорную операцию `defined (идентификатор)`. Операция `defined` дает ненулевое значение, если заданный `идентификатор` в данный момент определен; в противном случае выражение равно нулю (ложно). Операция может использоваться в сложном выражении в директиве неоднократно:

```
#if defined(name1) || defined(name2)
```

Приведем пример:

```
#if defined (COLOR)
color();
#elif defined (MONO)
mono();
#else
error();
#endif
```

Здесь условная директива управляет компиляцией одного из трех вызовов функции. Вызов функции `color()` компилируется, если определена именованная константа `COLOR`. Если определена константа `MONO`, компилируется вызов функции `mono()`, если ни одна из двух констант не определена, компилируется вызов функции `error()`.

10.4. Директивы условного определения

Использование директив `#ifdef` и `#ifndef` эквивалентно применению директивы `#if` совместно с операцией `defined(идентификатор)`. Эти директивы поддерживаются для совместимости с предыдущими версиями компиляторов Си. Синтаксис директив следующий:

```
#ifdef идентификатор
#ifndef идентификатор
```

При обработке #ifdef препроцессор проверяет, определен ли в данный момент идентификатор директивой #define. Если это так, условие считается истинным, иначе ложным.

Директива #ifndef противоположна по действию: если идентификатор не был определен директивой #define или его определение отменено директивой #undef, то условие считается истинным. В противном случае условие ложно.

Аналогично #if, за #ifdef и #ifndef может следовать набор директив #elif и/или директива #else. Набор должен быть завершен директивой #endif.

10.5. Управление нумерацией строк

Директива имеет следующий общий вид:

```
#line константа <"имя_файла">
```

Она сообщает компилятору об изменении имени исходного файла и порядка нумерации строк. Изменение отражается только на сообщениях компилятора, исходный файл будет теперь именоваться как "имя_файла", а текущая компилируемая строка получит номер "константа". После обработки очередной строки счетчик номеров строк увеличивается на единицу. В случае изменения номера строки и имени файла директивой #line компилятор продолжает работу с новыми значениями.

Директива используется, в основном, автоматическими генераторами программ. Константа может быть произвольным целым значением. Имя_файла может быть комбинацией символов, заключенной в двойные кавычки. Если имя файла опущено, оно остается прежним. Пример:

```
#line 1000 "file.cpp"
```

Здесь устанавливается имя исходного файла file.cpp и текущий номер строки 1000.

Текущий номер строки и имя исходного файла доступны в программе через псевдопеременные с именами __LINE__ и __FILE__. (см. п. 10.8). Они могут быть использованы для

выдачи сообщений о местоположении ошибки во время исполнения программы.

10.6. Директива обработки ошибок

Директива имеет формат

```
#error текст
```

Чаще всего ее используют для обнаружения некоторой недопустимой ситуации. По директиве `#error` препроцессор прерывает компиляцию и выдает сообщение вида `Fatal: имя_файла номер_строки Error directive: текст`. Здесь `имя_файла` — имя исходного файла, `номер_строки` — текущий номер строки, `текст` — диагностическое сообщение. Пример:

```
#if (Boolean!= 0 && Boolean!=1)
#error Boolean имеет значение не 0 и не 1!
#endif
```

10.7. Указания компилятору Си

Указания компилятору или *прагмы* предназначены для исполнения компилятором в процессе его работы. Они имеют общий синтаксис вида

```
#pragma текст
```

где `текст` задает определенную инструкцию, возможно, имеющую аргументы. Прагмы различны для различных компиляторов, как правило, познакомиться с ними можно при изучении документации к системе программирования.

10.8. Псевдопеременные

Псевдопеременными называют системные именованные константы, которые можно использовать в любом исходном файле на Си. Имена псевдопеременных начинаются и заканчиваются двумя символами подчеркивания (__). Как правило, определены следующие 4 псевдопеременные:

`__LINE__` — десятичная константа, задает номер текущей обрабатываемой строки файла. Первая строка имеет номер 1.

__FILE__ — содержит имя компилируемого файла — символьную строку. Значение псевдопеременной изменяется каждый раз при обработке директивы #include или #line, а также по завершении включаемого файла. Значением __FILE__ является строка, представляющая имя исходного файла, заключенное в двойные кавычки. Поэтому для вывода имени файла не требуется заключать идентификатор __FILE__ в двойные кавычки.

__DATE__ — дата начала компиляции текущего файла, записанная как символьная строка. Каждое вхождение __DATE__ в заданный файл дает одно и то же значение, независимо от того, как долго уже продолжается компиляция. Дата хранится в формате mmm dd yyyy, где mmm — месяц (Jan, Feb и т. д.), dd — число месяца (от " 1" до "31"), yyyy — четырехзначный год (например, 2008).

__TIME__ — время начала компиляции текущего исходного файла, записанное как символьная строка в формате hh:mm:ss, где hh — час от 00 до 23, mm — минуты от 00 до 59, ss — секунды от 00 до 59.. Каждое вхождение __TIME__ в файл дает одно и то же значение, независимо от того, как долго уже продолжается обработка.

11. Модели памяти

Применение *моделей памяти* позволяет контролировать ее сегментное распределение и делать его более эффективным или адекватным решаемой задаче. По умолчанию при компиляции и редактировании связей генерируется код для работы в малой (small) модели. Если программа удовлетворяет хотя бы одному из двух следующих условий, следует использовать другую модель памяти:

- размер кода программы превышает 64 Кб;
- размер статических данных программы превышает 64 Кб.

Имеется два варианта выбора модели памяти для программы:

- назначить нужную модель в опциях компилятора;
- использовать в объявлении объектов программы модификаторы `near`, `far` и `huge`.

Можно комбинировать эти способы.

Архитектура процессоров, основанных на базе 8086/8088, предусматривает разбиение оперативной памяти на физические *сегменты*, способные содержать информацию объемом до 64 Кб. Минимальное количество сегментов, выделяемое программе, равно двум: сегмент кода и сегмент статических данных. К статическим данным при этом относятся все объекты, объявленные с классом памяти `extern` или `static`. Формальные параметры функций и локальные переменные не являются статическими. Они хранятся не в сегменте данных, а в *стеке* (однако при этом стек может быть совмещен со стандартным сегментом данных физически).

Программа на Си может работать с динамической памятью с помощью библиотечных функций семейства `malloc`. При этом память может выделяться как в отдельном сегменте (*дальняя* динамическая память), так и в стандартном сегменте данных между концом занятой данными области и стеком (*ближняя* динамическая память).

Адрес оперативной памяти состоит из двух частей:

- *базовый адрес сегмента* — 16-битовое число;
- *смещение* относительно начала сегмента — также 16-битовое число.

Для доступа к коду или данным, находящимся в единственном стандартном сегменте, достаточно использовать только смещение. В этом случае применяются указатели, объявленные с модификатором `near` (ближний). Поскольку для доступа к объекту используется только одно двухбайтовое число, применение ближних указателей дает компактный по занимаемой памяти код с хорошим быстродействием.

Если код или данные находятся в другом сегменте по отношению к адресующему, для доступа к ячейке памяти

должны и адрес сегмента, и смещение. В этом случае указатели объявляются с модификатором `far` (дальний). Доступ к объектам по дальним указателям позволяет программе адресовать всю оперативную память, а не только в пределах сегмента размером 64 Кб.

Наконец, указатели с модификатором `huge` (максимальный) также включают адрес сегмента, и смещение, но имеют иную адресную арифметику. Поскольку объекты, адресуемые `far` указателями, не выходят за границу адресуемого сегмента, действия по вычислению адресов выполняются только над смещением дальних указателей. Это ускоряет доступ, но ограничивает размер одного программного объекта объемом 64 Кб. Для указателей `huge` арифметические действия выполняются над всеми 32 битами адреса.

Тип адреса `huge` определен только для данных, таких как массивы и указатели на них. Таким образом, никакой из исходных файлов, составляющих программу, не должен генерировать более 64 Кб кода.

Кратко опишем шесть имеющихся в Си моделей памяти.

Минимальная модель (tiny). Код вместе с данными не превышает по объему 64 Кб. Применялась для исполняемых файлов, преобразуемых к формату *.com.

Малая модель (small). Программа занимает 2 стандартных сегмента: сегмент кода и сегмент данных, в котором размещается также стек. Как код, так и данные программы не могут превышать 64 Кб. Модель подходит для большинства несложных программ и назначается компилятором по умолчанию. Для доступа к объектам кода или данных по умолчанию используются указатели типа `near`.

Средняя модель (medium). Для данных и стека выделяется один сегмент, для кода — произвольное число сегментов. Каждому исходному модулю программы выделяется собственный сегмент кода. Модель применяется для программ с большим количеством кода (более 64 Кб) и небольшими объемами данных (менее 64 Кб). Для доступа к функциям по умолчанию используются указатели `far`, а для доступа к

данным — указатели near. Модель предлагает компромисс между скоростью выполнения и компактностью кода, поскольку многие программы чаще обращаются к данным, чем к функциям.

Компактная модель (compact). В этой модели выделяется один сегмент для кода, и произвольное число сегментов для данных. Модель применяется для небольших программ, работающих со значительными объемами данных. Доступ к функциям производится по указателям near, а к данным — по указателям far.

Большая модель (large) использует по несколько сегментов и для кода, и для данных. Модель подходит для больших программ со значительным объемом данных. В этой модели доступ к элементам кода и данных производится по указателям типа far. Как и во всех предшествующих моделях умолчания можно обойти, явно используя модификаторы near, far и huge там, где они требуются при объявлении данных и функций.

Максимальная модель (huge) аналогична большой за исключением того, что в ней снимается ограничение на размер массивов в 64 Кб. Однако для больших массивов не допускается пересечения элементами границ сегмента, из чего следует, что элемент массива не может превышать по размеру 64 Кб. Кроме того, для обеспечения эффективной адресации размер в байтах элемента большого массива должен быть степенью двойки.

Максимальная модель требует учитывать некоторые особенности языка при обращении к операции sizeof и вычитании указателей. По умолчанию значение sizeof имеет тип unsigned int, однако число байтов в huge массиве должно быть представлено типом unsigned long. Для получения правильного значения следует делать приведение типа:

```
(unsigned long)sizeof(huge_array)
```

Аналогично, результат вычитания указателей определен в Си как значение типа int. При вычитании указателей типа

`huge` результат может иметь тип `long`. В этом случае также требуется сделать приведение типа:

```
(long)(huge_item1—huge_item2)
```

12. Динамические структуры данных

Можно выделить следующие основные классы динамических объектов:

- *упорядоченные совокупности* характеризуются тем, что для них значим физический порядок встраивания элементов. Пример — массивы;
- *неупорядоченные совокупности* или *коллекции* — физический порядок объектов в представлении данных незначим, встраивание объектов в коллекцию произвольно. Примеры — хэши, одно- и двусвязные списки, деревья;
- *последовательности* — физический порядок объектов в представлении данных незначим, но значим порядок вставки/удаления, которые выполняются только в определенных точках последовательности. Примеры — стеки, очереди.

Существуют 2 основных возможности для организации динамических структур:

- библиотеки функций classlib;
- организация динамических наборов структур, связанных указателями на структурные типы.

Реализация динамических структур через предопределенные классы требует подключения библиотеки TCLASSS.LIB (если выбрана модель Small, также есть TCLASSL.LIB для Large) из папки BORLANDC\CLASSLIB\LIB к файлу проекта, а также включения в строку опций Include Directories маршрута \BORLANDC\CLASSLIB\INCLUDE. Если предположить, что среда установлена в папку d:\BORLANDC, настройки путей могут иметь следующий вид:

```
Include Directories:
d:\BORLANDC\INCLUDE;
d:\BORLANDC\CLASSLIB\INCLUDE;
d:\BORLANDC\TVISION\DEMOS;.
```

```
Library directories:
d:\BORLANDC\LIB;d:\BORLANDC\TVISION\LIB;
d:\BORLANDC\CLASSLIB\LIB;.
Output Directory: .
Source Directories: d:\BORLANDC\BIN;.
```

Подробное рассмотрение иерархии классов библиотек classlib можно найти в специальной литературе, в данном пособии мы подробнее рассмотрим независимое программирование динамических списков структур.

В простейшем случае элемент списка представляет собой структурную переменную, содержащую один или несколько указателей на следующие или предшествующие элементы и любое число других полей, называемых *информационными*. Если список располагается в оперативной памяти, то информацией для поиска следующего элемента служит адрес (указатель) в памяти. Если список хранится в файле, информация о следующем элементе может включать смещение от начала файла, относительное положение указателя записи/считывания файла, ключ записи и любую другую информацию, позволяющую однозначно отыскать следующий элемент.

Описанный далее односвязный (однонаправленный) список имеет одно информационное поле и один указатель на следующий элемент:

```
typedef struct list {
 list * next;
 int info;
};
```

Для работы с элементами такого списка достаточно определить статический указатель на начало списка

```
list * head;
```

и хотя бы один "буферный" элемент, который будет служить для ввода и временного хранения данных:

```
list work;
```

Как минимум, функции работы с элементами списка, принимающие в качестве параметра указатель на некоторый его

элемент, должны реализовывать операции вывода по порядку всех или избранного элемента, добавление и удаление элементов с выделением и высвобождением памяти, возможно, также поиск элементов, определение того, содержится ли уже в списке нужный элемент и т.п.

Например, функция вывода всех элементов списка, использующая глобальный указатель root на его вершину, могла бы иметь следующий вид:

```
struct list *root;
 //Указатель на вершину списка
void List (void) {
 struct list *iter = root;
 int i = 0;
 while(iter != NULL) {
  //Пока указатель не пуст
  printf ("\n элемент %2d: (%d)",
   i++,iter->info);
  //Напечатать очередной элемент списка
  iter = iter->next;
  //и перейти к след. элементу
 }
}
```

Разумеется, информационные поля списочной структуры также могут выделяться динамически, как показано в примере ниже. Здесь для простоты элементы добавляются всегда в конец списка и выводятся в порядке ввода. Ввод нуля с клавиатуры служит признаком конца ввода.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
typedef struct list {
 char *s;
 list *next;
};
void main () {
 list head, *ptr;
 int i=0;
```

```
 char s[80];
 ptr=&head;
 ptr->next=NULL;
 do {
  printf (" %d) ",i+1);
  fgets (s,80,stdin);
  if (!strcmp(s,"0\n")) {
   break;
  }
  else {
   ptr->s=(char *)malloc(strlen(s));
   strcpy (ptr->s,s);
   ptr->next=(list *)malloc(sizeof(list));
   ptr=ptr->next;
   ptr->next=NULL;
  }
  i++;
 } while (1);
 ptr=&head;
 i=1;
 while (ptr->next!=NULL) {
  printf ("%d) %s",i++,ptr->s);
  ptr=ptr->next;
 }
}
```

В реальном списке может потребоваться множество дополнительных функций, таких как встраивание элементов в определенном порядке, перестановка, удаление, модификация и т. д. Приведем пример функции `add()`, добавляющий элемент, определяемый указателем `new_ptr`, в список с вершиной `head`. Встраивание элемента производится по возрастанию значения поля `info`, предполагается, что перед этим элемент уже проверен на уникальность функцией `hasMember()`. Использование указателя `list **head` в функции `add()` позволяет избежать прямого сравнения указателей в коде. При описании указателя на начало списка в

виде `list *head;` и указателя на новый элемент `list *new_ptr;`, функция могла бы быть вызвана оператором вида `add(&head, new_ptr);`.

```
int add(list ** head, list * new_ptr) {
 list * first, * second;
 if ((*head)==NULL) { //список пуст
  (* head) = new_ptr;
  new_ptr -> next = NULL;
  return 0;
 }
 if ((*head)->next == NULL) {
  //в списке один элемент
  if ( (*head)->info >  new_ptr->info) {
   // новый элемент - первый в списке
   second = (*head);
   // сохраним указатель на 2-й элемент
   (*head) = new_ptr;
   new_ptr -> next = second;
   second -> next = NULL;
  }
  else { // новый элемент в конец списка
   (*head) -> next = new_ptr;
   new_ptr -> next = NULL;
  }
  return 1;
 }
 else { //в списке более 1 элемента
  if ( (*head)->info > new_ptr->info) {
   // новый элемент - первый в списке
   second = (*head);
   (*head) = new_ptr;
   new_ptr -> next = second;
   return 4;
  }
  first  = (* head);
  second = first -> next;
  while (first->next != NULL) {
```

```
    //цикл поиска места в списке
    if (first->info <= new_ptr->info &&
        second->info >= new_ptr->info) {
     // вставляем элемент между
     // first и second
     first -> next   = new_ptr;
     new_ptr -> next = second;
     return 2;
    }
    first = second;
    second = first -> next;
   }
   // если добрались сюда,
   // ставим элемент в конец списка
   first -> next    = new_ptr;
   new_ptr  -> next = NULL;
   return 3;
  }
}
int hasMember (list * head, list * work) {
 while(head != NULL) {
  // цикл сканирования списка
  if (head->info == work -> info) return 1;
  head = head -> next;
 }
 return(0);
}
```

Аналогичным образом с использованием структур и указателей могут быть реализованы другие динамические стуктуры данных. Приведем пример на работу с деревом, каждый узел которого может иметь произвольное количество узлов-потомков.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
typedef struct tree {
 int info;
```

```
 tree **childs;
 unsigned count;
};
int const size1=sizeof(tree);
void error (int c) {
 printf ("\nError: ");
 switch (c) {
  case 1:  printf(" no memory"); break;
  default: printf(" unknown");   break;
 }
 exit (c);
}
tree *init (tree *p,int info) {
 p->info=info;
 p->count=0;
 p->childs=NULL;
 return p;
}
void print1 (tree *p) {
 printf ("\nnode %6d: %2d child(s)",
  p->info,p->count);
}
void printnode (tree *p) {
 print1 (p);
 for (int i=0; i<p->count; i++)
  print1 (p->childs[i]);
}
tree *searchnode (tree *t, int info) {
 if (t->info == info) return t;
 else if (t->count>0) {
  for (int i=0; i<t->count; i++) {
   tree *p=searchnode (t->childs[i],info);
   if (p!=NULL) return p;
  }
  return NULL;
 }
 else return NULL;
```

```
}
tree *addnode (tree *ptr, int parentinfo,
 int info) {
 tree *p=searchnode (ptr,parentinfo);
 if (p!=NULL) {
  if (p->childs==NULL) {
   p->childs = (tree **)malloc
    (sizeof(tree *));
   if (p->childs==NULL) error (1);
   p->childs[0]=(tree *)malloc(size1);
   if (p->childs[0]==NULL) error (1);
   (p->count)=1;
  }
  else {
   p->childs = (tree **)
   realloc(p->childs,sizeof(tree *)
    *(p->count+1));
   if (p->childs==NULL) error (1);
   p->childs[p->count]=(tree *)
    malloc(size1);
   if (p->childs[p->count]==NULL)
    error (1);
   (p->count)++;
  }
  return init (p->childs[p->count-1],info);
 }
 return NULL;
}
void main () {
 tree head,temp,*ptr;
 ptr=&head;
 init(ptr,1);
 addnode (ptr,1,2);
 addnode (ptr,1,3);
 addnode (ptr,2,4);
 tree *s=searchnode (ptr,2);
 printf ("\n With node after search:");
```

```
 if (s!=NULL) printnode (s);
 else printf ("\nNULL");
 printf ("\n With root:");
 printnode (ptr);
}
```

13. Классы

Классы появились в языке Си++, служащем расширением классического Си. Формально описание класса выглядит следующим образом:

```
class ИмяКласса {
 private:
  Список членов класса;
 protected:
  Список членов класса;
 public:
  Список членов класса;
};
```

Список членов класса включает описание типов и имен как данных, так и функций. Переменные, перечисляемые в классе, по умолчанию имеют область видимости в пределах класса (`private`), и доступ к ним имеют только функции-члены данного класса. Обычно функции-члены имеют тип доступа `public`, т.е., видимы вне класса, к ним может осуществляться доступ извне. Атрибут `protected` назначается тем членам класса, которые могут использоваться методами данного и производных от него классов.

Функции-члены связаны с классом специальным оператором `::` и обычно описаны сразу после описания класса, к которому принадлежат. В остальном они выглядят как обычные функции Си.

Рассмотрим работу с классом на подробном примере.

```
#include <stdio.h>
#include <stdlib.h>
class cList { //Класс для описания списка
```

```
 unsigned int item;
 cList *next;
  // Указатель на следующий элемент
public:
 void show(); // Просмотр списка
 cList();//Конструктор по умолчанию -
         //сгенерировать item случайно
 cList (FILE *);
         //Другой конструктор - прочитать
         //элемент из файла
 cList (unsigned int);
         //Третий конструктор -
         //ввести значение с клавиатуры
 ~cList();   // Деструктор
 void Start (); //Прототипы
 void End();    //методов класса
};
cList *first=NULL;  //начало списка
void cList::show() {
 cList *p; int i=1;
 for (p=first; p !=NULL; p=p->next) {
  printf ("\n Элемент %d) %u",i++,p->item);
 }
}
cList::cList() {
 randomize();
 item=random(32000);
 next=this;
}
cList::cList(FILE *f) {
 unsigned int n=0;
 fscanf (f,"%u",&n);
 fclose (f);
 item=n;
 next=this;
}
cList::cList(unsigned int n) {
```

```
 item = n;
 next = this;
}
void cList::Start () {
 this->next = first;
 first = this;
}
void cList::End () {
 cList *sled,*pred;
 for (sled=first,pred=NULL;
  sled !=NULL; pred=sled,sled=sled->next);
 pred->next = this;
 this->next = NULL;
 sled = this;
}
void main () {
 cList *List1 = new cList(3);
 List1->Start();
 cList *List2 = new cList ();
 List2->End();
 List1->show();
 // ...
}
```

Оператор . (точка), как и в Delphi, позволяет связать функцию с конкретным экземпляром класса. При использовании указателей, как и со структурами, конструкция `(*указатель).поле` заменяется на `указатель->поле`.

Для того, чтобы вызываемая функция точно "знала", с каким из объектов класса она работает, Си++ неявно передает в нестатическую функцию еще один скрытый параметр - указатель на текущий объект, называемый `this`. Параметр `this` видим в теле вызываемой функции, устанавливается при вызове в значение адреса начала объекта и может быть использован для доступа к членам объекта.

В Си++ разрешено описание прототипов функций с формальными параметрами, имеющими значение по

умолчанию. Такие параметры должны быть последними в списке при объявлении функции:

```
int f (int i,int k=5) {
 //тело функции
}
```

Вызвать функцию `f()` можно, например, так:

```
f(i,j); //формальный параметр
        //i=фактическому i, k=j
f(1);   //i=1, k=5
```

Класс на Си++ может иметь любое количество *конструкторов*, предназначенных для создания экземпляров класса. Конструктор всегда имеет то же имя, что и класс, в котором он определен, с созданием экземпляра всегда связано явное или неявное выполнение конструктора. Если отсутствует явно описанный конструктор, создается конструктор по умолчанию. Конструктор вызывается компилятором явно при создании объекта и неявно при выполнении оператора `new`, применяемого к объекту, а также при копировании объекта данного класса.

Конструктор копирования для класса `X` имеет 1 аргумент типа `X` и, возможно, параметры по умолчанию, например

```
class X {
 public:
  X() { ... } //конструктор по умолчанию
  X(const &X) { ... }
   //конструктор копирования
  X(const &X,int=4) { ... }
   //конструктор по умолчанию
   //с параметром по умолчанию
}
```

Конструктор копирования вызывается, когда происходит копирование объекта, обычно при объявлении с инициализацией:

```
X one;     //вызван конструктор по умолчанию
X two=one; //вызван конструктор копирования
```

Функция-*деструктор* разрушает объект данного класса и вызывается явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Явное уничтожение объекта выполняет оператор `delete`. Деструктор имеет то же имя, что класс, но предваренное символом `~`. Деструкторы не могут получать аргументы и быть перегружены. Класс может объявить только один общедоступный деструктор. Если класс не содержит объявления деструктора, компилятор автоматически создаст его.

Как и для обычных функций, для членов класса поддерживается "перегрузка", то есть, использование одинаковых имен для функций, отличающихся числом и типом параметров. Функции не могут перегружаться, если они отличаются только типом возвращаемого значения, но не списком параметров, а также, если их типы аргументов неявно совпадают (`int` и `int &`).

Функции, не являющиеся членами класса, но объявленные его "друзьями" с помощью ключевого слова `friend`, имеют полный доступ к данным класса.

Многие операторы и встроенные функциональные вызовы Си++ также могут быть перегружены в пределах контекста класса, где сделано соответствующее определение. При этом приоритет и порядок выполнения операций не меняются. Переопределены могут быть следующие лексемы:

```
+ - * / = < > += -= *= /= << >> <<= >>= == !=
<= >= ++ -- % & ^ ! | ~ &= ^= != && || %= []
() new delete
```

Смысл перегрузки оператора в том, что создается функция, выполняемая каждый раз, когда в контексте класса встречается этот оператор. Синтаксис определения перегрузки оператора таков:

```
ИмяТипа operator СимволОперации
  (СписокПараметров)
```

Производный (англ. derived) класс — это расширение существующего класса, именуемого в этом случае *базовым*.

Производный класс может модифицировать права доступа к данным класса, добавить новые члены или перегрузить существующие функции. Описание производного класса делается так:

```
class ИмяПроизводногоКласса : БазовыйСписок {
 СписокЧленов;
}
```

Здесь `БазовыйСписок` содержит перечень разделенных запятой спецификаторов атрибутов доступа (`public` или `private`) и имен базовых классов.

Производный класс наследует все члены перечисленных базовых классов, но может использовать только члены с атрибутом `public` или `protected`.

Эти и другие особенности классов на Си++ рассмотрим на примере. Код ниже представляет класс `Person` и расширение класса `Student`, а также демонстрирует создание экземпляров этих классов.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
class Person {
 private: //Данные объекта; атрибут,
          //применяемый по умолчанию
  char *Name;
  int Age;
  friend Person *upcase (Person *p);
   //эта функция - не член класса, но его
   //друг - она имеет полный доступ
   //к его членам
 protected: //Данные и методы могут
     //использоваться функциями-членами и
     //друзьями класса, для которого данный
     //класс является базовым
  char *putName (char *name);
 public:
      //Данные и методы, доступные извне
```

```
  static int Count;
    //Статический член класса –
    //один для всех его экземпляров!
  Person () : Name (NULL), Age (0) {
   //Конструктор без аргументов
   //аргументы установлены по умолчанию
   Name = NULL;
    //Тело конструктора встроено
   Count++;
  }
  Person (char *name, int age);
   //Прототип конструктора с аргументами
  ~Person ();
   //У класса есть деструктор,
   //освобождающий память, занятую объектом
  Person operator + (char *);
   //Переопределение оператора +
   //для сложения строк –
   //действует в этом классе
   //на входе - параметр-указатель,
   //на выходе - новый объект Person
  //Методы класса:
  void Print (); //Перегружаемая функция,
  Person Print (Person *p);
   //т.е. у нас нес-ко функций с 1 именем
   //и разными аргументами
  void Input ();
  inline char * getName () {
   return Name;   //Встроенная функция
  }; //ее код каждый раз подставляется
     //в точку вызова
  inline int getAge () { return Age; };
};
//Описание производного класса:
class Student : public Person {
  // : - оператор наследования
  //Класс, порожденный от Person,
```

```
  //наследует его члены и может
  //использовать члены с атрибутом public
 protected:
  int Group;
 public:
  Student (char *name=NULL, int age=0,
   int group=0) :
   Group(group),Person(name,age) {}
   //Конструктор со значениями аргументов
   //по умолчанию и вызовом конструктора
   //родительского класса Person
  void Print ();
   // Метод производного класса
};
 int Person::Count=0;
 //один раз инициализировали статический
 //член вне описания класса!
// Оператор Класс :: показывает, что
//объект относится к классу:
Person :: Person (char *name, int age) {
  //Тело конструктора с аргументами
 this->Age = age;
  //this - указатель на текущий объект
  //из метода класса
  //неявно передается в любую нестатическую
  //функцию класса
 this->Name=putName (name);
 Count++;
}
Person :: ~Person () {
 //Деструктор всегда без аргументов,
 //не м.б. перегружен
 free (Name);
}
//Служебная функция:
char * Person :: putName (char *name) {
 Name = (char *) calloc
```

```
  (strlen(name),sizeof(char));
 if (Name!=NULL) strcpy (Name, name);
 return Name;
}
inline void Person :: Print () {
 //Эта функция - встроенная в класс
 printf ("\n%s (%d)",Name,Age);
}
Person Person :: Print (Person *p) {
 printf ("\nOther print: %s, %d",
  p->Name, p->Age);
 return *p;
}
//Перегрузка оператора
Person Person :: operator + (char *s) {
 Person temp; char name[80];
 strcpy(name,strcat (this->Name,s));
 temp.Name = putName(name);
 temp.Age=this->Age;
 return temp;
}
void Person :: Input () {
 //Создаем объект и возвращаем его
 char s[80];
 printf ("\nName? ");
 scanf ("%s",s);
 strcpy (Name,s);
 printf ("\nAge? ");
 scanf ("%d",&Age);
 putName (s);
}
Person *upcase (Person *p) {
 //функция-друг класса
 strupr (p->Name);
 return p;
}
void Student :: Print () {
```

```
 printf ("\n%s (%d,%d)",
  getName(),getAge(),Group);
}
void main () {
 Person *First = new Person ();
  //new вернет указатель на новый объект
  //происходит вызов конструктора
 First->Print ();
 delete First; //удаляем объект, неявно
               //вызвав деструктор
 Person *Second = new Person ("Ivanov",20);
 Second->Print ();
 Person Third; Third.Input ();
 upcase (&Third); Third.Print (&Third);
 Person Next = Third + " .N";
 Next.Print ();
 printf ("\nAll: %d",Person::Count);
 delete &Next;
 Student *Test = new Student ("Newer",
  Second->getAge(),319);
 Test->Print (); }
```

Рекомендуемая литература

1. Бочков С.О., Субботин Д.М. Язык программирования Си для персонального компьютера. — М.: "Радио и связь", 1990. — 384 с.
2. Жешке Р. Толковый словарь стандарта языка Си. — Спб.: "Питер", 1994. — 224 с.
3. Керниган Б.В,, Ричи Д.М. Язык С. — М.: "Финансы и статистика", 1992. — 232 с.
4. Страуструп Б. Язык программирования С++ — Спб.: "Невский диалект", 2007. — 1099 с.