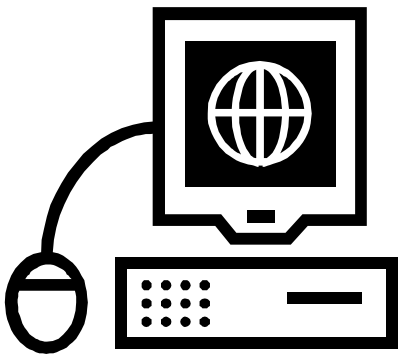
А.В. Романенко, А.И. Попов

ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ И РЕШЕНИЯ ТВОРЧЕСКИХ ЗАДАЧ



• ИЗДАТЕЛЬСТВО ТГТУ •

УДК 681(075)
ББК ←973-018я73
Р69

Р е ц е н з е н т ы:

Доцент МГТУ им. Баумана
В.А. Мартынюк

Директор института информационных технологий и коммуникаций
Астраханского государственного технического университета
О.М. Проталинский

Профессор, доктор физико-математических наук
С.М. Дзюба

Романенко, А.В.

Р69 Основы программирования для автоматизированного проектирования и решения творческих задач : учеб. пособие / А.В. Романенко, А.И. Попов. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2007. – 96 с. – 150 экз. – ISBN 5-8265-0595-8 (978-5-8265-0595-3).

Рассмотрены теоретические вопросы программирования на одном из широко распространенных языков – языке С. Приведены приемы и алгоритмы для обработки данных различных типов. Даны задания для самостоятельной подготовки и рассмотрены примеры решения задач повышенной сложности.

Предназначено для подготовки студентов по дисциплинам, связанным с программированием, может быть использовано профессорско-преподавательским составом и инженерно-педагогическими работниками.

УДК 681(075)
ББК ←973-018я73

ISBN 5-8265-0595-8
(978-5-8265-0595-3)

Министерство образования и науки Российской Федерации

ГОУ ВПО "Тамбовский государственный технический университет"

А.В. РОМАНЕНКО, А.И. ПОПОВ

ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ И РЕШЕНИЯ ТВОРЧЕСКИХ ЗАДАЧ

Допущено Учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению 230100 "Информатика и вычислительная техника", специальности 230104 "Автоматизированные системы проектирвоания"

Тамбов
Издательство ТГТУ
2007

Учебное издание

РОМАНЕНКО Александр Васильевич
ПОПОВ Андрей Иванович

ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ И РЕШЕНИЯ ТВОРЧЕСКИХ ЗАДАЧ

Учебное пособие

Редактор З.Г. Ч е р н о в а
Инженер по компьютерному макетированию М.Н. Р ы ж к о в а

Подписано в печать 14.05.2007
Формат 60 × 84/16. 5,58 усл. печ. л. Тираж 150 экз. Заказ № 343

Издательско-полиграфический центр
Тамбовского государственного технического университета
392000, Тамбов, Советская, 106, к. 14

ВВЕДЕНИЕ

Современные техника и многие технологии немыслимы без использования вычислительной техники. Специализированные ЭВМ и компьютеры общего назначения находят широкое применение практически во всех направлениях человеческой деятельности. Одним из таких направлений является разработка новых технических устройств и систем, в процессе создания которых широко применяется человеко-машинный комплекс, называемый системой автоматизированного проектирования (САПР).

Однако, сами САПР являются сложными системами обработки информации, для создания и эксплуатации которых необходимо присутствие грамотных специалистов в этой области.

В настоящее время для создания программных комплексов различной сложности часто используется язык программирования С. Язык С задумывался как язык программирования "среднего" уровня вследствие наличия некоторых особенностей, целью которых было предоставление программисту удобного инструментального языка, заменяющего язык ассемблера и обеспечивающего легкий доступ к аппаратным средствам ЭВМ. Он был разработан в первой половине 70-х годов XX века сотрудниками фирмы Bell Laboratories Д. Ритчи и Б. Керниганом. Популяризации языка С способствовало написание на этом языке текстов операционной системы Unix. В результате ОС Unix получила максимальную переносимость на разные типы ЭВМ, что способствовало приобретению ею огромной популярности, а язык С стал ее базовым языком программирования. Распространению языка долгое время препятствовало отсутствие полного и однозначного его описания. Вследствие этого в 1983 году в Американском Национальном Институте Стандартов (ANSI) был образован комитет по стандартизации языка С и в 1989 году стандарт на язык программирования С был утвержден и началось беспрепятственное распространение языка С в программистской среде.

Данное учебное пособие предназначено для выработки у студентов навыков программирования задач различной сложности с помощью одного из широко распространенных языков программирования – языка С.

Компоненты языка С в пособии выделены **жирным шрифтом**, а фрагменты, на которые следует обратить внимание – *курсивом*.

Во второй части пособия приведены практические задания, которые будут полезны для самостоятельного закрепления изучаемого материала.

Ч а с т ь I
ОСНОВЫ ЯЗЫКА С

Любой язык программирования образуют три составляющие части: алфавит, синтаксис и семантика.

Алфавит – фиксированный для данного языка набор основных символов, из которых должен состоять любой текст на этом языке. Никакие другие символы для записи текстов не допускаются.

Синтаксис – система правил, определяющих допустимые конструкции из букв алфавита. С помощью этих конструкций представляются отдельные компоненты алгоритма и алгоритм в целом, записанные на данном языке программирования.

Семантика – это система правил истолкования отдельных языковых конструкций, позволяющих однозначно воспроизвести процесс обработки данных по заданной программе.

Выделяют основные понятия языков программирования.

- *Операторы*. Понятие оператора является ключевым для любого языка программирования. Оператор представляет собой законченную фразу языка, содержащую полностью оформленный этап обработки информации. Операторы разделяют на две группы: основные (не содержащие других операторов) и производные (составные). В алгоритмических языках операторы обычно отделяют друг от друга точкой с запятой.
- *Идентификаторы.* В процессе обработки информации, алгоритму необходимо различать программные объекты. Для обозначения имен переменных, их свойств и атрибутов используются идентификаторы. Идентификатор представляет собой сочетание букв и цифр, которое начинается с буквы и не превышает заданной для языка программирования длины. В любом языке программирования существуют стандартные идентификаторы, произвольное использование которых запрещено.
- *Переменные.* Переменные представляют собой объекты, способные принимать различные значения. Они обладают следующими свойствами: в один момент времени переменная хранит не более одного значения; переменная имеет постоянный тип; переменная хранит текущее значение до записи в нее нового значения; в начале выполнения программы значение переменных считается неопределенными.

1 АЛФАВИТ И ИДЕНТИФИКАТОРЫ ЯЗЫКА С

Алфавит языка С включает следующие символы:

- латинские буквы строчные и прописные;
- цифры от 0 до 9;
- специальные знаки: " { } , | [ ] ( ) + – / % \ ; ' : ? < = > _ ! & # ~ ^ *.

Из символов алфавита формируются лексемы языка: идентификаторы, ключевые слова, константы, знаки операций, разделители.

В качестве идентификаторов в языке С допускаются любые сочетания символов, начинающиеся с буквы. Длина идентификатора не должна превышать тридцать два символа. Буквы верхнего и нижнего регистров считаются различными. Запрещено использование идентификаторов, совпадающих с ключевыми словами.

Ключевыми словами считаются идентификаторы, имеющие специальное значение для компиляторов языка С, их использование ограничено предопределенным смыслом.

1 Ключевые слова языка С

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

Помимо этого некоторые компиляторы языка С считают ключевыми слова **cdecl**, **far**, **huge**, **near, interrupt**.

Текст, заключенный между знаками /*…*/ или стоящий в строке после знаков //, считается комментарием.

2 ТИПЫ ДАННЫХ В ЯЗЫКЕ С

Типы данных служат для описания переменных. Имени переменной соответствует адрес участка памяти, выделенного для хранения значения переменной, а длина этого участка определяется выбранным типом данных. Тип переменной помимо этого определяет множество допустимых значений, которые может хранить переменная, а также набор операций, для которых переменная может служить операндом. Множество допустимых значений обычно совпадает с множеством допустимых констант того же типа. Существуют символьные, целочисленные и вещественные переменные. Причем, символьные переменные в языке С могут рассматриваться как целочисленные.

Переменные типизируются на основе *определений* и *описаний*. Определение, в отличие от описания, не только вводит программный объект, но и дает указание компилятору выделить участок памяти для его размещения в памяти ЭВМ. Для определений и описаний переменных используются следующие ключевые слова: **char**, **short**, **int**, **long**, **float**, **double**. Помимо этого как отдельно, так и с другими ключевыми словами могут использоваться слова **signed** и **unsigned**. Они обозначают знаковый или беззнаковый вид хранения целого числа.

2 Простые типы данных языка С

Тип	Размер, бит	Диапазон значений
char	8	-128...127
unsigned char	8	0...255
enum	16	-32768...32767
short	16	-32768...32767
unsigned short	16	0...65535
int	16	-32768...32767
unsigned int	16	0...65535
long	32	-2147483648...2147483647
unsigned long	32	0...4294967295
float	32	3.4e-38...3.4e+38
double	64	1.7e-308...1.7e+308
long double	80	3.4-4932...1.1e+4932

Особенностью типа **int** является то, что его длина в байтах соответствует ширине шины данных компьютера, для которого разработан используемый компилятор.

Типы данных **float**, **double** и **long double** считаются неупорядоченными.

Для описания типа возвращаемого функцией значения может быть использовано ключевое слово **void**, обозначающее отсутствие результата. В этом случае функция ничего не возвращает.

3 КОНСТАНТЫ В ЯЗЫКЕ С

Константа представляет собой языковую конструкцию, обозначающую изображение фиксированного числового, строкового или символьного значения. Они разделены на пять групп: целые, вещественные, перечислимые, символьные, строковые. Перечислимые константы обычно относятся к целочисленному типу данных.

Целая константа может быть десятичной, восьмеричной или шестнадцатеричной.

Десятичная константа представляет собой набор десятичных цифр, начинающийся с цифры, отличной от нуля, если это не ноль. Отрицательные константы представляют собой константы без знака, к которым применена операция изменения знака.

Восьмеричные константы всегда начинаются с нуля. В их записи недопустимо использование цифр 8 и 9.

Шестнадцатеричная константа начинается с сочетания '0x'. В ее состав могут входить цифры от 0 до 9 и буквы латинского алфавита от A до F (обозначающие числа от 10 до 15).

По умолчанию целочисленные константы относятся к типу **int**. Можно явным образом повлиять на выбор типа данных для константы. Для этого служат суффиксы **L**, **U** и **UL**. Суффикс **L** – соответствует типу **long**, **U** – типу **unsigned int**, **UL** определяет тип **unsigned long**.

Вещественная константа имеет другую форму представления, использующую арифметику с плавающей точкой. Константа с плавающей точкой может иметь семь частей: знак, целая часть, десятичная точка, дробная часть, признак экспоненты **e** или **E**, показатель десятичной степени, суффикс **L** или **F**. При отсутствии суффиксов вещественная константа относится к типу **double**. Суффикс **F** относит константу к типу **float**, а суффикс **L** – к типу **long double**.

Перечислимые константы вводятся с помощью служебного слова **enum**. По существу это обычные целочисленные константы, которым приписаны уникальные идентификаторы, не совпадающие с другими программными объектами и служебными словами. Каждой такой константе присваивается целочисленное значение. Первому идентификатору в списке присваивается значение нуля, а значение каждого следующего увеличивается на единицу.

П р и м е р 1 enum {zero, one, two, three};

Это правило действует и в том случае, когда идентификаторам явно присвоены значения (enum{ten = 10, three = 3, four, five}). Имена констант должны быть уникальными, однако к значениям констант это не относится, одно значение могут иметь несколько констант. Значения перечислимых констант могут быть заданы выражениями (enum {two = 2, four = two * 2}). Для перечислимых констант может быть введено имя соответствующего типа.

П р и м е р 2 enum tip {two = 2, tree = 3};

Строка представляет собой последовательность символов, заключенных в кавычки. Размещая строку в памяти, компилятор автоматически добавляет в ее конец нулевой код ('\0').

Символьные константы представляют собой один или два символа, заключенные в апострофы. Односимвольные константы относятся к стандартному типу **char**. Запись кодов и символов ', \, ?, " должна начинаться с символа '\'. Последовательности, начинающиеся с символа '\', называются *Esc-последовательностями*.

3 Esc-последовательности

\0	Null	\v	Табуляция вертикальная
\a	Bell(звонок)	\\	Обратный слэш
\b	Возврат на шаг	\’	Апостроф
\f	Перевод страницы	\”	Двойная кавычка
\n	Перевод строки	\?	Знак вопроса
\r	Возврат каретки	\000	Восмиричный код
\t	Табуляция горизонтальная	\xhh	Шестнадцатиричный код

4 ЗНАКИ ОПЕРАЦИЙ

Знаки операций обеспечивают формирование и последующее вычисление выражений. Один и тот же знак операции может употребляться в различных выражениях и по-разному восприниматься компилятором в зависимости от ситуации. Все знаки операций по количеству операндов делят на унарные, бинарные и трехразрядные. Бинарные операции бывают следующих видов: аддитивные, мультипликативные, сдвигов, поразрядные, операции отношений, логические, присваивания, адресной арифметики, операция "запятая".

Унарные операции (операции с одним операндом):

&	получение адреса операнда;
*	обращение по адресу (операндом должен быть адрес);
-/+	унарные операции изменения знака операнда ('+' – сделать операнд положительным);
~	поразрядное инвертирование внутреннего кода скалярного операнда;
!	логическое отрицание значения операнда (применяется к операндам скалярных типов), результат – 0 (если операнд отличен от нуля) и 1 (если операнд равен нулю); в качестве логических значений в языке С используются целые числа: 0 – ложь, ≠0 – истина;
++	увеличение скалярного операнда на 1 (инкремент); пре-

фиксная операция – увеличение значения операнда до его использования, постфиксная операция – увеличение операнда после его использования; операнд не может быть константой;

-- уменьшение скалярного операнда на 1 (декремент); правила оформления аналогичны инкременту;

Sizeof (*операнд*) вычисление размера в байтах для объекта того типа, который имеет операнд;

(*тип*) *операнд* операция явного преобразования типа операнда.

Бинарные операции:

Аддитивные операции:

+ арифметическое сложение операндов;

- арифметическое вычитание операндов.

Мультипликативные операции:

* умножение операндов;

/ деление операндов; для целочисленных операндов возвращается только целая часть от результата деления;

% получение остатка от деления целочисленных операндов.

Операции сдвига:

<< сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда;

>> сдвиг вправо (аналогично).

Поразрядные операции:

& поразрядная конъюнкция битовых представлений значений целочисленных операндов;

| поразрядная дизъюнкция битовых представлений значений целочисленных операндов;

^ поразрядное "исключающее или" битовых представлений значений целочисленных операндов.

Операции отношения:

<, >, <=, >=, == (равно), != (не равно).

Логические бинарные операции:

&& конъюнкция арифметических операндов или отношений;

|| дизъюнкция арифметических операндов или отношений.

Операции присваивания:

=, *=, /=, %=, +=, -=, <<=, >>=,&=, |=, ^=.

Операции адресной арифметики:

. играет роль операции адресной арифметики при прямом выборе компонента структурного объекта;

-> косвенный выбор компонента структурного объекта;

[ ] играет роль операции адресной арифметики при индексировании элементов массива;

( ) играет роль операции адресной арифметики при вызове функций.

В некоторых случаях в качестве знака операции может выступать запятая: несколько выражений, разделенных запятыми, будут вычисляться последовательно слева направо.

Условная (трехразрядная) операция:

операнд1 ? *операнд2* : *операнд3* – если значение *операнда1* истинно, то результат – *операнд2*, иначе – *операнд3*.

Для того, чтобы получить возможность однозначно вычислять результат при объединении в сложную операцию множества простых операций, введено понятие *приоритета выполнения операций*. Приоритет операции объясняет порядок ее выполнения в составе некоего выражения.

4 Приоритеты выполнения операций

Уровень приоритета	Операции	Порядок выполнения
1	[ ] . -> ( )	→
2	& * + – ++ -- sizeof() (*тип*)	←

3	* / % (мультипликативные)	→
4	+ – (аддитивные)	→
5	<< >>	→
6	< <= > >=	→
7	== !=	→
8	&	→
9	^	→
10	|	→
11	&&	→
12	||	→
13	? : (условная операция)	←
14	операция присваивания	←
15	,	→

5 РАЗДЕЛИТЕЛИ В ЯЗЫКЕ С

В составе языка программирования С присутствуют знаки пунктуации, использование которых позволяет изменять смысл выражения.

[ ] – ограничивают индексы массивов и индексированных элементов;

( ) – группируют выражения, используются в оформлении операторов, используются при оформлении функций, преобразовании типов, при оформлении макроопределений;

{ } – обозначают границы составного оператора или блока;

, – разделяет элементы списков;

; – завершает все операторы, все определения и все описания;

: – отделяет метку от помечаемого оператора;

... – обозначает переменное число параметров у функции при ее определении и последующем описании;

= – отделяет переменную от списка инициализации;

– обозначает директивы препроцессора.

6 ОПЕРАТОРЫ ЯЗЫКА С

Оператор представляет собой законченную фразу языка, содержащую полностью оформленный этап обработки информации. Любое выражение, заканчивающееся ';' воспринимается компилятором как оператор. В языке С представлены следующие виды операторов: *простые* – в состав которых не входят другие операторы (оператор присваивания, пустой оператор и операторы изменения порядка выполнения программы), а также *сложные* – составной оператор, оператор ветвления, переключатель, циклы.

- **Оператор присваивания** предназначен для оформления процесса обработки информации. Он имеет формат

$$\textit{имя переменной} = \begin{cases} \textit{константа} \ , \\ \textit{переменная} \ , \\ \textit{выражение} \ , \\ \textit{вызов функции} \ . \end{cases} \quad \textbf{;}$$

Переменная, расположенная слева от знака операции присваивания и являющаяся приемником результата, должна иметь тот же тип данных, что и значение результата.

- **Пустой оператор** используется в некоторых случаях для оформления других операторов. Он представляет собой точку с запятой, перед которой не указаны никакие действия:

```
;
```

При компиляции такая конструкция заменяется особым кодом, который требует пропустить выполнение текущего действия.

- **Составной оператор** представляет собой группу операторов, заключенных в фигурные скобки:

```
{
    операторы
}
```

Указанная в скобках последовательность действий воспринимается как одно действие. Это бывает необходимо сделать при записи составных операторов.

- **Условный оператор** управляет потоком обработки данных путем изменения последовательности передачи управления. В языке С условный оператор записывается следующим образом:

```
if (условие) оператор1; else оператор2;
```

Условие задается константой, переменной или выражением скалярного типа или указателем. Если результат проверки условия не равен нулю, то условие считается истинным и выполняется *Оператор1*. В противном случае выражение считается

ложным и выполняется *Оператор2*. Если операторов несколько, – нужно использовать составной оператор для их объединения. В качестве операторов нельзя использовать описания и определения переменных.

Схема работы условного оператора представлена на рис. 1.

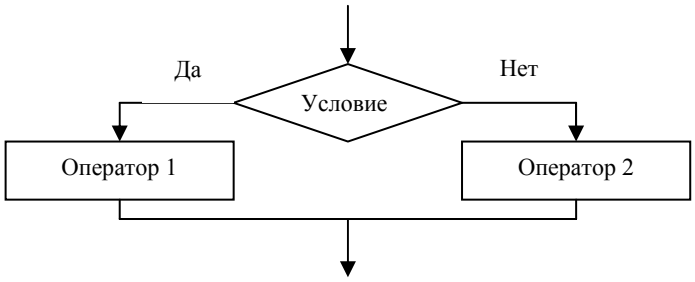


Рис. 1 Порядок работы условного оператора

Допускается сокращенная форма условного оператора:

if (условие) оператор1;

В этом случае ветвь, помеченная на рис. 1 надписью "нет" считается пустой и при ложном результате проверки условия не производится никаких действий, а управление передается следующему оператору.

При использовании вложенных друг в друга условных операторов, следует следить за отношением ветви **else**.

- **Переключатель** используется для организации множественного ветвления. В языке С оператор-переключатель записывается следующим образом:

```
switch (переключающее_выражение)
{
    case константное_выражение1: оператор1;
    case константное_выражение2: оператор2;
    .....
    default: оператор;
}
```

Управление передается к тому из помеченных с помощью ключевого слова **case** операторов, для которого значение *константного выражения* совпадает со значением *переключающего выражения*. Переключающее выражение должно быть целочисленным или его значение должно быть приведено к целочисленному типу. Значение константного выражения приводится к типу переключающего выражения. Все константные выражения должны быть различны, но принадлежать к одному типу. Любой из операторов в фигурных скобках может быть помечен несколькими метками.

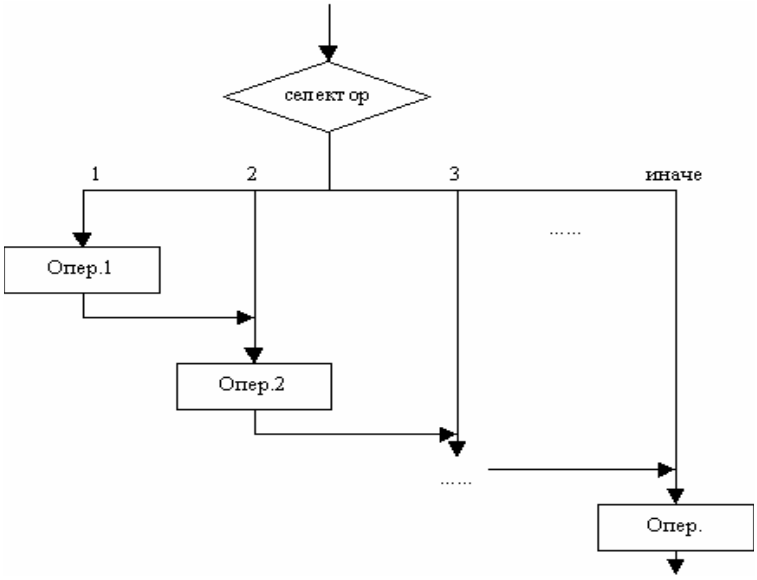


Рис. 2 Порядок работы переключателя

Порядок работы переключателя в языке С проиллюстрирован на рис. 2. После проверки значения переключательного выражения (селектора) осуществляется поиск помеченного ключевым словом **case** варианта с совпадающим значением константы. Первым будет выполнен оператор, помеченный данной меткой. Далее будут выполнены в порядке следования все остальные операторы, находящиеся в переключателе после найденного.

Указанный порядок работы оператора-переключателя не всегда удобен: гораздо чаще требуется выполнить только тот оператор, который помечен меткой, совпадающей в данный момент со значением селектора. Выполнение всех остальных операторов в данном случае не требуется. Этот режим работы переключателя представлен на рис. 3.

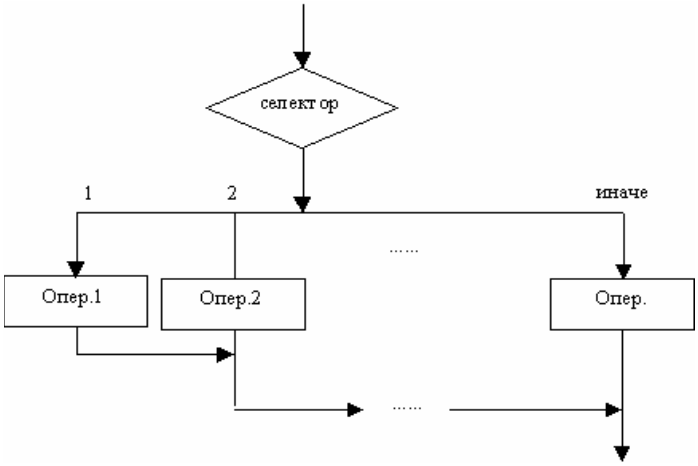


Рис. 3 Альтернативный порядок работы переключателя

Для приведения функционирования переключателя к этому виду последним действием в каждой ветви варианта необходимо использовать оператор **break**, изменяющий порядок работы оператора-переключателя.

В переключателе не должно быть операторов, стоящих перед первым оператором, помеченным с помощью ключевого слова **case**, так как они игнорируются.

Для задания многократного выполнения отдельных частей программы предусмотрены операторы циклов. Оператор, попадающий под действие оператора цикла, составляет *тело цикла*. Тело цикла не может быть описанием или определением. Для прекращения выполнения тела цикла, каждый раз после его выполнения осуществляется проверка *условия окончания* работы цикла, которое должно быть скалярным выражением. Если значение условия не равно нулю, то выполняется еще один шаг работы цикла. Работа цикла прекращается только тогда, когда результат проверки условия окончания станет равен нулю или в случае явной передачи управления за пределы тела цикла.

- **Оператор цикла с предусловием** требует сначала проверить условие окончания работы цикла. Если результат проверки не равен нулю, то выполняется тело цикла. В противном случае управление передается оператору, расположенному в тексте программы после оператора цикла. Порядок функционирования оператора цикла с предусловием проиллюстрирован на рис. 4. В языке С оператор цикла с предусловием записывается

while (условие) тело_цикла;

В качестве тела цикла может выступать один оператор.

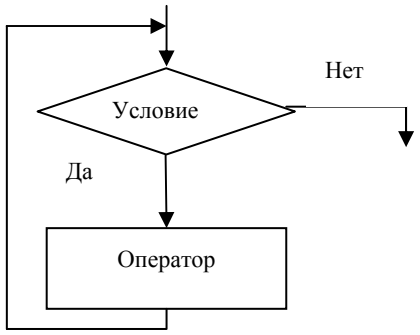


Рис. 4 Порядок работы цикла с предусловием

Если тело цикла образовано более чем одним оператором, то необходимо их объединение в составной оператор.

- **Оператор цикла с постусловием** устроен таким образом, что сначала выполняется тело цикла, а затем проверяется условие окончания работы цикла. Если результат проверки условия не равен нулю, то выполняется еще один шаг цикла. Порядок работы оператора цикла с постусловием иллюстрирует рис. 5. В языке С оператор цикла с постусловием записывается

do тело_цикла while (условие);

В теле цикла допускается использование одного оператора. Несколько операторов в теле цикла должны быть объединены в составной оператор.

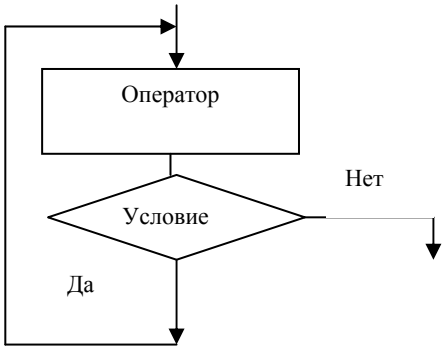


Рис. 5 Порядок работы цикла с постусловием

- **Итерационный цикл** предназначен для выполнения тела цикла заданное число раз.

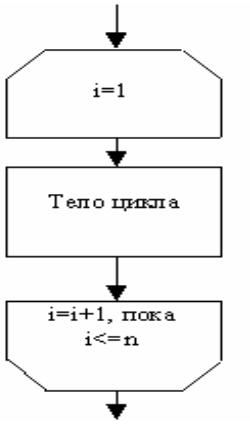


Рис. 6 Функционирование итерационного цикла

В языке С оператор итерационного цикла записывается

for (инициализация; условие_окончания; выражение) тело_цикла;

здесь *тело_цикла* – один оператор, задающий необходимые действия; *инициализация* – последовательность разделенных запятыми выражений, задающих начальные значения используемым в цикле переменным (инициализирующие выражения вычисляются при первом входе в цикл); *условие_окончания* то же, что и у ранее рассмотренных циклов; *выражение* – последовательность разделенных запятыми выражений, позволяющих изменить значения переменных между шагами работы цикла.

П р и м е р 3 for (i = 1, a = 0; i < 5; a += i++) ;

В результате работы приведенного в примере 3 итерационного цикла переменная *a* получит значение равное 10. Тело цикла в данном случае представлено пустым оператором.

При выполнении программы, передача управления между операторами осуществляется последовательно друг за другом в порядке их расположения в программе. Для изменения этого порядка передачи управления предусмотрена группа специальных операторов.

- **Оператор безусловного перехода** выглядит в языке С следующим образом:

goto метка;

здесь *метка* – обычный идентификатор языка С, расположенный в той же функции, что и сам оператор **goto**.

При обработке оператора безусловного перехода происходит передача управления от текущего оператора к оператору, перед которым находится указанная метка.

П р и м е р 4

```
goto a;
…
a: x = 1;
```

Не рекомендуется переходить извне внутрь блока, внутрь условного оператора, переключателя или цикла.

- **Оператор возврата из функции** предназначен для прекращения выполнения функции в указанной точке и возвращения управления функции, инициировавшей исполнение данной функции. Вид оператора зависит от описания типа возвращаемого функцией значения. Если был указан тип **void**, то оператор будет выглядеть

return;

Иначе оператор возврата из функции должен выглядеть следующим образом:

return выражение;

Тип *выражения* должен совпадать с типом результата функции.

- **Оператор принудительного выхода из цикла или переключателя** используется для немедленного прекращения выполнения тела цикла или переключателя с передачей управления следующему по порядку оператору:

break;

- **Оператор перехода к следующей итерации цикла** предназначается для прекращения выполнения текущего шага цикла с возвращением управления заголовку цикла для проверки условия окончания работы цикла:

continue;

Этот оператор употребляется только внутри операторов цикла.

- **Оператор объявления типов данных** используется для создания новых ассоциаций с уже существующими типами данных:

typedef спицификация_типа описание;

В результате работы оператора *описание* приобретает такие же свойства, что и *спецификация типа*, т.е. если спецификация типа является сложной и ей не совсем удобно пользоваться, то с помощью этого оператора возможно создать синоним, который будет ассоциироваться с указанным в операторе типом данных.

П р и м е р 5

```
typedef int integer;
```

В результате работы указанного в примере 5 оператора будет создана ассоциация *integer* со стандартным типом данных языка С **int**.

7 СТРУКТУРА ПРОГРАММЫ В ЯЗЫКЕ С

Важной отличительной особенностью любого языка программирования является порядок объединения отдельных операторов в программу, т.е. осмысленный текст, описывающий процесс обработки исходных данных для получения результата. Устройство программы во многом может повлиять на удобство программирования на том или ином языке как с точки зрения разработчика программы, так и с точки зрения компилятора, который переводит исходный текст программы в машинные коды.

В основе устройства программы на языке С лежат принципы *модульного программирования*. Согласно этим принципам программа разбивается на отдельные функции, которые вызывая друг друга осуществляют обработку исходных данных. С точки зрения языка С каждая функция представляет собой обособленный фрагмент программы, содержащий в себе последовательность некоторых операторов. Отдельные операторы из тела функции не доступны вне ее тела, поэтому у пользователя при работе с функцией складывается впечатление, что она выполняет некоторое большое абстрактное действие. Универсальность функций обеспечивается наличием у функции формальных параметров, посредством которых пользователю предоставляется возможность обрабатывать разные данные одной и той же функцией.

Программа на языке С состоит из совокупности следующих элементов: *директивы препроцессора*, *указания компилятору*, *описания* и *определения*. *Директивы препроцессора* определяют действия особой программы – *препроцессора* по преобразованию текста программы перед его компиляцией. *Указания компилятору* представляют собой инструкции, которыми руководствуется компилятор языка С во время работы. Порядок появления этих элементов в программе является существенным, так как он влияет на возможность использования переменных, функций и типов данных в различных частях программы.

Состав директив препроцессора и указаний компилятору будет рассмотрен позднее. Здесь отметим только то, что текст программы на языке С может быть разделен на несколько исходных файлов. При компиляции программы каждый файл должен быть скомпилирован отдельно, а затем связан с другими файлами компоновщиком. Однако отдельные исходные файлы можно объединить в один исходный файл, компилируемый как единое целое, посредством директивы препроцессора **#include**. Ее формат выглядит следующим образом:

#include <имя файла>
#include "имя файла"

В результате текст указанного в этой директиве файла будет записан в текущем файле на месте директивы. Если имя файла указано в угловых скобках, то препроцессор будет искать этот файл в директории со стандартными заголовочными файлами. Если имя файла указано в кавычках, то препроцессор будет искать файл в текущей директории.

Для того, чтобы программа могла быть скомпилирована и выполнена, она должна содержать как минимум одну функцию. Эта функция определяет действия, выполняемые программой, с нее начинается выполнение программы, она вызывает другие функции:

тип main (параметры функции)
{
тело функции
}

В качестве *типа* возвращаемого значения можно использовать типы **int** и **void**. **main** – стандартное название главной функции в программе на языке С. Главная функция, также как и любая другая, может получать аргументы. Однако то, что функция **main** при запуске программы вызывается операционной системой, накладывает ограничения на структуру этого списка. Его состав рассматривается в части этой книги, посвященной функциям.

Приведем пример программы на языке С.

П р и м е р 6

```
#include <stdio.h>
int main()
{
        printf("Здравствуй, мир!!!");
        return 0;
}
```

8 ОПРЕДЕЛЕНИЯ И ОПИСАНИЯ ПРОГРАММНЫХ ОБЪЕКТОВ. ПРОДОЛЖИТЕЛЬНОСТЬ СУЩЕСТВОВАНИЯ ПРОГРАММНЫХ ОБЪЕКТОВ

В процессе работы программам практически всегда необходимо осуществлять временное хранение некоторой информации в промежутках между ее использованием в операциях. Этой цели служат *программные объекты*. Программным объектом в языке С принято считать некоторую область оперативной памяти ЭВМ, занятую информацией. Для обращения к этой области памяти программе необходимо знать адрес ее начала и длину в байтах. Одним из частных случаев программного объекта является переменная. Для создания переменных в языке С используются *определения* и *описания*. Определение сообщает компилятору о необходимости немедленного размещения в оперативной памяти программного объекта, задает его имя и атрибуты. Описание же только сообщает свойства программного объекта с которыми ассоциируется его имя, так как сам объект создан в другой части программы, с которой налаживает связи компоновщик после компиляции.

Определение переменных заданного типа в языке С имеет следующий формат:

спецификатор модификатор тип имя = инициализатор;

здесь *спецификатор* – описатель класса памяти; *модификатор* – описатель особых свойств объекта; *тип* – описатель одного из основных типов данных (см. раздел 2); *имя* – идентификатор, задающий название программного объекта; *инициализатор* – возможное определение начального значения программного объекта.

Описание объекта становится определением, если описывает переменную; содержит инициализатор; полностью описывает функцию (включая ее тело); описывает структуру или объединение вместе с их компонентами.

В языке С определены три типа продолжительности существования программных объектов: локальная, статическая и динамическая. *Локальная продолжительность существования* связана с функцией

или *блоком* (составным оператором, внутри которого определяются новые программные объекты). Все локальные объекты создаются только в момент начала обработки функции или блока и уничтожаются при завершении их работы. *Статическая продолжительность существования* также связана с функциями или блоками. Создаются такие объекты в момент первого обращения к функции или блоку, а уничтожаются они в момент завершения работы программы. Объекты *с динамической продолжительностью существования* создаются отдельно в процессе работы программы и удаляются из памяти ЭВМ с помощью специальных функций.

Продолжительность существования программного объекта и его размещение в памяти ЭВМ задает *класс памяти*. Для явного определения класса памяти объекта используются спецификаторы. Спецификаторы классов памяти языка С перечислены в табл. 5.

5 Спецификаторы класса памяти

auto	Задается только внутри блоков. Этим объектам память выделяется при входе в блок и освобождается при выходе из него автоматически. Вне блока объекты класса auto не существуют
register	Аналогичен классу auto, но для размещения значений объектов используются регистры процессора, а не основная память. В случае отсутствия свободной регистровой памяти компилятор обрабатывает объекты класса register подобно классу auto

static	Статическая продолжительность существования. Объект будет существовать в пределах файла, с исходным текстом которого он определен. Класс static может приписываться переменным или функциям
extern	Такой объект глобален, то есть он создан вне функций или блоков и доступен во всех модулях программы. Класс может быть приписан переменной или функции

В языке С определены три типа *области действия идентификаторов* – блок, функция и файл. Область действия идентификаторов представляет собой часть программы, в которой идентификатор может быть использован для доступа к описываемому объекту. Если идентификатор определен в блоке или функции, то область его действия – от точки описания до конца блока или функции. Файл с текстом программы является сферой действия всех глобальных имен, т.е. имен объектов, описанных вне любых функций. Каждое глобальное имя действует от точки описания, до конца файла. С их помощью удобно связывать функции по данным, т.е. создавать "общее поле данных".

Имена переменных и функций, названия определенных пользователем типов и имена элементов перечислений должны быть уникальными в границах своей области действия.

Модификаторы способны задать программному объекту особые свойства. Они перечислены в табл. 6.

6 Модификаторы языка С

const	Значение переменной нельзя изменять во время работы программы. Оно должно быть задано инициализацией при определении
volatile	Значение объекта может быть изменено в промежутках между явными обращениями к нему в программе

Инициализатор способен задать начальное значение переменной еще на стадии компиляции. Он отделяется от имени объекта знаком присваивания. Для объектов, определенных вне функций и блоков, инициализатором может выступать константа или уже созданная и инициализированная переменная соответствующего типа. Если же объект локален, то в качестве инициализатора может выступать еще и выражение, возвращающее результат соответствующего объекту типа.

9 ОПЕРАЦИИ ВВОДА-ВЫВОДА В ЯЗЫКЕ С

Операции ввода-вывода в языке С осуществляются внешними функциями, хранящимися в стандартной библиотеке. Для их использования в начале программы нужно указать директиву

#include <stdio.h>

Она включает в текст программы заголовочный файл, содержащий описания соответствующих функций:
int getchar(void); – считывает символ из стандартного потока ввода;

int putchar(int); – записывает символ в стандартный поток вывода;

char *gets(char *); – считывает строку из стандартного потока ввода и помещает ее по указанному в качестве аргумента адресу;

int puts(char *); – помещает строку в стандартный поток вывода;

int scanf(char *format, ...); – форматированный ввод из стандартного потока ввода;

В строке *format* необходимо задать описание типов данных всех переменных, значения которых необходимо считать из стандартного потока ввода. Список переменных размещается после строки *format*. К каждой переменной должна быть применена операция взятия адреса. Описания типов должны быть расположены друг за другом без разрывов. Спецификации формата обозначаются символом **%** и могут иметь следующий состав: %[*][width][F|N][h|I|L]type. Здесь

* – запрещает присвоение полученного аргумента по указанному адресу переменной;
width – положительное десятичное целое число, указывающее что должно быть считано количество символов не более, чем задано;
F|*N* – учитывают способ адресации в используемой модели памяти;
h|*I*|*L* – short/long вариант поля type.
В качестве описателей типов данных используются символы:
D,d – десятичное целое;
U,u – десятичное целое без знака;
O,o – восьмеричное целое без знака;
X,x – шестнадцатеричное целое без знака;
i – любое целое;
e, f, g – значение с плавающей точкой;

c – символ;
s – строка.

int printf (char *format, ...); – форматированный вывод в стандартный поток вывода.

В строке *format* необходимо задать описание типов данных всех переменных, значение которых необходимо поместить в стандартный поток вывода. Помимо этого в строке *format* можно располагать константные величины, предназначенные для вывода. Список переменных размещается после строки *format*. Спецификации формата обозначаются символом **%** и могут иметь следующий состав: %[flag][width][.precision][F|N|h|l]type. Здесь

flag – управление печатью знаков ('-' – выравнивание по правой границе, '+' – добавление знаков '+' или '-');
width – общая длина поля;
precision – длина поля после запятой.

В качестве описателей типов данных используются символы:
d, i – десятичное целое со знаком;
u – десятичное целое без знака;
o – восьмеричное целое без знака;
x – шестнадцатеричное целое без знака;
f – число с плавающей точкой;
e – экспоненциальная форма чисел с плавающей точкой;
g – компактная форма чисел с плавающей точкой;
c – символ;
s – строка.

П р и м е р 7

```
int i;
double f;
scanf("%d%lf", &i, &f);
printf("Значения переменных i= %d  f= %g", i, f);
```

Следует обратить внимание на то, что функции scanf следует в качестве аргументов передавать адреса переменных, в которых предполагается хранить полученные из потока ввода данные.

10 ПОНЯТИЕ АДРЕСАЦИИ. УКАЗАТЕЛИ В ЯЗЫКЕ С

Логическая основа функционирования современной вычислительной техники базируется на принципах, сформулированных американским математиком Джоном фон Нейманом. Один из них – *принцип произвольного доступа к основной памяти* – гласит, что основная память ЭВМ разбивается на дискретные элементы, называемые *ячейками памяти*. В соответствии с этим принципом процессору в любой момент времени должна быть доступна любая ячейка памяти. В программах ячейкам памяти, в которых хранится обрабатываемая программой информация, присваиваются имена, посредством которых осуществляется доступ к их содержимому. Другой принцип называется *принципом хранимой программы*. Он гласит о том, что программа решения некоторой задачи должна храниться в основной памяти ЭВМ вместе с обрабатываемыми данными. Поэтому подготовленная к исполнению компьютером программа должна быть переведена компилятором на язык, понятный машине, т.е. в машинные коды.

Исполняемая программа в машинных кодах обычно состоит из трех частей:

1) *сегмент кода* – хранит машинные коды команд, составляющих программу;
2) *сегмент данных* – находятся ячейки памяти, хранящие обрабатываемые данные;
3) *стек программы* – особая область памяти, используемая программой для специфических операций по временному хранению информации.

От того, каким образом исполняемая команда получает доступ к информации, зависит универсальность программы. *Способ адресации* определяет правила доступа команды к данным. Рассмотрим основные способы адресации.

Если обрабатываемая информация входит непосредственно в код исполняемой команды, то такой способ адресации называется *непосредственной адресацией*. Команде в данном случае не нужно затрачивать время на загрузку данных из ячеек сегмента данных в регистры процессора, так как она попадает туда одновременно с кодом команды. Примером может служить оператор

```
a = 1;
```

здесь число 1 находится в режиме непосредственной адресации относительно операции присваивания. Недостатком данного вида адресации является то, что в момент исполнения программы это число изменить практически невозможно. Каждый раз при возникновении потребности менять данные придется изменять текст исходной программы.

Можно хранить данные в регистрах процессора. Такой способ адресации носит название *прямой адресации*. В языке С при определении переменных можно использовать спецификатор **register**. Однако процессоры обладают недостаточно большим количеством внутренней рабочей памяти (регистров), для того, чтобы помимо информации, обеспечивающей нормальную работу самого процессора, еще и осуществлять длительное хранение промежуточных данных программы. Поэтому этот способ адресации данных в целом является малоэффективным.

Вместо регистров процессора для хранения информации необходимо использовать ячейки основной памяти. Здесь возможны два случая. В первой команде сообщается машинный адрес ячейки памяти, хранящей нужную ей информацию. Этот способ адресации носит название *абсолютной адресации*. Сама ячейка находится в сегменте данных. Способ довольно удобный, но команда может обратиться только к одной ячейке памяти, адрес которой ей сообщен.

Гораздо большей эффективностью будет обладать способ адресации, при котором команда получит адрес ячейки памяти в виде суммы двух элементов: некоторого базового адреса и относительного смещения. Эти элементы записываются в коде команды. Такой способ адресации получил название *косвенной адресации*. При использовании этого способа команды получают возможность обращаться к целому диапазону ячеек памяти.

Таким образом, мы подошли к особому типу данных, позволяющему хранить информацию непосредственно в том виде, в каком ее воспринимает и обрабатывает вычислительная техника. Этот тип данных называется *указатель*. Указатели представляют собой программные объекты, содержащие адреса других программных объектов. Они могут быть указателями-переменными и указателями-константами. Указатели подразделяются на указатели на функции и указатели на другие программные объекты. Это разделение связано с различиями в свойствах и правилах их использования.

В простейшем случае определение и описание указателя выглядит следующим образом:

тип *имя_указателя = инициализирующее_выражение;

здесь *тип* – базовый тип указателя; *имя указателя* – идентификатор; * – унарная операция обращения по адресу (разыменования); *инициализирующее выражение* – возможный инициализатор соответствующего типа данных.

В совокупности знак '*' и имя типа воспринимаются как обозначение особого типа данных "указатель на объект данного типа". В качестве инициализирующего выражения должно выступать константное выражение, представленное:

- явно заданным адресом участка памяти;
- указателем, уже имеющим значение;
- выражением, позволяющим получить адрес объекта с помощью операции получения адреса программного объекта '&'.

Для обозначения "пустого" указателя используется специальная константа **NULL**.

П р и м е р 8

```
char ch = 'y';
char *pch =&ch, *pc = pch;
int *i =NULL;
```

В результате появилась ячейка памяти, к которой можно обращаться непосредственно через ее имя *ch* или при помощи указателя. В последнем случае необходимо использовать операцию '*' – обращение по адресу (разыменование). Попытка разыменования пустого указателя не имеет смысла.

Для того, чтобы связать указатель с некоторой областью памяти, необходимо пользоваться стандартными функциями, прототипы которых находятся в заголовочном файле **stdlib.h**:

void *malloc(unsigned n); – возвращает указатель на блок динамически распределяемой памяти длиной *n*-байт;

void *calloc(unsigned m, unsigned n); – возвращает указатель на начало области динамически распределяемой памяти для размещения *m* элементов по *n*-байт каждый;

void *realloc(void *blok, unsigned n); – изменяет размер блока ранее выделенной динамической памяти с адресом начала *blok* до размера *n*-байт;

void free(void *blok); – освобождает ранее выделенный блок динамической памяти.

В случае неудачи функции *malloc*, *calloc* и *realloc* возвращают значение NULL.

При определении указателя и он сам и его значение могут быть объявлены константами. Ближний к имени указателя модификатор **const** говорит о постоянстве значения указателя. Такой указатель должен получить значение при инициализации:

тип const *const имя_указателя = инициализатор;

Если же указать **const** перед операцией разыменования '*', то это говорит о постоянстве значения, на которое указывает указатель.

П р и м е р 9

```
void main (void)
{
        char *const key = (char *) 0x 0417;
        *key = 'E';
}
```

В результате работы программы из примера 9 в операционных системах MS DOS и WINDOWS 9X будут включены индикаторы Caps Lock, Num Lock и Scroll Lock на клавиатуре. При этом в программе нельзя изменять значение указателя *key*.

В языке С указатели тесно связаны с базовыми типами данных. При организации доступа к памяти с помощью операции разыменования указателя компьютеру необходима информация не только о размещении, но и о размерах используемого участка памяти. Эту информацию компьютер получает из базового типа указателя. В качестве типа может быть использован как простой, так и сложный тип данных (массив, указатель, структура, объединение, созданный пользователем тип).

П р и м е р 10

```
#include <stdio.h>

void main()
{
        long L = 0x12345678;
        char *c = (char *) &L;
        short *s = (short *) &L;
        long *l = (long *) &L;
        printf("*c = %x", (int) *c);
        printf("\n*s = %x", *s);
        printf("\n*l = %x", *l);
}
```

В результате работы этой программы на печать будут выведены следующие результаты:

```
*c =78
*s =5678
*l =12345678
```

Исключение составляет тип **void** (отсутствие значения). Указатель типа **void *** отличается от других указателей отсутствием сведений о размере соответствующего ему участка памяти. При его использовании обязательно необходима операция приведения типа.

П р и м е р 11

```
#include <stdio.h>
void main()
{
        void *v;
        int i = 10;
        double d = 10.1;
        v = &i;
        printf("i = %d", *(int *) v);
        v = &d;
        printf("d = %g", *(double *) v);
}
```

Возможности связывания указателей типа **void *** с объектами разных типов называется "родовым программированием". Разрешено неявное преобразование любого константного указателя к типу **void ***.

Поскольку с точки зрения компьютера указатель хранит такие же данные как и любая другая переменная, то к ним разрешается применять операции по преобразованию информации с учетом смысла указателя. Над указателями разрешается выполнять следующие действия:

- разыменование;
- преобразование типов;
- присваивание;
- получение адреса;
- аддитивное сложение и вычитание;
- инкремент и декремент;
- операции отношения.

Аддитивные операции могут применяться по отношению к указателю и константе и иногда к двум указателям.

Наиболее полно для указателей реализована операция вычитания, которая применима к указателям на объекты одного типа или к указателю и целочисленной константе. Вычитая два указателя можно определить "расстояние" между двумя участками памяти в единицах, кратных длине в байтах объекта того типа, к которому отнесен указатель. Вычитая из указателя целочисленную константу *k*, можно переместиться к другой ячейке памяти на расстояние, равное в байтах

k * sizeof(тип).

Операция сложения относительно указателей реализована в языке С беднее: можно складывать указатель и целочисленную константу *k*. При этом происходит перемещение к ячейке памяти, отстоящей от исходной на расстояние, равное в байтах

k * sizeof(тип).

П р и м е р 12

```
short i, *Pi = &i;
Pi = Pi + 2;
```

В этом примере произойдет смещение указателя на четыре байта.

Инкремент (++) и декремент (--) указателей полностью аналогичен операциям уменьшения и увеличения указателя на единицу, т.е. указатель перемещается к соседней ячейке памяти на расстояние длиной sizeof(тип).

В некоторых случаях указатель необходимо переместить к ячейке памяти, начинающейся с адреса, не кратного размеру участка памяти относительно его базового типа, для чего необходимо в качестве операндов соответствующих операций использовать числовые значения указателей.

П р и м е р 13

```
int i, *Pi, *P1;
Pi = &i;
P1 = (int *) ((int) Pi + 1);
```

11 СЛОЖНЫЕ ТИПЫ ДАННЫХ В ЯЗЫКЕ С

В начале развития программирования очень часто программы обрабатывали переменные, содержавшие только одно значение. Однако сразу же выявились недостатки подобного метода разработки программ: для большого числа однотипных данных необходимо было держать большое число программных объектов со своими именами. Выходом из сложившейся ситуации стало введение сложных типов данных, которые в языке С образованы на основе указателей. Посредством косвенной адресации, используя константный указатель как базовый адрес, стало возможным, используя одно имя переменной, адресоваться к целой группе ячеек памяти, которые занимают непрерывный участок в памяти ЭВМ. Далее рассмотрены сложные типы данных языка программирования С (массив, структура, объединение, файл) и основы работы с ними.

11.1 Массивы

Массив представляет собой объединение конечного числа однотипных данных, хранящихся в памяти компьютера непосредственно друг за другом. Для доступа к конкретному элементу массива необходимо указать его номер в последовательности. В основе идеи массивов лежит заимствованное из математики понятие матрицы.

Определение одномерного массива в языке С выглядит следующим образом:

тип имя_массива [константное_выражение];

здесь *тип* – базовый тип элементов массива; *имя массива* – идентификатор; *константное выражение* – определение размера массива (в некоторых случаях может отсутствовать).

Допускается описание массива без явного указания количества элементов в нем:

а) при описании внешнего массива, определенного в другой части программы,

```
extern long A[];
```

б) при явной инициализации массива в случае его определения

```
int A[] = {10,50,1,100};
```

в этом случае число элементов подсчитывается при обработке компилятором списка инициализации.

Массивы, как и простые переменные, могут инициализироваться при определении. Число элементов в списке инициализации не должно превышать размеры массива:

```
int A[5] = {0,1,2,3,4};
short MASSIW[5] = {4,2,0};
```

В случае, если количество элементов в списке инициализации меньше, чем размерность массива, начальные значения приписываются первым по счету элементам массива.

Необходимо отметить различное поведение массивов на этапах определения (описания) и использования. При определении массива ему выделяется память, а его имя ассоциируется со всем массивом. При использовании массива его имя воспринимается как константный указатель базового типа элементов массива. Исключение составляет операция **sizeof(имя_массива)**. Она вычисляет длину в байтах всего участка памяти, выделенного под массив. Особенность операции взятия адреса (&), примененной к имени массива, является:

имя_массива == &имя_массива == &имя_массива[0].

Доступ к элементам массива осуществляется с помощью индексированных переменных:

имя_массива[индекс]

Для первого элемента массива индекс равен нулю. В записи индексированной переменной квадратные скобки являются операцией с двумя операндами, выполняемой по правилам адресной арифметики. Здесь *имя массива* – константный указатель, являющийся адресом начала массива в основной памяти ЭВМ, а *индекс* – выражение целого типа, являющееся смещением от начала массива. Эта операция аналогична операции обращения по адресу:

*(имя_массива + индекс).
Так как сложение коммутативно, то эта операция эквивалентна записи

*(индекс + имя_массива).

Таким образом, выражение *имя_массива[индекс]* адресует тот же элемент в памяти, что и выражение *индекс[имя_массива]*.

Многомерный массив в языке С представляется как массив массивов. Определяется он следующим образом:

тип имя_массива [N1][N2];

Разрешается создавать массивы до тридцать второй степени вложенности.

Многомерные массивы инициализируются по строкам по возрастанию номеров элементов.

П р и м е р 14

```
int A[2][3] = {0, 1, 2, 3};
```

В указанном примере элементы массива получат следующие значения: A[0][0] = 0; A[0][1] = 1; A[0][2] = 2; A[1][0] = 3;.

В случае, когда надо изменить порядок инициализации элементов массива, используют фигурные скобки, выделяющие элементы, предназначенные для одной строки.

П р и м е р 15

```
int A[4][3] = {{0}, {1, 2}, {3, 4, 5}};
```

здесь элементы списка инициализации расположатся в следующем порядке: A[0][0] = 0; A[1][0] = 1; A[1][1] = 2; A[2][0] = 3; A[2][1] = 4; A[2][2] = 5; остальные элементы массива начальных значений не получили.

При обращении к элементам массива можно пользоваться операцией обращения по адресу. Для элемента массива *B[i][j][k]* такая конструкция примет вид

((*(B + i) + j) + k) или *(B[i][j] + k).

Устройство массивов в языке С позволяет связать с указателем как явно определенный массив, так и динамический массив. В любом случае массив будет представлять собой последовательность ячеек известной длины указанного базового типа, а указатель связывается с адресом первого элемента массива.

П р и м е р 16

```
int A[5], *Pa = A;
```

Доступ к элементам массива с помощью указателя может осуществляться двумя способами:

1) с помощью операции разыменования указателя '*'

```
*Pa;
```

здесь необходимо перемещаться по области памяти, выделенной под массив с помощью аддитивных операций '+' и '-' или унарных '++' и '--'. Этот метод неприемлем для динамического массива, так как может быть утерян адрес его начала;

2) с помощью операции индексирования элементов массива '[ ]'

```
Pa[i];
```

Так как имя *A* ассоциируется с адресом начала массива в памяти, а указатель *Pa* содержит этот адрес, то операция индексирования может одинаково применяется к этим объектам. *A[0]* адресует тот же элемент массива, что и *Pa[0]*.

Для работы со стоками предусмотрено несколько функций, описанных в заголовочном файле **string.h**:

char *strcat(char *Sp, char *Si); – приписывает строку *Si* к строке *Sp*;

int strcmp(char *Str1, char *Str2); – сравнивает строки *Str1* и *Str2*;
Результат будет меньше нуля, если *Str1* < *Str2*, будет равен нулю, если строки равны и будет больше нуля, если *Str1* > *Str2* .

unsigned strlen(char *Str); – вычисляет длину строки в байтах.

В заголовочном файле **stdio.h** описаны функции:

int sscanf(char *S, char *format [, argument,...]); – функция форматированного извлечения информации из строки *S*;

int sprint(char *S, char *format [, argument,...]); – функция форматированной записи информации в строку *S*.

Приведем пример использования указателя для связи с динамическим массивом.

П р и м е р 17

```
int n = 5;
int *Pa = (int *) malloc(n * sizeof(int));
```

Созданный динамический массив не имеет своего имени, поэтому необходимо запомнить адрес его начала в указателе, чтобы потом получить возможность использовать его элементы.

Устройство одномерного динамического массива схоже с устройством явно определенного одномерного массива. Однако, устройство многомерных динамических и явно определенных массивов различно.

Рассмотрим устройство многомерного динамического массива на примере двумерного массива. При создании массива сначала необходимо разместить в памяти одномерный массив, каждый элемент которого указывает на строку, хранящую данные, а затем создать сами строки для хранения информации. Организация такого массива показана на рис. 7, а создание массива приведено в примере 18.

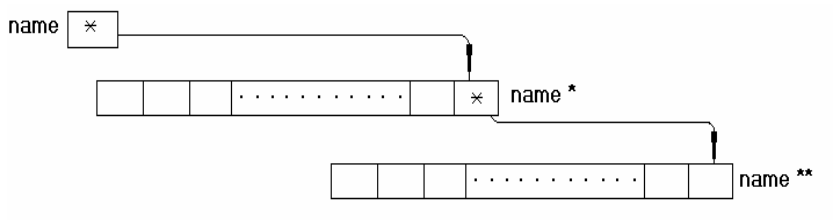


Рис. 7 Устройство двумерного динамического массива

П р и м е р 18

```
int **a, n, m, i;
scanf("%d%d", &n, &m);
a = (int **) malloc(n *sizeof(int *));
for (i = 0 ; i < n; i++) a [i] = (int *) malloc(m *sizeof(int));
```

Удаление элементов динамического массива производится в порядке, обратном процессу их создания.

П р и м е р 19

```
for (i = 0 ; i < n; i++) free(a[i]);
free(a);
```

11.2 Структуры

Структура представляет собой объединение в единую ячейку памяти конечного множества именованных элементов разного типа. В отличие от массива, где все элементы имеют единое имя, а доступ к ним осуществляется путем уточнения имени элемента, в структуре каждый элемент обладает уникальным именем, посредством которого осуществляется доступ к его содержимому. На тип элементов, входящих в структуру, ограничений нет, т.е. в состав структуры могут входить как переменные простых типов, так и массивы, указатели, структуры и объединения. Элементы структур называют *полями*.

В языке С определение структурного типа выглядит следующим образом:

```
struct имя_структурного_типа
{
    тип_1 идентификатор_1;
    тип_2 идентификатор_2;
    ...
    тип_n идентификатор_n;
} имя_переменной;
```

При описании структуры возможно пропустить или имя структурного типа или имя переменной. В зависимости от этого данная запись будет восприниматься как определение, либо как описание структуры. Имя структурного типа в дальнейшем может быть использовано для описания переменных. Для этого используется сокращенное имя структуры.

```
struct имя_структурного_типа имя_1;
```

Каждая создаваемая структурная переменная содержит свои собственные данные, а их состав определяется структурным типом. Для доступа к элементам конкретной структуры используется уточненное имя, которое в общем виде выглядит следующим образом:

```
имя_структуры.имя_элемента;
```

При определении структуры возможна ее инициализация:

```
struct
{
    double re, im;
} complex = {0.2, 3.1};
```

Имя структурного типа обладает всеми свойствами имен типов данных, поэтому можно определить указатель на структуру:

struct имя_структурного_типа *имя_указателя;

Значением указателя на структуру станет номер байта, с которого начинается размещение структуры в памяти.

Обратиться к полям структуры, используя имя указателя на нее, можно двумя способами:

1) (*имя_указателя).поле;
2) имя_указателя->поле.

Для полей структур существует только одно ограничение: поле структуры не может иметь тот же тип, что и тип самой структуры.

```
struct A {struct A a; int i;};   // Это запрещено
```

Однако допускается использование указателя на этот тип:

```
struct A {struct A *a; int i;};
```

Возможно использование перекрестных указателей структур друг на друга:

```
struct A; // Неполное определение структурного типа
struct B {struct A *Pa;};
struct A {struct B *Pb;};
```

11.3 Объединения

Объединения в языке С очень похожи на структуры. Главным их отличием является то, что все поля объединения начинаются с одного адреса, а общая его длина равна длине в байтах наибольшего поля. Определение объединения выглядит следующим образом:

```
union имя_типа_объединения
{
        тип1 поле1;
        тип2 поле2;
        …
        типn полеn;
} переменная_объединение;
```

Обращение к полям объединений осуществляется с помощью уточненных имен:

имя_объединения.имя_элемента;

Определение указателей на объединения и работа с ними осуществляется аналогично подобным операциям со структурами.

11.4 Битовые поля структур и объединений

В составе структур и объединений могут в качестве компонентов использоваться *битовые поля*. Каждое такое поле представляет собой целочисленное значение, занимающее в памяти фиксированное число битов. Битовые поля не имеют адресов, а следовательно, они не могут объединяться в массивы, на них нельзя объявить указатель. Главное назначение битовых полей – обеспечение удобного доступа к отдельным битам. Определение структуры с битовыми полями в языке С выглядит следующим образом:

```
struct
{
        тип_поля1 имя_поля1 : ширина_поля;
        тип_поля2 имя_поля2 : ширина_поля;
        …
        тип_полян имя_полян : ширина_поля;
} имя_структуры;
```

здесь *тип_поля* – целочисленный тип данных; *ширина_поля* – целое десятичное число без знака.

Обращение к битовым полям структур и объединений осуществляется так же, как и к их обычным полям (см. раздел 11.2).

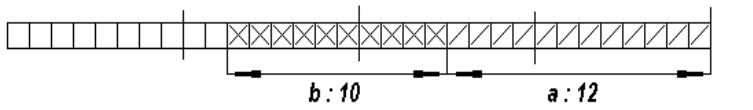


Рис. 8 Размещение битовых полей в структуре без выравнивания

П р и м е р 20

```
struct
{
        short a: 12;
        short b: 10;
} Pr;
```

Существуют два варианта размещения битовых полей в оперативной памяти ЭВМ. Если в структуре или объединении нет дополнительных управляющих элементов, то битовые поля будут расположены компилятором друг за другом. Рассмотрим пример 20. В структуре *Pr* определены два поля. Общая длина структуры составляет четыре байта, однако, использовать для хранения информации можно только первые двадцать два бита. Остальные биты остаются неиспользуемыми. Этот порядок размещения полей проиллюстрирован на рис. 8.

Часто реализация компиляторов позволяет изменить порядок размещения битовых полей: их можно выравнивать по границам байтов. Для выравнивания битовых полей в структуру вводят неиспользуемые биты, формирующие промежуток между полями. Для их создания вводят поля без имени.

П р и м е р 21

```
struct
{
    short a: 12;
    short : 4;
    short b: 10;
} Pr ;
```

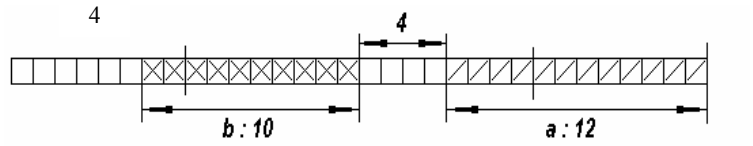


Рис. 9 Размещение битовых полей в структуре с выравниванием

В примере 21 между полями *a* и *b* находится поле из четырех бит. Так как это поле не имеет имени, то обратиться к нему невозможно. Главное назначение этого поля – создать промежуток между именованными битовыми полями, чтобы их размещение в оперативной памяти совпадало с размещением обычных полей. Этот порядок иллюстрирует рис. 9.

П р и м е р 22

Применение битовых полей.

```
union
{
        unsigned char c;
        struct
        {
                unsigned b0: 1;
                unsigned b1: 1;
                unsigned b2: 1;
                unsigned b3: 1;
                unsigned b4: 1;
                unsigned b5: 1;
                unsigned b6: 1;
                unsigned b7: 1;
        } byte;
} cod;
…
cod.c = 'a';
printf("%u", cod.byte.b4);
```

В примере 22 показана возможность обращения к отдельным битам поля *c*, которое хранит код символа.

11.5 Файлы

Файл представляет собой последовательность данных произвольной длины, которая имеет некоторую логическую организацию и хранится на внешнем носителе. Особенностью языка С является отсутствие в самом языке средств работы с файлами и, как следствие, заранее сформированных представлений о структуре файлов. Поэтому все файлы рассматриваются как последовательность байтов, заканчивающаяся особым кодом, обозначающим их окончание. Функции языка С позволяют осуществлять потоковый ввод-вывод и ввод-вывод нижнего уровня. При потоковом вводе-выводе обмен данными производится побайтно, но возможно работать с данными различных размеров и форматов посредством организации буферизованного потока. Таким образом, можно сказать, что *поток* – это файл вместе со средствами буферизации. Для работы с потоком предусмотрен ряд функций, описанных в заголовочном файле **stdio.h**.

Запущенная на выполнение программа на языке С автоматически связывается с пятью потоками:

1) **stdin** – стандартный поток для ввода данных;
2) **stdout** – стандартный поток для вывода данных;
3) **stderr** – стандартный поток для сообщений об ошибках;
4) **stdaux** – асинхронный порт;
5) **stdprn** – порт принтера.

Потоки stdaux и stdprn могут быть не установлены.

Все остальные потоки необходимо явно открывать в процессе работы программы. В исполняемой программе поток связывается со структурой типа **FILE**, определенной в файле **stdio.h**. В структуре **FILE** содержатся компоненты, при помощи которых осуществляется работа с потоком. При открытии потока в программу возвращается указатель на поток, соответствующий указателю на объект типа **FILE**. Указатель, связывающий исполняемую программу с файлом, должен быть объявлен в виде

FILE *имя_потока;

Связать указатель с файлом для ввода-вывода данных через поток можно с помощью функции, имеющей следующий описатель:

```
FILE *fopen(const char *filename, const char *mode);
```

здесь *filename* – указатель на строку со спецификацией открываемого файла; *mode* – указатель на строку символов, описывающую режим работы с файлом.

Если файл открыть не удалось, функция *fopen* возвращает значение **NULL**.

Режим работы с файлом во-первых определяет что произойдет с файлом в момент его открытия и какие особенности будет носить работа с ним. Возможные варианты открытия потока в программе приведены в табл. 7.

7 Варианты открытия потоков

Описатель	Режим работы
r	Файл открывается в режиме чтения. Он должен существовать. Иначе возвращается значение NULL
w	Файл открывается для записи. Если указанный файл существует, то его содержимое стирается. Если файл не существует, то он создается в указанной директории. В случае неудачи возвращает значение NULL
a	Режим дополнения. Файл открывается для записи, но его содержимое не стирается, а указатель устанавливается на байт с кодом "конец файла". Если файл не существовал, то он создается. В случае неудачи возвращает значение NULL
r+	Открытие файла для обновления. Файл доступен для чтения и записи в рамках существующих границ. Информация, записанная вне границ файла, игнорируется
w+	Файл доступен для записи и чтения. Содержимое существовавшего файла стирается при его открытии. В случае неудачи возвращается значение NULL
a+	Файл доступен для чтения и записи. Указатель при открытии файла устанавливается на его конец. В случае неудачи возвращается значение NULL

Возможны два режима обмена информацией с файлом: бинарный (двоичный) и текстовый. При работе с файлом в бинарном режиме средства буферизации отсутствуют. Информация считывается непосредственно в таком виде, в каком она хранится в файле. Никакие ее дополнительные преобразования не выполняются. В этом режиме работы с файлом информа-

ция побайтно передается в переменные из файла по указанным адресам и таким же образом считывается из них и записывается в файл. Вся информация воспринимаются равнозначной.

При работе с файлом в текстовом режиме содержимое файлов разбивается на строки. В конце каждой строки записывается пара кодов для обозначения конца строки (CR и LF), получаемых преобразованием символа окончания строки '\n', а при чтении данных из файла производится обратное преобразование кодов CR и LF в '\n'.

Существуют дополнительные спецификаторы, позволяющие указать способ обмена информацией с открываемым файлом.

8 Режимы открытия файла

Символ	Режим работы
b	Файл открывается в двоичном режиме
t	Файл открывается в текстовом режиме

П р и м е р 23

Показана рекомендуемая последовательность операторов, которую необходимо использовать при открытии файла.

```
#include <stdio.h>
int main()
{
        FILE *f;
        char str[15];
        scanf("%s", str);
        if ((f = fopen(str, "r+b")) == NULL)
        {
                puts("произошла ошибка");
                return 1;
        }
        …
        return 0;
}
```

После завершения работы с потоком его необходимо закрыть. При этом файловая переменная разрушается и производится корректное завершение работы с файлом. Закрытие файла осуществляется функцией

int fclose(FILE *);

Если необходимо завершить работу с несколькими файлами, можно использовать функцию

int fcloseall(void);

Для осуществления операций ввода-вывода из потока используются функции:

1) для посимвольного ввода-вывода используются функции:

int fgetc(FILE *); – ввод одного символа;

int fputc(int , FILE *); – вывод одного символа;

2) для построчного ввода-вывода используются функции:

char *fgets(char *str, int n, FILE *f); – передает *n* байт из потока, связанного с указателем *f*, в строку по адресу *str*;

int fputs(char *str, FILE *f); – переносит байты из строки *str* в поток с указателем *f*;

3) функции блокового ввода-вывода работают в двоичном режиме открытия файла:

int fread(void *ptr, int size, int n, FILE *f); – считывает *n* элементов размером *size* каждый из открытого файла *f* и помещает их в оперативную память по адресу *ptr*, в случае неудачи возвращает значение EOF;

int fwrite(void *ptr, int size, int n, FILE *f); – записывает *n* элементов размером *size* байт каждый, определяемых адресом *ptr*, в открытый файл с указателем *f*, в случае неудачи возвращает значение EOF;

4) функции форматного ввода-вывода для текстового режима

int fscanf(FILE *f, char *format, ...);

int fprintf(FILE *f, char *format, ...);

Эти функции форматного ввода-вывода аналогичны функциям **scanf** и **printf**.

Содержимое файла, представляющее последовательность байт, часто сравнивают с магнитной лентой. Место на ленте, с которым в данный момент осуществляется работа, определяется значением указателя типа **FILE ***, связанного с файлом. После проведения операции чтения/записи указатель соответствующим образом изменяет значение, настраиваясь на следующую порцию информации. Однако неудобство последовательного чтения при работе с файлом состоит в том, что для нахождения *n*-й записи нужно последовательно считать (*n*–1) предыдущие записи в файле. В библиотеке **stdio.h** описан ряд функций, позволяющих изменить такой режим работы, устанавливая указатель на нужное место в файле:

void rewind(FILE *); – устанавливает указатель потока на его начало;

int feof(FILE *); – проверка окончания файла (1 – если файл закончен);

int fseek(FILE *f, long offset, int n); – перемещает указатель в заданное место файла. "Сдвиг" выполняется на указанное в *offset* число байт. Переменная *n* указывает точку отсчета в файле: 0 – от начала файла; 1 – от текущей позиции; 2 – от конца файла. Если *offset* > 0 – сдвиг выполняется в строку конца файла, если *offset* < 0 – в сторону начала файла. При удачном завершении функция возвращает ноль.

11.6 Алгоритмы для сложных типов данных

Устройство массивов и файлов привело к появлению часто возникающих задач в процессе хранения информации с их помощью. При использовании массивов такой задачей является задача сортировки хранимой информации, а при использовании файлов – задача поиска информации.

При хранении информации в массивах, используется три базовых алгоритма сортировки: метод простых вставок, метод обменов и метод выбора.

Сортировка простыми вставками. Смысл алгоритма состоит в последовательном переносе элементов упорядочиваемого массива в уже упорядоченный массив на положенные им места.

Для осуществления сортировки одномерного массива *A* нужен второй массив *B* такого же размера. Первый элемент массива *A* переносится в массив *B* без условий. Новая последовательность считается упорядоченной. Каждый следующий элемент массива *A* последовательно сравнивается с элементами массива *B* начиная с первого его элемента. В этом процессе устанавливается место для вставки нового элемента в массив *B*. Элементы, расположенные справа от указанной позиции, сдвигаются на одну позицию вправо. Новый элемент из массива *A* вставляется на освободившееся место.

Сортировка обменами (метод "пузырька"). Метод состоит в последовательном сравнении соседних элементов в массиве *A* и замене их местами, если не выполняется требуемое условие сортировки. При выполнении этой операции над всеми элементами массива больший из них (при сортировке по возрастанию) займет последнюю позицию. При многократном выполнении указанных действий соответствующие элементы займут места с номерами *n-1*, *n-2*, *n-3* и так далее, а массив будет упорядочен.

Усовершенствованный метод обмена с изменением направления предполагает найденный элемент, который не удовлетворяет поставленному условию сортировки, сдвигать в направлении противоположном направлению сортировки до тех пор, пока он не удовлетворит условию решения задачи.

Метод выбора. Сортировка выбором требует найти среди рассматриваемых элементов наибольший (наименьший) и осуществить обмен его с последним элементом в массиве. Далее уменьшаем число рассматриваемых элементов в массиве на один, так как последний элемент в массиве уже отсортирован. Повторяя эти действия над наборами элементов массива уменьшающейся длины, мы получаем упорядоченную последовательность.

Для организации поиска информации в файле можно использовать два алгоритма: *последовательного перебора* и *бинарного поиска*.

Для поиска информации с помощью алгоритма бинарного поиска, информация в файле должна быть упорядочена по некоторому свойству. Алгоритм состоит в последовательном сравнении ключевого значения искомой информации и информации, находящейся в файле в центре рассматриваемого диапазона с последовательным исключением из рассмотрения ненужной половины.

12 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

При программировании реальных задач очень часто программисты сталкиваются с проблемой организации гибкого подхода к объемам обрабатываемой информации, которая тесно взаимосвязана с задачей создания эффективных программ. Эта проблема состоит в том, что во многих задачах нельзя заранее предугадать точное количество данных, а следовательно, и количество переменных для их хранения. Малоэффективным здесь оказывается и применение переменных сложных типов данных, так как в момент компиляции в программе четко должны быть указаны размеры всех массивов, структур и объединений. Поэтому при использовании переменных сложных типов данных стараются задать им такие размеры, которых, исходя из здравого смысла, программе хватит всегда. С одной стороны, это может повлечь за собой захват программой больших объемов оперативной памяти ЭВМ, которую программа может и не использовать в данный конкретный момент, а держать "на всякий случай", что негативно скажется на эффективности функционирования всей системы в целом. С другой стороны, нельзя дать гарантию, что в процессе эксплуатации программы не возникнет ситуация, когда программе не хватит выделенной оперативной памяти и она просто станет непригодной для решения задачи.

Выходом в этой ситуации стало создание целого класса *динамических структур* данных, которые обладают всеми свойствами типов данных, но исходя из сложности их воплощения отдельно, они реализуются на базе уже существующих типов данных. Главной их отличительной особенностью является возможность быстрого и гибкого видоизменения в процес-

се работы программы. Наиболее удобной базой для их построения стали структуры. Оказалось, что возможность использования в составе структур указателей того же типа, что и сам структурный тип, связана с целым рядом новых свойств. На их основе строятся динамические структуры данных: *списки*, *буферы*, *стеки*, *деревья*.

12.1 Однонаправленный линейный список

Рассмотрим, какой эффект можно получить от включения в состав структуры указателя на саму структуру:

```
struct lSp
{
        struct lSP *next;
        int key;
};
```

Определим в программе два указателя вышеописанного типа:

```
struct lSp *first, *other;
```

Создадим в основной памяти ЭВМ, не занятой нашей программой, ячейку памяти типа *lSp* и запомним ее адрес в указателе *first*, а в поле *next* этой ячейки запишем **NULL**:

```
first = (struct lSp *) malloc(sizeof(struct lSp));
first->next = NULL;
```

Таким образом, появилась новая ячейка памяти, которая не занимает область сегмента данных программы, а вынесена в свободную память компьютера, называемую *кучей*.

Далее мы можем создать неограниченное количество подобных ячеек, каждая из которых будет хранить свою информацию. Для того, чтобы программа не потеряла связь с этими ячейками, необходимо объединить их в цепочку. В этом случае наша программа может обойтись всего двумя указателями для работы с созданным списком: указатель *first* будет помнить адрес первой ячейки в списке, а указатель *other*, переходя от ячейки к ячейке, может быть настроен на любую из них. Окончание списка показывает ячейка, у которой в поле *next* находится значение NULL. Созданный таким образом список носит название *однонаправленного линейного списка*. Процесс его создания приведен в примере 24, а устройство показано ни рис. 10.

П р и м е р 24

```
other = first;
while (other->next != NULL) other = other->next;
other->next = (struct lSp *) malloc(sizeof(struct lSp));
other = other->next;
other->next = NULL;
scanf("%d", &other->key);
```

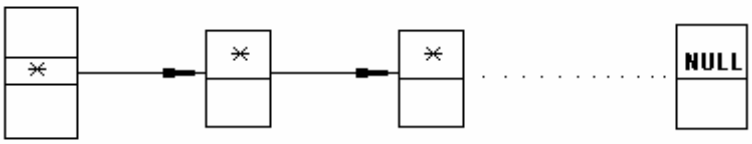


Рис. 10 Устройство однонаправленного списка

Наиболее распространенными операциями, применяемыми к элементам динамических структур данных, являются поиск элемента по некоторому ключевому полю, добавление нового элемента в список и удаление элемента по ключевому значению. Рассмотрим эти действия применительно к однонаправленному линейному списку.

Поиск элемента осуществляется последовательным перебором элементов списка, начиная с первого, до тех пор, пока нужный элемент не будет обнаружен или не закончатся элементы в списке.

П р и м е р 25

```
int key;
scanf("%d", &key);
other = first;
while (other != NULL && key != other->key) other = other->next;
```

Добавление элемента в уже созданный список. Для осуществления этой операции необходимо установить в списке место расположения элемента, после которого планируется осуществить вставку нового элемента. После этого создается новая ячейка памяти (отдельно от списка). В поле *next* данной ячейки заносим содержимое поля *next* ячейки из списка, после которой будет осуществлена вставка, а сама ячейка из списка должна запомнить адрес новой ячейки. После осуществления этих действий новая ячейка окажется встроенной внутрь списка.

П р и м е р 26

```
int key;
struct lSP *new;
printf("Задайте ключ для вставки…");
scanf("%d", &key);
other = first;
while (other->next != NULL && other->key != key)
        other = other->next;
new = (struct lSp *) malloc(sizeof(struct lSp));
new->next = other->next;
other->next = new;
```

Если новая ячейка должна стать в списке первой, то последовательность действий будет следующей.

П р и м е р 27

```
new = (struct lSp *) malloc(sizeof(struct lSp));
new->next = first;
first = new;
```

Удаление из списка заданного элемента. Так же как и при добавлении, процедуры удаления из списка первого элемента и любого другого будут выполняться по-разному. При удалении первого элемента необходимо соответствующим образом изменить значение указателя на первый элемент списка, после чего первый элемент можно удалить.

П р и м е р 28

```
scanf("%d", &key);
if (key == first->key)
{
        other = first;
        first = first->next;
        free(other);
}
```

При удалении любого другого элемента списка необходимо найти элемент, расположенный в списке перед ним. Далее необходимо обойти указателем текущего элемента удаляемый элемент, перенаправив его на следующий элемент после удаляемого. Таким образом будет сохранена целостность списка. После этого найденный элемент можно удалить.

П р и м е р 29

```
scanf("%d", &key);
other = first;
while (other->next->key != key && other->next != NULL)
        other = other->next;
if (other->next->key == key)
{
        new = other->next;
        other->next = other->next->next;
        free(new);
}
```

После окончания работы со списком необходимо освободить память, которую он занимал.

П р и м е р 30

```
do
{
        other = first->next;
        free(first);
}while ((first = other) != NULL);
```

12.2 Двунаправленный список

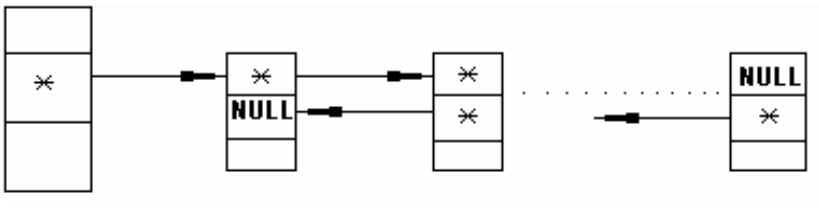
Следует заметить, что использование однонаправленных списков не всегда удобно, так как мы можем перемещаться по списку только в одну сторону. Между тем мы можем в процессе решения задачи столкнуться с необходимостью перемещаться по цепочке в обратном направлении. Для достижения этой цели введем в состав описания динамической структуры еще один указатель:

```
struct lSp1
{
```

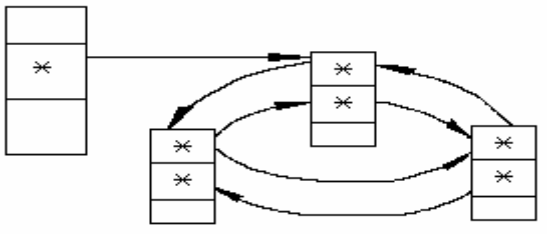
```
        struct lSP1 *next, *last;
        int key;
};
```

В этом случае можно связать указатель *next* со следующим элементом списка, а указатель *last* с предыдущим.

При создании списка возможны два варианта организации цепочки. Можно при создании первого элемента обоим указателям присвоить константу NULL. При этом получится *линейная структура списка* (рис. 11, *а*). Если же обоим указателям присвоить адрес созданного элемента, то получится *кольцевая структура списка* (рис. 11, *б*).



а)



б)

Рис. 11 Устройство двунаправленного списка:
а – линейная структура; *б* – кольцевая структура

П р и м е р 31

```
struct lSp1 *first;
first = (struct lSp1 *) malloc(sizeof(struct lSp1));
// При создании линейного списка
first->next = first->last = NULL;
// При создании кольцевого списка
first->next = first->last = first;
```

Поиск элемента в двунаправленном списке осуществляется аналогично поиску в однонаправленном списке.

Добавление и удаление элементов осуществляются проще, чем подобные операции для однонаправленного списка. Это стало возможным благодаря наличию связей у любого элемента не только со следующим, но и с предыдущим элементом списка.

Рассмотрим добавление элемента в двунаправленный список.

П р и м е р 32

```
struct lSp1 *other, *new;
scanf("%d", &key);
other = first;
while (other->key != key) other = other->next;
new = (struct lSp1 *) malloc(sizeof(struct lSp1));
new->next = other->next;
new->last = other;
other->next->last = new;
other->next = new;
```

Таким образом новый элемент встроен в структуру списка.

При удалении элемента необходимо его соседям слева и справа присвоить ссылки друг на друга. Рассмотрим процедуру удаления заданного элемента списка подробнее.

П р и м е р 33

```
other = first;
while (other->key != key) other = other->next;
other->next->last = other->last;
```

```
other->last->next = other->next;
free(other);
```

Очистка памяти при окончании работы программы осуществляется аналогично удалению однонаправленного линейного списка из памяти ЭВМ.

12.3 Буфер

Динамическая структура типа *буфер* (очередь) является распространенным динамическим типом данных, моделирующим поведение реальных очередей в различных процессах. Иногда их называют структурами типа FIFO (от англ. "first in – first out", т.е. первый пришел – первый ушел). Базой ее построения может выступать как однонаправленный, так и двунаправленный список. Главным ограничением при работе с буфером является то, что добавление нового элемента всегда осуществляется в конец списка, а удаляется всегда первый элемент в списке.

12.4 Стек

Другой распространенной динамической структурой данных является *стек*. Иначе он называется структурой типа LIFO (от англ. "last in – first out", т.е. последний пришел – первый ушел). Стеки часто используются для организации правильного функционирования операционных систем и системных процессов. Базой его построения также может выступать однонаправленный или двунаправленный список. Главным ограничением при работе со стеком является то, что добавление нового элемента всегда осуществляется в конец списка. Удаляется всегда также последний элемент в списке.

12.5 Бинарное дерево

Бинарное дерево является одной из немногих динамических структур, внутренняя организация которых определяется хранимой информацией. Внутренняя организация бинарного дерева решает задачу удобного хранения в оперативной памяти ЭВМ больших объемов информации с учетом возможности быстрого доступа к требуемым данным. При этом данные хранятся упорядоченно по ключевому полю.

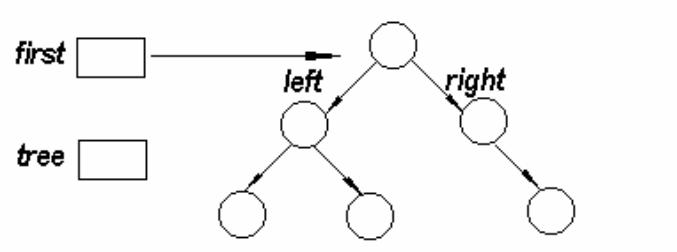


Рис. 12 Устройство бинарного дерева

Организовать дерево можно на основе структурного типа, подобного двунаправленным спискам:

```
struct Tree
{
        struct Tree *left, *right;
        int key;
};
```

Каждый элемент дерева имеет в своем составе два указателя. Однако эти указатели связывают его не с предыдущим и последующим элементами списка, а определяют две *ветви* с элементами следующего уровня (см. рис. 12). При этом в левой ветви хранятся все данные с ключевым полем меньшим, чем у текущего элемента, а в правой – все данные с ключевым полем большим, чем у текущего элемента. Элемент первого уровня называется *корнем дерева*. Порядок расположения элементов определяется при вставке новых данных в структуру дерева.

Вставка элементов в бинарное дерево. Если дерево пустое (указатель на первый элемент равен NULL), то необходимо создать корневой элемент дерева. Оба его указателя в момент создания не связаны с ветвями, поэтому оба указателя должны быть пустыми (NULL). В противном случае необходимо обойти все элементы дерева, начиная с корневого элемента до тех пор, пока не будет обнаружено место для вставки новых данных. Если у текущего элемента ключевое поле меньше, чем у вставляемых данных, то нужно продолжить обход дерева в направлении правой ветви от текущего элемента. В противном случае обход продолжается в направлении левой ветви. Необходимо продолжать обход до тех пор, пока не будет обнаружен элемент, у которого указатель на ветвь в направлении перехода содержит константу NULL. Это и будет место для вставки нового элемента в структуру дерева (создания новой его ветви).

П р и м е р 34

Последовательность действий, необходимая для вставки нового элемента в бинарное дерево.

```
struct Tree *first = NULL, *tree;
int key;
…
```

```
scanf("%d", &key);
if (first == NULL)
        first = (struct Tree *) malloc(sizeof(struct Tree)),
        first->left = first->right = NULL,
        first->key = key;
else
{
        int i = 1;
        tree = first;
        while (i)
        {
                if (key < tree->key)
                {
                        if (tree->left != NULL)
                                tree = tree->left;
                        else
                                tree->left = (struct Tree *)
                                malloc(sizeof(struct Tree)),
                                tree = tree->left,
                                i = 0;
                }
                else
                {
                        if (tree->right != NULL)
                                tree = tree->right;
                        else
                                tree->right = (struct Tree *) malloc(sizeof(struct Tree)),
                                tree = tree->right,
                                i = 0;
                }
        }
        tree->left = tree->right = NULL;
        tree->key = key;
}
```

Удаление элементов бинарного дерева. Эта процедура сопряжена с некоторыми трудностями, вызванными необходимостью обеспечения целостности дерева в процессе проведения операции.

Если удаляемый элемент является конечным элементом дерева (не имеет ни левой, ни правой ветвей), то он удаляется обычным образом. При этом соответствующему указателю элемента, в ветви которого находится удаляемый элемент, нужно присвоить NULL, так как эта ветвь удаляется.

Если удаляемая вершина имеет одну ветвь продолжения, то необходимо передать указателю верхнего элемента адрес этой ветви, после чего сам элемент можно удалить.

При удалении вершины, имеющей две ветви, главную трудность составляет то, что на удаляемую вершину указывает один элемент. Поступают в этом случае следующим образом: на место удаляемой вершины перемещают подходящий элемент, который сохранит принцип организации дерева в целом. Таким звеном является либо самый правый элемент в левой ветви, либо самый левый элемент в правой ветви относительно удаляемого элемента. Его содержимое переписывают в ячейку с удаляемой информацией, а сам элемент после этого удаляют.

Поиск информации в бинарном дереве осуществляется аналогично процедуре добавления нового элемента. Отличие состоит в том, что нужно найти элемент, у которого ключевое поле совпадает со значением ключа поиска.

13 РАБОТА С ФУНКЦИЯМИ В ЯЗЫКЕ С

13.1 Описание и порядок исполнения функций

Понятие функции является для языка программирования С ключевым, так как в основе языка лежат принципы модульного программирования. *Функция* представляет собой обособленный фрагмент текста программы, предназначенный для выполнения определенных действий и, часто, возвращающий некоторый результат.

Каждая программа на языке С должна содержать единственную главную функцию с именем **main**, которая обеспечивает начало работы откомпилированной программы. Помимо нее программа может содержать неограниченное число других функций, вызов на исполнение которых осуществляется функцией **main**. Каждая функция в программе должна иметь уникальное имя, по которому осуществляется обращение к ней. Всем именам функций по умолчанию присваивается класс памяти **extern**, т.е. функции имеют внешний тип компоновки и статическую продолжительность существования. Для доступности в модуле функция должна быть определена в нем до первого вызова.

В определении функции указываются: последовательность выполняемых действий, имя функции, тип возвращаемого значения и описания формальных параметров с указанием соответствующих типов данных. Определение функции разбивают на две части – *заголовок* и *тело функции*:

тип имя_функции(список формальных параметров)
тело_функции

В качестве *типа функции* может выступать любой тип данных, в том числе и **void**, если функция не возвращает результат явно. *Список формальных параметров* может быть пустым, вместо него может стоять **void**, либо здесь перечисляются формальные параметры с описанием их типов данных. Список параметров переменной длины должен заканчиваться многоточием. *Тело функции* представляет собой ограниченный фигурными скобками блок, идущий сразу за заголовком функции. Внутри блока размещаются определения необходимых функции программных объектов и операторы. Последним (иногда необязательным) оператором является оператор возврата из функции **return** (return; или return *выражение*;). Возвращаемое выражение должно иметь тот же тип данных, что и тип функции, либо тип, допускающий автоматическое преобразование к типу возвращаемого функцией результата.

Для того, чтобы компилятор смог проверить правильность обращения к функции в программе (правильное указание ее имени и соответствие типов передаваемых данных и результата), он должен иметь представление о функции. Поэтому до первого обращения к функции должно быть размещено или определение функции, или ее описание (*прототип функции*). Фактически прототип является заголовком функции с той разницей, что в нем могут отсутствовать имена передаваемых объектов в списке формальных параметров:

тип имя_функции(спецификация параметров);

Для обращения к функции используется выражение с использованием операции "круглые скобки":

имя_функции(список фактических параметров)

В качестве имени функции может выступать указатель на нее. Список *фактических параметров функции* (аргументов) представляет собой список выражений, число и тип данных которых должны совпадать с формальными параметрами функции. Соответствие между формальными и фактическими параметрами устанавливается по их взаимному расположению в списках. Если типы данных формальных и фактических параметров не совпадают, то компилятор пытается автоматически преобразовать тип фактических параметров, если это допустимо.

Большое внимание следует уделять правилам передачи параметров при обращениях к функциям. В языке С предусмотрена только передача параметров по значениям. Поэтому формальные параметры функции локализованы в ней и не доступны вне определения функции. Никакие операции над параметрами функции, а также ее внутренними переменными не влияют на фактические параметры. Передача параметров по значению предполагает выполнение следующих действий:

1) при подготовке функции к выполнению выделяются участки памяти для формальных параметров. После этого формальные параметры становятся внутренними объектами функции;
2) вычисляются значения выражений, использованных в качестве фактических параметров при вызове функций;
3) значения выражений – фактических параметров копируются в участки памяти, выделенные для формальных параметров;
4) преобразование данных в теле функции выполняется с использованием внутренних объектов, а в точку вызова функции передается только возвращаемое значение;
5) после выхода из функции память, выделенная для формальных параметров, освобождается.

Вызов функции всегда является выражением, но его использование в программе зависит от типа возвращаемого значения. Если в качестве типа возвращаемого значения указан тип **void**, то функция не возвращает ничего. Следовательно, ее нельзя использовать в выражениях, а нужно вызывать в виде отдельного оператора:

имя_функции(список фактических параметров);

Помимо этого может отсутствовать и список фактических параметров, если вместо списка формальных параметров функции использовано ключевое слово **void** в ее заголовке.

В случае использования только параметров-значений функция не имеет никакой возможности изменять объекты вызывающей функции при выполнении своих операторов. Однако подобная необходимость появляется достаточно часто (примером может служить функция scanf). Для этого существует косвенная возможность – использование в качестве формальных параметров функции указателей. При помощи указателя в вызываемую функцию можно передать адрес фактического объекта, принадлежащего вызывающей функции. Применяя операцию разыменования '*' можно получить доступ к любому объекту программы.

П р и м е р 35

Определение модуля числа

```
#include <stdio.h>
void modul(int *i)
{*i = *i > 0 ? *i : -*i;}
void main()
{
        int n;
        scanf ("%d", &n); printf("%d", modul(&n));
}
```

13.2 Массивы в качестве параметров функции

Если в качестве параметра функции выступает массив, то внутрь функции передается только адрес его начала. Поэтому абсолютно равноправными будут прототипы функций

double sum(int n, double a[]);

и

double sum(int n, double *a);

Так как массив всегда передается в функцию как указатель, то внутри функции производится изменение элементов массива – фактического параметра, который указан как параметр при вызове функции. Имя массива внутри вызываемой функции уже не воспринимается как константный указатель, а обладает всеми свойствами указателя.

13.3 Указатели на функции

Рассмотрим выражение "вызов функции" подробнее:

обозначение_функции(список фактических параметров);

здесь *обозначение функции* воспринимается как "указатель на функцию, возвращающую значение указанного типа". В языке С указателем на функцию может выступать выражение или переменная, представляющие адрес функции. Самым простым указателем на функцию является ее имя, которое используется в определении функции и при ее описании. В этом случае имя функции воспринимается как константный указатель, который жестко связан с функцией и не может быть перенастроен на другой программный объект.

Указатель на функцию как переменная должен быть определен отдельно от определения или прототипа какой-либо функции:

тип (*имя_указателя)(спецификация параметров);

здесь *тип* – тип возвращаемого функцией значения; *имя указателя* – идентификатор; *спецификация параметров* – определение состава и типов параметров функции.

Единственным требованием является полное соответствие определения указателя-переменной и прототипа функции, адрес которой ему присваивается.

К указателям на функцию можно применять только операцию присваивания.

Переменная-указатель на функцию, связанная с конкретной функцией, может использоваться для вызова этой функции. Возможны два варианта построения выражения:

1) (*указатель)(список фактических параметров);
2) указатель(список фактических параметров).

П р и м е р 36

```
#include <stdio.h>
#include <math.h>
int main()
{
        unsigned i;
        double x;
        double (* fn)(double);
        puts("Что считаем?");
        puts("1. Синус");
        puts("2. Косинус");
        puts("3. Логарифм");
        scanf("%d%lf", &i, &x);
        switch (i)
        {
                case 1: fn = sin; break;
                case 2: fn = cos; break;
                case 3: fn = log;
        }
        printf("%g", fn(x));
        return 0;
}
```

Указатель на функцию может являться параметром любой функции. В этом случае переменная-указатель на функцию включается в список формальных параметров вызываемой функции. Рассмотрим пример функции для вычисления определенного интеграла методом трапеций.

П р и м е р 37

```
double integral(double (*f)(double), double a, double b, double h)
{
        double S = (f(a) + f(b)) * h / 2;
        for (a += h; a < b; a += h) S += f(a) * h;
        return S;
}
```

Указатель на функцию как возвращаемое функцией значение. Возможна такая организация программы, когда выполняемые действия определяются не в точке исполнения, а в некоторой "промежуточной" функции. В этом случае промежуточная функция должна вернуть в вызвавшую ее функцию в качестве результата адрес той функции, которая должна быть выполнена. Такой адрес можно возвратить в виде значения указателя на функцию. Прототип такой функции выглядит следующим образом:

тип (*имя(формальные_параметры1))(формальные_параметры2);

В вызывающей функции необходимо предусмотреть указатель, способный принять такой адрес.

П р и м е р 38

```
double (*variant(double))(int i)
{
        switch (i)
        {
                case 1: return sin;
                case 2: return cos;
                default: return NULL;
        }
}
int main()
{
        double (* p)(double);
        p = variant(1);
        if (p != NULL) printf("%g", p(3.14));
        return 0;
}
```

13.4 Функции с переменным числом параметров

В языке С существует возможность создавать функции, у которых во время компиляции список формальных параметров не определен и даже неизвестны типы данных, к которым могут принадлежать передаваемые параметры. Типы параметров и их количество становится известным в процессе выполнения программы. При определении и описании таких функций список формальных параметров должен заканчиваться запятой и многоточием:

тип имя(список_явных_параметров, ...);

здесь *список явных параметров* – список параметров, тип и количество которых известны на момент компиляции.

Многоточие означает для компилятора, что контроль количества и типов параметров при обращении к функции проводить не надо. Главная сложность при работе со списком параметров переменной длины состоит в том, что он не имеет имени, поэтому неизвестно его начало в оперативной памяти. Для определения начала списка параметров переменной длины необходимо, чтобы функция имела хотя бы один явный параметр.

Для нормальной работы со списком параметров переменной длины, функции прямо или косвенно необходимо определиться с длиной списка (количеством элементов в нем в данный момент). Существуют две идеи, объясняющие как выйти из сложившейся ситуации. Можно напрямую передавать функции число параметров в списке. Другим способом является помещение в конец списка признака его окончания. В этом случае функция должна проверять наличие в списке кода окончания списка. Выборка элементов из списка осуществляется с помощью указателей. Рассмотрим в качестве примера функцию для подсчета суммы целых чисел.

П р и м е р 39

Явная передача в функцию числа параметров

```
long sum(int n, ...)
{
        int *p = &n;
        long s = 0;
        for ( ; n; n--) s += *(++p);
        return s;
```

```
}
```

Главным недостатком функции является ее неправильная работа при ошибке в задании числа аргументов.

П р и м е р 40

Нулевой код при окончании списка аргументов

```
long sum(int n, ...)
{
        int *p = &n;
        long s = 0;
        for (; *p; p++) s += *p;
        return s;
}
```

Недостатком в данном случае является невозможность использования кода окончания списка внутри него как аргумента.

Для повышения мобильности, простоты и надежности функций, работающих со списками параметров переменной длины, могут быть использованы макросредства, описанные в заголовочном файле **stdarg.h**. В этом заголовочном файле описаны макросы:

void va_start(va_list param, *последний явный параметр функции*);

тип va_arg(va_list param, *тип*);

void va_end(va_list param);

здесь **va_list** – специальный тип данных, описанный в файле **stdarg.h**; *тип* – тип очередного параметра из списка переменной длины, к которому осуществляется доступ.

Макрос **va_start** используется для настройки указателя типа **va_list** на первый элемент списка параметров переменной длины. Вторым параметром в нем указывается последний из явно заданных параметров функции. Применяя операцию разыменования, можно получить доступ к элементам списка переменной длины. Для этого необходимо знать тип текущего элемента в списке. Применяя макрос **va_arg**, получаем значение текущего элемента списка переменной длины и переадресовываем указатель на следующий элемент списка. Макрос **va_end** служит для корректного завершения работы функции со списком параметров переменной длины. Он вызывается перед возвращением из функции после того, как будет обработан весь список параметров переменной длины. Описатели типов аргументов списка переменной длины лучше передавать в функцию в виде строки символов.

П р и м е р 41

```
double sum(char *format, ...)
{
        va_list p;
        double s = 0;
        va_start(p, format);
        for (; *format; format++)
                switch (*format)
                {
                        case 'i': s += va_arg(p, int); break;
                        case 'd': s += va_arg(p, double);
                }
        va_end(p);
        return s;
}
```

13.5 Рекурсивные функции

Рекурсивным считается объект, в определении которого содержится ссылка на самого себя. Таким образом, функция, содержащая вызов самой себя, будет считаться рекурсивной. Различают *прямую* и *косвенную* рекурсию. Если функция содержит явную ссылку на саму себя, то она называется явно рекурсивной. Если функция ссылается на функцию, которая вызывает исходную, то такая рекурсия называется косвенной. Рекурсивная функция подобна явному циклическому итерационному процессу и всегда может быть заменена им. Она появляется тогда, когда некоторый объект может быть определен через другой объект той же сущности, но с другим содержанием: P = P1(S, P). Важным требованием, относящимся к использованию рекурсии, является конечность объектов, порожденных рекурсивным определением. Классическим примером рекурсивного процесса является вычисление факториала: $n! = n*(n-1)*(n-2)*...*1$.

Рекурсивные алгоритмы обычно более просты и наглядны, чем циклические. Методика анализа любой задачи на предмет составления рекурсивного алгоритма ее решения состоит из следующих этапов:

1) *параметризация задачи*, т.е. выявление различных элементов, от которых зависит ее решение, с целью нахождения управляющего параметра; при этом размерность управляющего параметра должна убывать после каждого рекурсивного вызова;

2) *поиск тривиального случая и его решения*. На этом этапе должно быть найдено решение задачи не содержащее рекурсивного определения;

3) *декомпозиция общего случая*, требующая привести его к одной или нескольким более простым задачам меньшей размерности.

В случае с факториалом управляющим параметром является текущее значение числа *n*. Тривиальный случай представляет собой 0! = 1, а декомпозиция общего случая выглядит $n! = n*(n-1)!$.

Обычно функции имеют локальные объекты, определенные в функции и недоступные за ее пределами. При каждом рекурсивном вызове функции порождается новое множество локальных переменных. Они имеют те же имена, что и переменные функции "предыдущего поколения", однако имеют собственные значения. Все конфликты по именам между переменными разрешены следующим правилом: идентификаторы всегда относятся к самому последнему порожденному множеству переменных. Поэтому нужно убедится, что максимальная глубина рекурсии достаточно мала, так как при каждом новом рекурсивном вызове функции значения всех ее локальных переменных и "состояние вычислений" сохраняется в стеке программы. При достаточно большой вложенности рекурсии стек может быть быстро исчерпан, а программа будет аварийно завершена.

Таким образом, рекурсивные алгоритмы подходят там, где рекурсивно определяются обрабатываемые данные, а там, где есть очевидное итерационное решение, рекурсии следует избегать.

13.6 Параметры функции main

С точки зрения стандарта языка С функция **main** является обычной функцией, которой можно передать параметры. Однако то, что эта функция запускается из операционной системы накладывает определенный отпечаток на список ее формальных параметров, он не может быть произвольным. Прототип функции **main** выглядит следующим образом:

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

здесь **argv** – массив указателей на строки; **argc** – число элементов в массиве **argv**; **envp** – массив указателей на символьные строки, содержащие описание переменных окружения.

Данный список формальных параметров обеспечивает связь функции **main** с командной строкой, из которой можно передавать данные программе. Если функция **main** не обрабатывает параметры из командной строки, то их описание опускается.

Параметры в командной строке разделяются пробелами. Каждый такой параметр заносится в строку, соответствующую argv[i]
(0 < i < argc). Элементу argv[0] соответствует название программы, обязательно указываемое в командной строке.

14 ДИРЕКТИВЫ ПРЕПРОЦЕССОРА В ЯЗЫКЕ С

Обычно, вместе с компилятором языка С в пакет программ, обеспечивающих программирование на языке С, входит еще одна программа – *препроцессор*. Ее главным назначением является предшествующая компиляции обработка исходного текста разрабатываемой программы с целью внесения в него определенных изменений. В состав действий, выполняемых препроцессором, входит выполнение особых команд в тексте программы – *директив препроцессора*. Каждая директива препроцессора должна занимать одну строку в тексте программы и начинаться с символа '#', что позволяет отличать их от операторов программы. Сами директивы препроцессора приведены ниже.

1 Включение текстов из файлов.

#inclule <имя_файла>

или

#inclule "имя_файла"

После обработки текста программы препроцессором на месте этой директивы будет расположен текст указанного файла. Если *имя файла* выделено угловыми скобками (< >), то препроцессор будет искать файл в системных директориях. Если же *имя файла* выделено кавычками, то файл должен находиться в одной директории с обрабатываемым препроцессором файлом.

2 Выполнение замен в тексте программы:

#define идентификатор строка_замещения

После включения данной директивы в текст программы, препроцессор произведет замену всех встретившихся идентификаторов, включенных в состав директивы **#define** на текст *строки замещения*. Чтобы прекратить эти действия, в тексте программы нужно поместить директиву

#undef идентификатор

Если в составе директивы **#undef** указан идентификатор, то препроцессор прекратит осуществление замен именно этого идентификатора. Если директиву **#undef** указать вообще без идентификатора, то будут прекращены замены в соответствии со всеми ранее размещенными в тексте программы директивами **#define**.

3 Директивы условной компиляции. Препроцессорная обработка позволяет манипулировать текстом исходной программы таким образом, что при разных условиях будут компилироваться разные ее части. Делается это при помощи директив

#if константное_выражение
#ifdef препроцессорный_идентификатор
#ifndef препроцессорный_идентификатор
#elif выражение
#else
#endif

Директивы **#if**, **#ifdef** и **#ifndef** связывают выполнение компиляции блока операторов, следующих за директивой, с заданным условием. Окончание блока для условной компиляции обозначает директива **#endif**. Директива **#elif** используется вместе с директивой **#if** для формирования неограниченного количества альтернативных блоков. Директива **#else** используется вместе с директивами **#if** и **#elif** для создания единственного альтернативного варианта компиляции.

В директиве **#ifdef** проверяется, определен ли с помощью директивы **#define** к текущему моменту идентификатор, помещенный после **#ifdef**. А в директиве **#ifndef** проверяется обратное директиве **#ifdef** условие. Здесь истинной будет считаться ситуация, когда проверяемый идентификатор не был определен ранее.

П р и м е р 42

```
#define DEBUG 5
int a =
#if DEBUG == 5
DEBUG;
#elif DEBUG == 1
DEBUG * 2;
#else
1;
#endif
#ifdef DEBUG
printf("%d", a);
#endif
#undef DEBUG
#ifndef DEBUG
printf("Отладка программы закончилась");
#endif
```

В приведенном примере с помощью директив условной компиляции формируется инициализирующее выражения при определении переменной *a* и сообщение для вывода на печать.

4 Пустая директива:

#

5 Директива нумерации строк:

#line константа

Следующей за директивой строке в тексте программы будет присвоен указанный в директиве номер.

6 Выдача сообщений об ошибке:

#error сообщение

7 Задание настроек компилятора:

#pragma список_лексем

В директивах препроцессора разрешено использовать три препроцессорные операции: #, ## и **defined**.

Операции # и ## употребляются вместе с директивой **#define** для упрощения работы с лексемами в строке замещения. Унарная операция # требует, чтобы идущая следом лексема была выделена в кавычки. Бинарная операция ## обеспечивает конкатенацию лексем в стоке замещения.

Унарная операция

defined аргумент

применяется вместе с директивами условной компиляции **#if** и **#elif** для упрощения записи сложных условий выбора. Она проверяет, был ли определен с помощью директивы **#define** указанный аргумент. В случае положительного результата про-

верки условия возвращается величина 1L. В противном случае – возвращается 0L. Операцию запрещено применять в составе директив **#define** и **#undef**.

15 ПОСТРОЕНИЕ МАКРОСОВ В ЯЗЫКЕ С

Макрос представляет собой замену одной последовательности символов другой. Для осуществления замены необходимо предусмотреть соответствующее макроопределение. Простейший способ осуществления замен в тексте программ предоставляет директива препроцессора **#define**. Однако рассмотренные выше возможности этой директивы ограничены из-за фиксированной строки замещения. Большими возможностями обладает следующее построение строки замещения:

#define имя_макроса(список_параметров) строка_замещения

здесь *список параметров* – список идентификаторов, разделенных запятыми.

При определении макроса недопустимы пробелы между именем макроса и списком параметров.

Классическим примером макроса может служить определение в виде макроса операции определения абсолютного значения числа. Если в начале программы определить макрос

```
#define ABS(a) (a > 0 ? a : -(a))
```

а в тексте программы написать

```
ABS(x)
```

то после обработки текста программы препроцессором, на этом месте в программе появится выражение

```
(x > 0 ? x : -(x))
```

В отличие от функций, определение которых существует в программе в одном экземпляре, макросы вставляются препроцессором в текст программы столько раз, сколько они используются. Функция определяется для аргументов указанных в ее заголовке типов и возвращает результат строго определенного типа. Макрос может использоваться для данных любого типа, допустимого в формируемом препроцессором выражении. Тип возвращаемого макросом результата определяется типом используемых в нем операндов. Фактически макрос способен заменить несколько простых функций.

Для устранения неоднозначного толкования формируемых макроподстановок может быть полезным параметры макроса и строку замещения выделять в скобки.

Главным ограничением при записи директивы **#define** с определением макроса является то, что она должна состоять из одной строки в тексте программы. Если выражение, записываемое в определении макроса является длинным, то его можно разбить на несколько строк. Для этого применяется символ '\', который обозначает перенос текста на новую строку. Длину формируемой строки может сократить применение препроцессорных операций # и ##.

16 ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ ПРОГРАММИРОВАНИЯ

Процесс разработки программ для решения различных задач является творческим, а следовательно, трудно формализуемым процессом. Это означает, что отсутствует единая система правил, следуя которой можно быстро создать правильную программу решения поставленной задачи; т.е. создавая программу, разработчик каждый раз начинает свой труд "с нуля". Однако существуют рекомендации по организации труда программиста, которые позволяют максимально сократить время разработки программы при получении наиболее высокого качества создаваемого программного продукта. Эти правила рекомендуют работу над программой разбивать на ряд стадий.

1 *Постановка задачи* – определяются цели и условия решения задачи, раскрывается ее содержание, выявляются факторы, оказывающие влияние на ход вычислений или конечный результат.

2 *Формализация* – по результатам анализа сущности задачи определяется объем и специфика исходных данных, вводится система условных обозначений, устанавливается принадлежность данной задачи к одному из известных классов задач и выбирается соответствующий математический аппарат описания.

3 *Выбор и разработка метода решения* – построение на предыдущем шаге математической модели приводит к необходимости поиска способов решения задачи, т.е. необходимо установить зависимость искомых результатов от исходных данных и разработать такие способы получения результата, которые реализуемы на ЭВМ.

4 *Составление алгоритма*. Часто решение задачи не удается получить в виде явной формулы. В этом случае разработке программы предшествует разработка *алгоритма*. Главное назначение алгоритма – точное и детальное описание процесса обработки исходных данных для получения требуемого результата в соответствии с выбранным методом. На этом этапе последовательность действий, составляющая процесс решения задачи преобразуется в последовательность действий ЭВМ.

5 *Составление программы* – составленный алгоритм записывается на выбранном языке программирования в соответствии с правилами записи программы на этом языке.

Далее разработанная программа с помощью устройств ввода/вы-вода вводится в память ЭВМ. Получается файл с текстом программы. Дальнейшие преобразования текста программы показаны на рис. 13. Программу необходимо скомпилировать. После этого полученный объектный код программы обрабатывает компоновщик, налаживающий связи отдельных

функций между собой и с глобальными объектами. Перед компиляцией текст программы на языке С обрабатывает препроцессор. В результате указанных преобразований текста программы формируется рабочая программа ЭВМ.

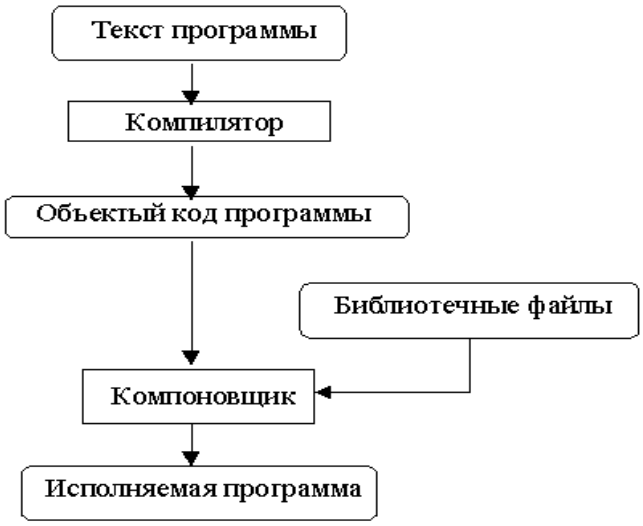


Рис. 13 Необходимые преобразования текста программы

6 *Отладка программы.* На любом из предшествующих этапов могли быть допущены ошибки, которые делают полученный код программы неработоспособным. Выявление и устранение подобных ошибок составляет данный этап проектирования программы.

7 *Тестирование* тесно связано с этапом отладки, однако он предназначен для выявления глубинных ошибок, которые позволяют программе быть работоспособной, однако не позволяют выдавать ей правильный результат. Для выявления подобного несоответствия пользователь готовит систему тестов, с помощью которой проверяется работа программы в различных режимах. Тесты должны содержать различные наборы исходных данных, для которых известны конечные результаты. Тесты нужно подбирать так, чтобы не только установить сам факт наличия ошибки, но и локализовать эту ошибку, то есть по возможности сузить подозреваемую часть программы, содержащую ошибку.

Прошедшая все вышеуказанные стадии задача готова для решения на ЭВМ.

При разработке программы отдельную трудность составляют этапы тестирования и отладки, в ходе которых устраняются присутствующие в программе ошибки. Рассмотрим возможные пути облегчения проводимой работы. Одним из простых путей облегчения этапа отладки программы может стать разделение программы на небольшие модули при ее создании. В этом случае подозрительное место в программе может быть обнаружено простым анализом характера ошибки. Здесь окажется полезной организация искусственных точек останова программы или периодический вывод промежуточных данных.

Целесообразно провести проверку ошибки на устойчивость и повторяемость. Устойчиво повторяющаяся ошибка может быть выявлена в программе с помощью анализа причин ее появления. Если в каком-то случае появилась неповторяющаяся ошибка, то возможны два варианта реакции на ее появление: 1) выполнить контрольный прогон программы и, в случае отсутствия ошибки, забыть о ее появлении; 2) провести проверку аппаратуры.

Ч а с т ь II
ПРАКТИКУМ ПО ЯЗЫКУ С

17 ТВОРЧЕСКИЕ ЗАДАЧИ ПО ПРОГРАММИРОВАНИЮ

Творчество, нахождение новаторских, прогрессивных выходов из создавшейся ситуации всегда было основным условием развития общества. Уровень развития производства и нарастание информационных процессов в условиях, когда специалист не в состоянии, используя традиционные методы, "переварить" такое количество информации, определяет актуальность освоения нового, более творческого подхода к организации информационно-профессиональной деятельности. При этом конкурентоспособный специалист должен обладать способностью к ранжированию информации, интуитивным чутьем на ее актуальность, умением в окружающей действительности уяснить наиболее злободневную проблему и сформулировать профессиональную задачу, определить основные информационные источники.

Интеллектуальный компонент творческой деятельности можно представить в виде комбинации следующих элементов: способности к видению проблемы, легкости и богатстве ассоциирования, гибкости мышления, легкости генерирования идей, критичности и антиконформизме; способности к переносу знаний, умений и навыков из одной сферы знаний в другие, готовности памяти к обработке больших информационных массивов.

Задачи, которые нередко предлагаются в пособиях и учебниках, имеют стандартную, привычную конструкцию, подразумевающую достижение искомого результата по заданной процедуре, и являются лишь слабым подобием реальных жизненных процессов. В процессе профессиональной деятельности специалист, как правило, сталкивается с производственными ситуациями, в которых действуют неопределенные, вероятностные условия, излишние, противоречивые и недостающие данные, когда нужно принимать решения в экстремальных условиях, связанных с ограниченностью временных, материальных и финансовых ресурсов. Производственные ситуации такого рода неизбежно возникают в условиях рыночной экономики, в процессе освоения или разработки новых производственных технологий, современного экономически выгодного и экологически надежного оборудования, ведения предпринимательской и коммерческой деятельности.

"Мышление всегда начинается с проблемы или вопроса, с удивления или недоумения, с противоречия. Этой проблемной ситуацией определяется вовлечение личности в мыслительный процесс" (С.Л. Рубинштейн). Результаты мыслительного анализа проблемной ситуации субъект излагает на каком-то языке (естественном или искусственном), тем самым возникает знаковая модель проблемной ситуации – задача. На следующем этапе деятельности специалист решает сформулированную задачу (или организует решение с помощью других людей), доказывает истинность ее решения, компетентность и качество предпринятых при этом действий и поступков. В случае, когда задача поступает извне в уже готовом сформированном виде, процесс мышления начинается с этапа субъективной трансформации задачи, что проявляется через стремление субъекта переформулировать ее по-своему, создать свою задачу, являющуюся как бы субъективной моделью полученной.

В основе олимпиадной задачи лежит творческая ситуация, требующая разрешения некоторого диалектического противоречия. Остановимся подробнее на изучении механизма творческой работы. Детерминированный процесс решения происходит не сразу, а постепенно, в виде конкретных стадий, в проявлении которых раскрываются новые условия его осуществления.

Процесс инверсионного мышления состоит из шести стадий (А.Э. Эсаулов), которые обучающийся проходит в условиях учебной олиампиадной среды (см. рис. 14).

Рис. 14 Процесс инверсионного мышления

Очень важным этапом инверсионного мышления является формирование замысла задачи. На занятиях по решению творческих инженерных задач обучающимся необходимо самим усматривать в реальном производстве и научных исследованиях проблемные ситуации, требующие творческого подхода к их решению, самим определять цель исследования, основные структурные элементы изучаемого объекта и их взаимосвязи, ограничения, накладываемые внешней средой на возможные решения, самим формулирую задачу и информировать о ней других членов коллектива.

Эффективное нахождение решения поставленной задачи возможно при наличии умения мыслить по ходу решения возникшей задачи, что, с одной стороны, заключается в умении воспроизвести и сохранить имеющуюся систему знаний и действий, которая предписывается этими знаниями, а с другой, – быть способными преобразовать и построить принципиально новую систему, зависящую от постепенно раскрывающихся и преобразующихся вопросов и целей задачи.

Выделяются следующие этапы модели решения олимпиадной задачи:

1) погружение в информационное поле предполагаемой задачи через постановку проблемы, восприятие условий и описание проблемы;

2) разработка информационно-логической модели задачи через установление взаимосвязи между исходными данными, выявление основных законов и границ их применения при решении данной задачи;

3) проверка адекватности разработанной модели условиям постановки задачи;

4) разработка алгоритмической структуры задачи, определение ее оптимальности;

5) разработка технологии реализации алгоритмической структуры задачи, проведение анализа адекватности технологии предложенным средствам реализации;

6) проведение анализа полученных результатов с позиции корректности постановки проблемы, адекватности разработанной информационно-логической модели постановке проблемы, оптимальности алгоритмической структуры и эффективности технологии реализации.

Можно выделить основные факторы, препятствующие успешному нахождению решения задачи:

а) на этапе погружения в информационное поле некоторые значащие элементы информации остаются не востребованными, недостаток или избыток данных вызывает психологический дискомфорт;

б) на этапе разработки информационно-логической модели взаимосвязь между основными структурными элементами устанавливается без учета основных закономерностей протекания процесса, что не позволяет говорить об адекватности модели поставленной проблеме;

в) практически всегда отсутствует проверка промежуточных этапов решения и конечного результата на адекватность, что является недопустимым для специалиста, претендующего на конкурентоспособность.

17.1 Задания повышенного уровня сложности

При выполнении заданий необходимо разработать алгоритм и составить программу решения творческих задач, предлагавшихся на Всероссийских и региональных олимпиадах по информатике. Проанализировать полученные результаты и выявить недостатки в своих знаниях и методах творческой работы.

Задача 1 (6 баллов)

Магическим квадратом называется матрица размера **n**×**n**, состоящая из целых чисел от 1 до $\mathbf{n^2}$, в которой суммы элементов по строкам, столбцам и главным диагоналям равны одному и тому же числу. Порядок матрицы лежит в диапазоне [3, 20].

Построить магический квадрат заданной размерности.

Входной файл содержит целое число, соответствующее порядку матрицы.

Выходной файл содержит найденную матрицу, элементы которой расположены по строкам и столбцам и разделены пробелами.

П р и м е р

Входной файл: INPUT.TXT
3
Выходной файл: OUTPUT.TXT
8 1 6
3 5 7
4 9 2

Предельное время вычисления – 1 минута.

Задача 2 (4 балла)

Построить кривые Серпинского различных порядков (рис. 15, *а* – первого, *б* – второго порядка)

Входной файл содержит целое число **n**, где n – порядок кривой. Результат представляется на экране в пошаговом режиме:

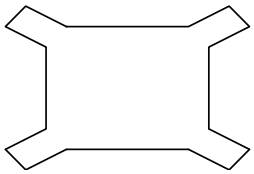
1 экран – кривая первого порядка,

2 экран – кривая второго порядка…

n экран – кривая n-го порядка.

Смена экранов производится по однократному нажатию любой клавиши. Кривые масштабировать так, чтобы кривая n-го порядка целиком помещалась на экране. $n_{max} = 7$.

а)



б)

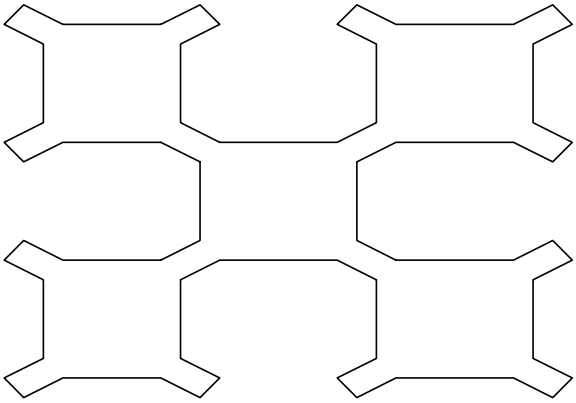


Рис. 15

Задача 3 (5 баллов)

Изобразить одностороннюю поверхность типа ленты Мебиуса.

Задача 4 (4 балла)

Составить программу, разгадывающую японские кроссворды (рис. 16). Входной файл должен содержать:
1-ая строка – количество строк
2-ая строка – количество столбцов
3 и последующие строки – количество подряд идущих единиц в каждой строке и каждом столбце.

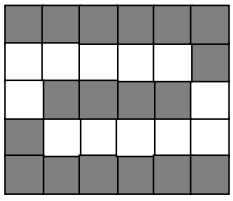


Рис. 16

Выходной файл: элементы матрицы, соответствующие решенному кроссворду, расположены по строкам и столбцам, отделяя элементы пробелами.

П р и м е р

Входной файл: INPUT.TXT

```
5
6
…
6
1
4
1
6
…
1 2
1 1 1
1 1 1
1 1 1
1 1 1
2 1
```

Выходной файл: OUTPUT.TXT

```
1 1 1 1 1 1
0 0 0 0 0 1
0 1 1 1 1 0
```

1 0 0 0 0 0
1 1 1 1 1 1

Задача 5 (6 баллов)

Машиной с неограниченным числом регистров (МНР) называется гипотетическое устройство, состоящее из бесконечного числа регистров (ячеек), в которые можно записывать целые числа. МНР – алгоритмы записываются с помощью следующих четырех команд:

Z(n) – *запись в n-й регистр*;

S(n) – *увеличение содержимого n-го регистра на 1*;

T(m,n) – *копирование m-го регистра в n-ый регистр*;

J(m,n,k) – *переход к команде с номером k, если содержимое m-го и n-го регистров совпадает. В противном случае переход на следующую команду*.

МНР–программа – это последовательность команд перечисленных выше типов.

Предполагаем:

а) перед началом программы задана начальная конфигурация регистров (начальные значения регистров);

б) результат программы помещается в регистр с номером 1;

в) программа останавливается при переходе к команде, номер которой больше номера последней команды;

г) после выполнения команд Z(n), S(n), T(m,n) осуществляется переход к следующей команде.

Определить значение, вычисляемое данным алгоритмом.

Входной файл INPUT.TXT содержит:

а) число регистров, используемых МНР – программой;

б) массив целых чисел – начальные значения регистров;

в) МНР – программу, т.е. последовательность команд.

Выходной файл OUTPUT.TXT:

содержит число – значение первого регистра.

П р и м е р

Входной файл INPUT.TXT

```
3
2 0 0
J(1,2,4)
S(2)
T(2,1)
```

Выходной файл OUTPUT.TXT:

```
1
```

Задача 6 (4 балла)

Телефонный номер называется "шахматным", если его цифры набираются на телефонном кнопочном номеронабирателе ходом шахматного коня. Написать программу, подсчитывающую, сколько можно набрать различных семизначных "шахматных" номеров, начинающихся с заданной цифры:

1 2 3
4 5 6
7 8 9
0

Технические требования:

Входной файл: INPUT.TXT содержит число [0..9].

Выходной файл: OUTPUT.TXT должен содержать единственное число – решение задачи.

Задача 7 (5 баллов) ("Минимальное покрытие")

Среди заданного множества отрезков прямой с целочисленными координатами концов [Li, Ri] необходимо выбрать подмножество наименьшей мощности, целиком покрывающее отрезок [0, M], где M – натуральное число.

Ограничения: $1 \le M \le 5000$; $|Li|, |Ri| \le 5000$; $I \le 10000$.

В первой строке входного файла INPUT.TXT указана константа M. В последующих строках перечислены пары чисел (L_i, R_i), каждая пара чисел начинается с новой строки. Числа в парах отделены друг от друга одним или несколькими пробелами. Список завершается парой чисел (0, 0).

Программа должна формировать в первой строке выходного файла OUTPUT.TXT требуемое минимальное число отрезков из исходного множества, необходимое для покрытия отрезка [0, M]. Далее должен следовать список покрывающего подмножества, упорядоченного по возрастанию координат левых концов отрезка. Список отрезков выводится в том же формате, что и во входном файле INPUT.TXT, завершающую пару (0, 0) выводить не следует.

Если покрытие отрезка (0, M) исходным множеством отрезков (L_i, R_i) невозможно, то файл OUTPUT.TXT должен содержать единственную фразу "No solution".

П р и м е р ы входного и выходного файлов.

1) INPUT.TXT	2) INPUT.TXT
1	1
-1 0	-1 0
-5 -3	0 1
2 5	0 0
0 0	
OUTPUT.TXT	OUTPUT.TXT
No solution	1
	0 1

Задача 8 (6 баллов)

На треугольном поле, устроенном так, как показано на рис. 17, клетки пронумерованы последовательными натуральными числами от единицы до бесконечности.

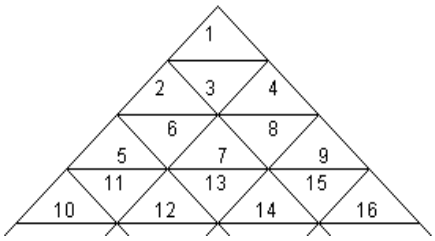


Рис. 17

Путешественнику требуется пройти из клетки с номером M в клетку с номером N. Путешественник может попасть в соседние клетки только через ребра треугольников (не через вершины). Количество ребер, которое ему нужно будет пересечь в пути, называется длиной маршрута.

Напишите программу, которая вычисляет длину кратчайшего маршрута для заданных точек M и N.

Во входном файле INPUT.TXT содержатся числа M и N, разделенные одним или несколькими пробелами. Числа M и N – натуральные, не менее единицы и не более одного миллиарда.

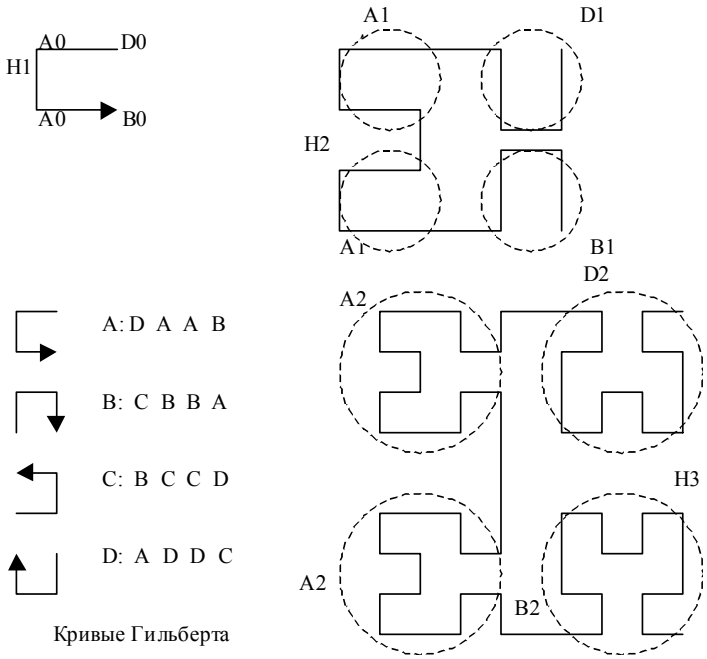
Программа должна выдать в выходной файл OUTPUT.TXT длину кратчайшего пути из M в N.

П р и м е р ы входного и выходного файлов.

INPUT.TXT
12
OUTPUT.TXT
3

Задача 9 (4 балла)

Построить кривые Гильберта порядка N. Принцип построения кривых приведен на рис. 18.



Кривые Гильберта

17.2 Примеры решения заданий повышенного уровня сложности

Для анализа приведено несколько вариантов решения задач. Восстановите по тексту каждой программы алгоритм ее действия, оцените его оптимальность, предложите свои варианты.

Решение задачи 1 ("Магический квадрат")

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

int **cub;


void error(int swt)

{
	switch(swt)
	{
	    case 1: printf("Can't find text file \"input.txt\".\n");
	    printf("Work of programm has finished. Press <ENTER> to halt");
	    getchar(); clrscr(); break;
	    case 2: printf("Can't create text file \"output.txt\".\n");
	    printf("Work of programm has finished. Press <ENTER> to halt");
	    getchar(); clrscr(); break;
	}
}

int StrToInt(char *number)
{
	int i, dlina, dec, num;
	num=0;
	dec=1;
	dlina=strlen(number);
	for(i=dlina;i>0;i--)
	{
	    num=num+(number[i-1]-48)*dec;
	    dec=dec*10;
	}
	return num;
}
void nechet(int num)
{
	int i, j, k, schet, a, b;
	cub=(int**)malloc(num*sizeof(int*));
	for(i=0;i<num;i++) cub[i]=(int*)malloc(num*sizeof(int));
	for(i=0;i<num;i++) for(j=0;j<num;j++) cub[i][j]=0;
	i=0;
	j=(num+1)/2-1;
	k=num*num;
	schet=1;
	while(schet!=k+1)
	{
	    if(cub[i][j]==0)
	    { cub[i][j]=schet; schet++; i--; j++;}
	    else {i+=2; j--;}
	    if(i<0) i=num+i;
	    if(j>num-1) j=0;
	    if(i>num-1) i=i-num; if(j<0) j=num+j;
	}
}

void chet4(int num)
{
	int i, j;
	int k, m, s, b, t;
```

```
	cub=(int**)malloc(num*sizeof(int*));
	for(i=0;i<num;i++) cub[i]=(int*)malloc(num*sizeof(int));
	for(i=0;i<num;i++) for(j=0;j<num;j++) cub[i][j]=i*num+j+1;
	j=2;
	m=num/2;
	for(i=1;i<=m;i++)
	    for(k=1;k<=m/2;k++)
	    {
			if(j==m+1) j=2;
			else if(j==m+2) j=1;
			s=num-i+1; b=num-j+1;
			t=cub[i-1][j-1]; cub[i-1][j-1]=cub[s-1][b-1];
			cub[s-1][b-1]=t;
			t=cub[i-1][b-1]; cub[i-1][b-1]=cub[s-1][j-1];
			cub[s-1][j-1]=t;
			j=j+2;
	    }
}
void chet2(int num)
{
	int i, j, schet;
	int k, r, m, s, b, t;
	cub=(int**)malloc(num*sizeof(int*));
	for(i=0;i<num;i++) cub[i]=(int*)malloc(num*sizeof(int));
	for(i=0;i<num;i++) for(j=0;j<num;j++) cub[i][j]=i*num+j+1;
	r=(num/2-1)/2; m=num/2;
	for(i=1;i<=m;i++)
	{
	    j=i;
	    for(k=1;k<=r;k++)
	    {
			if(j>m) j=1;
			s=num-i+1; b=num-j+1;
			t=cub[i-1][j-1]; cub[i-1][j-1]=cub[s-1][b-1];
			cub[s-1][b-1]=t;
			t=cub[i-1][b-1]; cub[i-1][b-1]=cub[s-1][j-1];
			cub[s-1][j-1]=t;
			j++;
	    }
	}
	i=1; j=r+1;
	for(k=1;k<=m;k++)
	{
	    if(j>m) j=1;
	    s=num-i+1;
	    t=cub[i-1][j-1]; cub[i-1][j-1]=cub[s-1][j-1]; cub[s-1][j-1]=t;
	    i++; j++;
	}
	i=1; j=r+2;
	for(k=1;k<=m;k++)
	{
	    if(j>m) j=1;
	    b=num-j+1;
	    t=cub[i-1][j-1]; cub[i-1][j-1]=cub[i-1][b-1]; cub[i-1][b-1]=t;
	    i++; j++;
	}
}

int main()
{
	int i, j, num, schet;
	char *number;
	FILE *input, *output;
	printf("Programm is working...\n\n");
	input=fopen("input.txt","rt");
	output=fopen("output.txt","wt");
	if(input==NULL) {error(1); return 0;}
```

```
        if(output==NULL) {error(2); return 0;}
        fscanf(input,"%s",number);
        num=StrToInt(number);
        if(num%2==1) nechet(num);
        if(num%4==0) chet4(num);
        if(num%2==0 && num%4!=0) chet2(num);
        if(num==3) schet=1;
        else
            if(num>3 && num<10) schet=2;
            else schet=3;
        switch(schet)
        {
            case 1: for(i=0;i<num;i++)
                    {
                      for(j=0;j<num;j++)
                                fprintf(output,"%i ",cub[i][j]);
                      fprintf(output,"\n");
                     }
            break;
            case 2: for(i=0;i<num;i++)
                    {
                      for(j=0;j<num;j++)
                                fprintf(output,"%.2i ",cub[i][j]);
                      fprintf(output,"\n");
                    }
            break;
            case 3: for(i=0;i<num;i++)
                     {
                      for(j=0;j<num;j++)
                                fprintf(output,"%.3i ",cub[i][j]);
                      fprintf(output,"\n");
                     }
        }
        free(cub);
        fclose(input);
        fclose(output);
        printf("End of work. Press <ENTER> to exit"); getchar(); clrscr();
        return 0;
        }
```

Решение задачи 5

```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>

int number,string;
int *massiv;
char temp[80];
FILE *input,*output;


void error(int swt)
{
        switch(swt)
        {case 1: printf("Can't find text file \"input.txt\".\n");
             printf("Work of programm has finished. Press <ENTER> to halt");
             getchar(); clrscr(); break;
         case 2: printf("Can't create text file \"output.txt\".\n");
             printf("Work of programm has finished. Press <ENTER> to halt");
             getchar(); clrscr(); break;
         case 3: printf("The number of registrs must be more than 0!\n");
             printf("Work of programm has finished. Press <ENTER> to halt");
             getchar(); clrscr(); break;
```

```
        case 4: printf("Wrong syntaksis in file!\n");
            printf("Work of programm has finished. Press <ENTER> to halt");
            getchar(); clrscr(); break;
        case 5: printf("No such command exists!\n");
            printf("Work of programm has finished. Press <ENTER> to halt");
            getchar(); clrscr(); break;
        case 6: printf("Command tried jump to non-existing string in file.\n");
            printf("Work of programm has finished. Press <ENTER> to halt");
            getchar(); clrscr(); break;
    }
}
void z(char swt)
{
    int i=-1, j, num=0;
    char litera, str[5];
    litera=fgetc(input);
    if(litera!='(')
    {fprintf(output,"With error\n%i",massiv[0]); error(4); exit(0);}
    while(litera!=')') {i+=1; str[i]=fgetc(input); litera=str[i];}
    for(j=0;j<i;j++) num+=(str[j]-48)*pow(10,i-j-1);
    if(swt=='z') massiv[num-1]=0;
    else massiv[num-1]+=1;
    fgetc(input);
}


void t(void)
{
    int i, j, k, num[2];
    char litera, str[5];
    for(i=0;i<2;i++) num[i]=0;
    litera=fgetc(input);
    if(litera!='(')
    {fprintf(output,"With error\n%i",massiv[0]); error(4); exit(0);}
    i=-1;
    for(k=0;k<2;k++)
    {
        while(litera!=',' && litera!=')')
        {i+=1; str[i]=fgetc(input); litera=str[i];}
        for(j=0;j<i;j++) num[k]+=(str[j]-48)*pow(10,i-j-1);
        i=-1;
        litera='0';
    }
    massiv[num[1]-1]=massiv[num[0]-1];
    fgetc(input);
}


void J(void)
{
    int i, j, k, num[3];
    char litera, str[5];
    for(i=0;i<3;i++) num[i]=0;
    litera=fgetc(input);
    if(litera!='(')
    {fprintf(output,"With error\n%i",massiv[0]); error(4); exit(0);}
    i=-1;
    for(k=0;k<3;k++)
    {
        while(litera!=',' && litera!=')')
        {i+=1; litera=fgetc(input); str[i]=litera;}
        for(j=0;j<i;j++) {num[k]+=(str[j]-48)*pow(10,i-j-1);}
        litera='0';
        i=-1;}
```

```
        if(num[2]<1)
        {fprintf(output,"With error\n%i",massiv[0]); error(4); exit(0);}
        if(massiv[num[1]-1]==massiv[num[0]-1])
        {
            if(num[2]>string)
            {fprintf(output,"With error\n%i",massiv[0]); error(6); exit(0);}
            else
            {
                        rewind(input);
                        for(i=-1;i<num[2];i++) fgets(temp,80,input);
            }
        }
        else litera=fgetc(input);
}


int main()
{
        int i, j;
        char litera;
        printf("Programm is working...\n\n");
        input=fopen("input.txt","rt");
        output=fopen("output.txt","wt");
        if(input==NULL) {error(1); return 0;}
        if(output==NULL) {error(2); return 0;}
        i=0;
        while(!feof(input)) {fgets(temp,80,input); i++;}
        string=i-2;
        rewind(input);
        fscanf(input,"%d",&number);
        if(number<1) {error(3); return 0;}
        massiv=(int*)malloc(number*sizeof(int));
        for(i=0;i<number;i++) fscanf(input,"%i",&massiv[i]);
        litera=fgetc(input);
        while(!feof(input))
        {
            litera=fgetc(input);
            switch(litera)
            {case 74: J(); break;
              case 83: z('s'); break;
              case 84: t(); break;
              case 90: z('z'); break;
              case 106: J(); break;
              case 115: z('s'); break;
              case 116: t(); break;
              case 122: z('z'); break;
              default: if(number>0) fprintf(output,"With error\n%i",massiv[0]);
              error(5); return 0;
            }
        }
        fprintf(output,"%i",massiv[0]);
        free(massiv);
        fclose(input);
        fclose(output);
        printf("End of work. Press <ENTER> to exit"); getchar(); clrscr();
        return 0;
}
```

Решение задачи 6

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

```
#define num 7

int summa=0;
int mas[10][4]={  {0,5,-1,-1},
                  {1,6,8,-1},
                  {2,7,9,-1},
                  {3,4,8,-1},
                  {4,3,9,-1},
                  {5,-1,-1,-1},
                  {6,1,7,-1},
                  {7,2,6,-1},
                  {8,1,3,-1},
                  {9,0,2,4}
                };
void schet(int a,int b)
{
      int i;
      for(i=1;i<4;i++)
      if(mas[a][i]>=0)
          if(b==num-1) summa++;
          else schet(mas[a][i],b+1);
}


int main()
{
      int i, number;
      FILE *input, *output;
      input=fopen("input.txt","rt");
      fscanf(input,"%i",&number); fclose(input);
      if(number>=10 || number<0)
      {printf("Error::Number in file must be in [0..9]!!!\n");
       printf("Press <ENTER> to halt programm");
       getch(); clrscr(); exit(0);
      }
      printf("You entered: %i\nProcess working...\n.....\n",number);
      output=fopen("output.txt","wt");
      schet(number,1); printf("summa=%i\n",summa);
      fprintf(output,"%i",summa); fclose(output);
      printf("You'll find this answer in file 'output.txt'\n");
      printf("End of work. Press <ENTER> to escape"); getch(); clrscr();
      return 0;
}
```

Решение задачи 7

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

struct pokr
{int a,b,in;
 struct pokr *next;
};
struct pokr *start,*uk;
int **mas,m,n;
FILE *input,*output;

void init(void)
{
      input=fopen("input.txt","rt");
      output=fopen("output.txt","wt");
      if(input==NULL || output==NULL)
      {printf("Error on open file 'input.txt' or on create file 'output.txt'!\n");
```

```
        printf("End of work. Press <ENTER> to halt"); getch(); clrscr(); exit(1);
       }
}

void work(void)
{
       int i, j;
       fscanf(input,"%i",&m); n=0; i=j=0;
       do
       {fscanf(input,"%i",&i); fscanf(input,"%i",&j);
        if(j<i) {i=i+j; j=i-j; i=i-j;}
        if(i<j) ;
         {if(start==NULL) {start=malloc(sizeof(struct pokr)); uk=start;}
          else {uk->next=malloc(sizeof(struct pokr)); uk=uk->next;}
         uk->next=NULL; uk->a=i; uk->b=j;
         if(i<=0 && j>=m) {uk->in=1; n++;}
          else uk->in=0;
         }
       }while(i<j);
       fclose(input);
}

void progon(void)
{
       int i, j;
       fprintf(output,"%i\n",n);
       for(i=0;i<n;i++)
       {for(j=i+1;j<n;j++)
         {if(mas[j][0]<mas[i][0])
          {mas[i][0]=mas[i][0]+mas[j][0]; mas[j][0]=mas[i][0]-mas[j][0];
        mas[i][0]=mas[i][0]-mas[j][0];
           mas[i][1]=mas[i][1]+mas[j][1]; mas[j][1]=mas[i][1]-mas[j][1];
        mas[i][1]=mas[i][1]-mas[j][1];
         }
        }
        fprintf(output,"%i %i\n",mas[i][0],mas[i][1]);
       }
}

void sort(void)
{
       int i, j;
       uk=start;
       switch(n)
       {case 0: fprintf(output,"No solution"); break;
        default: mas=(int**)malloc(n*sizeof(int*));
        for(i=0;i<2;i++) mas[i]=(int*)malloc(2*sizeof(int));
       uk=start;
       for(i=0;i<n;i++)
        {while(!uk->in) uk=uk->next;
        mas[i][0]=uk->a; mas[i][1]=uk->b;
        uk=uk->next;
        }
       progon();
       }
       free(mas); fclose(output);
}

void del(void)
{
       uk=start->next;
        while(start!=NULL)
        {start->next=NULL;
         free(start);
         start=uk;
         if(start!=NULL) uk=start->next;
```

```
    }
}

int main()
{
    init(); work();
    sort(); del();
    printf("End of work. Press <ENTER> to escape");
    getch(); clrscr();
    return 0;
}
```

18 ЗАДАНИЯ ДЛЯ САМОПОДГОТОВКИ

При выполнении заданий для самостоятельного закрепления изученного материала рекомендуется предварительно создавать алгоритм решаемой задачи. Разрабатываемые программы должны быть максимально универсальными и наиболее полно использовать возможности языка С.

1 Дано натуральное число n. Получить все тройки натуральных чисел a, b, c, каждое из которых не более n, соответствующих условию теоремы Пифагора: $a^2 + b^2 = c^2$.

2 Дано натуральное число n. Наити в диапазоне 1...n все числа Мерсена. Числом Мерсена называется простое число, если его можно представить в виде $2^p - 1$ (здесь p – простое число).

3 Заданы два натуральных числа n и m ($n < m$). Найти и вывести на печать все пары дружественных чисел в диапазоне от n до m. Два натуральных числа называются дружественными, если каждое из них равно сумме делителей другого числа, исключая само это число.

4 Дано натуральное число n. Среди чисел 1...n найти все числа, запись которых совпадает с младшими цифрами их квадратов (пример: $\mathbf{25}^2 = 6\mathbf{25}$).

5 Дано натуральное число $n > 9$. В диапазоне 10...n найти все числа Армстронга. Число из k цифр считается числом Армстронга, если оно равно сумме своих цифр, возведенных в k-ю степень (пример: $153 = 1^3 + 5^3 + 3^3$).

6 Число считается палиндромом, если его запись читается слева направо и справа налево одинаково. Проверить, является ли заданное натуральное число n палиндромом. При написании программы не пользоваться массивом.

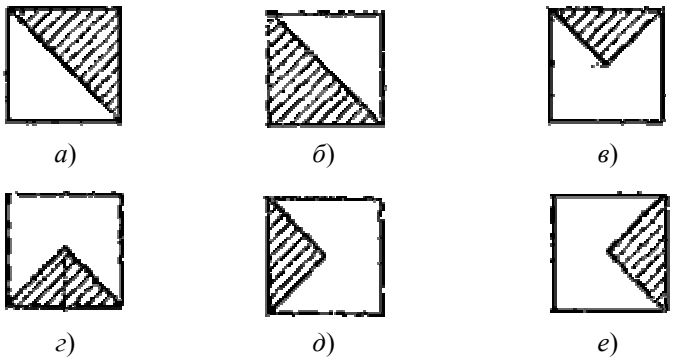
7 Заданы два натуральных числа n и m. С помощью алгоритма Евклида найти наибольший общий делитель (НОД) этих чисел. Алгоритм Евклида основывается на том, что НОД двух неотрицательных чисел $n \le m$ можно вычислить следующим образом: если $n = 0$, то НОД (n, m) = m. Иначе НОД (m, n) = НОД (n, r), где r – остаток от деления m на n.

8 Использовать алгоритм Евклида (см. задачу 7) для нахождения наименьшего общего кратного двух натуральных чисел n и m.

9 Написать программу для перевода натурального числа n из десятичной системы счисления в систему счисления по основанию k ($k < 10$).

10 Дано натуральное число $n < 4000$. Записать его римскими цифрами. При записи чисел римскими цифрами используются следующие обозначения: **I** – 1, **V** – 5, **X** – 10, **L** – 50, **C** – 100, **D** – 500, **M** – 1000.

11 Дана квадратная матрица порядка n (рис. 19). Найти ее наибольший и наименьший элементы в указанных частях матрицы.



а) *б*) *в*)

г) *д*) *е*)

Рис. 19

12 Дана матрица порядка $n \times m$ (рис. 20). Сортировать ее с помощью метода простых вставок. После сортировки расположить элементы указанным образом.

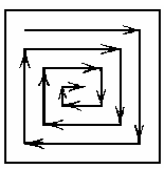
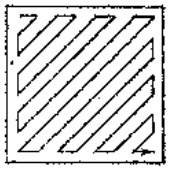


Рис. 20

13 Дана матрица порядка $n \times m$ (рис. 21). Сортировать ее с помощью метода бинарных вставок. После сортировки расположить элементы указанным образом.



14 Дана матрица порядка $n \times m$ (рис. 22). Сортировать ее с помощью метода обменов. После сортировки расположить элементы указанным образом.

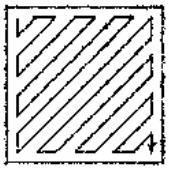


Рис. 21

15 Дана матрица порядка $n \times m$ (рис. 23). Сортировать ее с помощью метода обменов с изменением направления. После сортировки расположить элементы указанным образом.

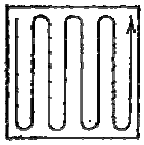


Рис. 23

16 Дана матрица порядка $n \times m$ (рис. 24). Сортировать ее с помощью метода выбора. После сортировки расположить элементы указанным образом.

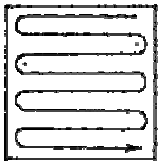


Рис. 24

17 Дана квадратная матрица порядка n (рис. 25). Получить новую матрицу путем перестановки указанных ее частей.

а) *б*)

Рис. 25

18 Проверить, является ли квадратная матрица порядка n магическим квадратом. Магическим квадратом является квадратная матрица, в которой суммы элементов по строкам, столбцам и диагоналям равны.

19 Для заданного числа x ($x \neq 0$) произвести вычисления с точностью ε ($\varepsilon > 0$) по формуле:

а) $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{i!(2i+1)}$;

б) $\sum_{i=0}^{\infty} \frac{(-1)^i x^{4i+1}}{(i+1)!2^{i+1}}$;

в) $\sum_{i=0}^{\infty} \frac{(-1)^{i+1}}{i!(i+1)!}\left(\frac{x}{2}\right)^{2i+1}$;

г) $\sum_{i=0}^{\infty} \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)(2i+1)!}$;

д) $\sum_{i=0}^{\infty} \frac{(-1)^i x^{i+2}}{(2i)!(4i+1)}$;

е) $\sum_{i=0}^{\infty} \frac{(-1)^i}{((i+1)!)}\left(\frac{x}{4}\right)^{2(i+1)}$.

20 Задан текстовый файл. Группы символов, разделенные пробелами, считаются словами. Группа символов, не содержащая буквы, считается числом. Найти максимальное число в файле (число может быть вещественным, т.е. содержать десятичную точку).

21 С клавиатуры вводится арифметическое выражение, содержащее неотрицательные целые числа, меньшие тысячи, и знаки операций '+', '-', '*', '/'. Записать это выражение в файл словами.

22 Задано натуральное число n и текстовый файл. Отформатировать строки текста в файле по длине n. Начало абзаца отметить табуляцией.

23 Задан файл, хранящий записную книжку – фамилию абонента, его адрес и телефон. Организовать работу с записной книжкой: добавление новых сведений, коррекцию и удаление существующих записей, а также поиск информации по указанной с клавиатуры фамилии. При написании программы использовать бинарный режим работы с файлом.

24 Задан текстовый файл. Осуществить шифрование/дешифро-вание информации следующими методами:

а) методом смещения кода – к коду каждого считанного символа прибавляется фиксированное смещение;

б) методом перестановки – в считанной из файла последовательности из *k* символов осуществляется замена символов местами по заданной перестановке;

в) методом решетки – символы записываются из файла в матрицу порядка *n* в одном направлении, а считываются в другом.

Результат криптографической операции сохранить в другом файле.

25 Заданы три натуральных числа *d*, *m*, *y*, обозначающие дату. Вывести на печать дату, следующую за указанной.

26 Заданы три натуральных числа *d*, *m*, *y*, обозначающие дату. Определить на какой день недели приходится указанная дата.

27 Заданы натуральные числа *d1*, *m1*, *y1*, *d2*, *m2*, *y2*, обозначающие две даты. Определить полное число дней между этими датами.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1 Абрамов, С.А. Задачи по программированию / С.А. Абрамов [и др.]. – М. : Наука, 1988.

2 Ахо, А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. – М. : Издательский дом "Вильямс", 2000.

3 Баррон, Д. Рекурсивные методы в программировании / Д. Бар-рон. – М. : Мир, 1974.

4 Бентли, Д. Жемчужины творчества программистов / Д. Бентли. – М. : Радио и связь, 1990.

5 Бочков, С.О. Язык программирования Си для персонального компьютера / С.О. Бочков, Д.М. Субботин. – М. : Радио и связь, 1990.

6 Ван Тассел, Д. Стиль, разработка, эффективность, отладка и испытание программ / Д. Ван Тассел. – М. : Мир, 1985.

7 Дал, У. Структурное программирование / У. Дал, Э. Дейкстра, К. Хоор. – М. : Мир, 1975.

8 Данные в языках программирования: абстракция и типология. – М. : Мир, 1982.

9 Зиглер, К. Методы проектирования программных систем / К. Зиглер. – М. : Мир, 1985.

10 Зиновкина, М.М. Креативное инженерное образование. Теоретические и инновационные креативные педагогические технологии / М.М. Зиновкина. – М. : МГИУ, 2003.

11 Калинин, А.Г. Универсальные языки программирования. Семантический подход / А.Г. Калинин, И.В. Мацкевич. – М. : Радио и связь, 1991.

12 Кнут, Д. Искусство программирования : в 7 т.; т. 3. Сортировка и поиск / Д. Кнут. – М. : Мир, 1978.

13 Липаев, В.В. Проектирование программных средств / В.В. Липаев. – М. : Наука, 1990.

14 Нешумова, К.А. Электронные вычислительные машины и системы / К.А. Нешумова. – М. : Высшая школа, 1989.

15 Подбельский, В.В. Язык Си++ / В.В. Подбельский. – М. : Финансы и статистика, 1995.

16 Попов, А.И. Решение творческих профессиональных задач : учеб. пособие / А.И. Попов. – Тамбов : Изд-во Тамб. гос. тех. ун-та, 2004.

17 Программно-информационные комплексы автоматизированных производственных систем / под ред. С.А. Клеймено-ва. – М. : Высшая школа, 1990.

18 Сергеев, А.П. Алгоритмизация и программирование / А.П. Сергеев, Л.Н. Домнин. – М. : Радио и связь, 1984.

19 Требования и спецификации в разработке программ. – М. : Мир, 1984.

ОГЛАВЛЕНИЕ