Подбельский В. В., Фомин С. С.

Курс программирования на языке Си

Учебник

Допущено Учебно-методическим объединением вузов РФ по образованию в области экономики, менеджмента, логистики и бизнес-информатики в качестве учебника для студентов высших учебных заведений, обучающихся по направлению «Бизнес-информатика»

Москва, 2013

УДК 681.3.06:800.92(075.8)
ББК 32.973.2-018.1
П44

Подбельский В. В., Фомин С. С.

П44 Курс программирования на языке Си: учебник. – М.: ДМК Пресс, 2012. – 384 с.

ISBN 978-5-94074-449-8

Книга является полным курсом программирования на стандартном языке Си. Рассматриваются все средства языка Си, не зависящие от реализаций, существующие в компиляторах на современных ПК различных платформ.

Для студентов и преподавателей вузов, а также для желающих освоить самостоятельно программирование на языке Си.

УДК 681.3.06:800.92(075.8)
ББК 32.973.2-018.1

Подбельский Вадим Валериевич
Фомин Сергей Сергеевич

Курс программирования на языке Си

Учебник

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 29.08.2012. Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 24. Тираж 1000 экз.

Веб-сайт издательства: www.dmk-press.ru

ISBN 978-5-94074-449-8
© Подбельский В. В., Фомин С. С., 2012
© Оформление, ДМК Пресс, 2012

Содержание

ПРЕДИСЛОВИЕ 6

Глава 1
БАЗОВЫЕ ПОНЯТИЯ ЯЗЫКА 10

1.1. Алфавит, идентификаторы, служебные слова 11
1.2. Литералы 14
1.3. Переменные и именованные константы 21
1.4. Операции 30
1.5. Разделители 39
1.6. Выражения 44
Контрольные вопросы 54

Глава 2
ВВЕДЕНИЕ
В ПРОГРАММИРОВАНИЕ НА Си 56

2.1. Структура и компоненты простой программы 56
2.2. Элементарные средства программирования 66
2.3. Операторы цикла 77
2.4. Массивы и вложение операторов цикла 87
2.5. Функции 95
2.6. Переключатели 108
Контрольные вопросы 111

Глава 3
ПРЕПРОЦЕССОРНЫЕ СРЕДСТВА 113

3.1. Стадии и директивы препроцессорной обработки 113
3.2. Замены в тексте 117
3.3. Включение текстов из файлов 122
3.4. Условная компиляция 125
3.5. Макроподстановки средствами препроцессора 129
3.6. Вспомогательные директивы 135
3.7. Встроенные макроимена 138
Контрольные вопросы 139

Глава 4
УКАЗАТЕЛИ, МАССИВЫ, СТРОКИ ... 141
4.1. Указатели на объекты ... 141
4.2. Указатели и массивы ... 150
4.3. Символьная информация и строки ... 165
Контрольные вопросы ... 174

Глава 5
ФУНКЦИИ ... 176
5.1. Общие сведения о функциях ... 176
5.2. Указатели в параметрах функций ... 181
5.3. Массивы и строки как параметры функций ... 186
5.4. Указатели на функции ... 196
5.5. Функции с переменным количеством аргументов ... 211
5.6. Рекурсивные функции ... 223
5.7. Классы памяти и организация программ ... 227
5.8. Параметры функции main( ) ... 234
Контрольные вопросы ... 237

Глава 6
СТРУКТУРЫ И ОБЪЕДИНЕНИЯ ... 239
6.1. Структурные типы и структуры ... 239
6.2. Структуры, массивы и указатели ... 252
6.3. Структуры и функции ... 262
6.4. Динамические информационные структуры ... 267
6.5. Объединения и битовые поля ... 274
Контрольные вопросы ... 281

Глава 7
ВВОД И ВЫВОД ... 283
7.1. Потоковый ввод-вывод ... 283
7.1.1. Открытие и закрытие потока ... 284
7.1.2. Стандартные потоки и функции для работы с ними ... 288
7.1.3. Работа с файлами на диске ... 308
7.2. Ввод-вывод нижнего уровня ... 322
7.2.1. Открытие/закрытие файла ... 323
7.2.2. Чтение и запись данных ... 328
7.2.3. Произвольный доступ к файлу ... 331
Контрольные вопросы ... 333

Глава 8
ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ 334

8.1. Схема подготовки программ 334
8.2. Подготовка программ в операционной системе UNIX.. 336
8.3. Утилита make 338
8.4. Библиотеки объектных модулей 343
Контрольные вопросы 349

Приложение 1
ТАБЛИЦЫ КОДОВ ASCII 350

Приложение 2
КОНСТАНТЫ ПРЕДЕЛЬНЫХ ЗНАЧЕНИЙ 357

Приложение 3
СТАНДАРТНАЯ БИБЛИОТЕКА ФУНКЦИЙ ЯЗЫКА СИ 359

Приложение 4
МОДЕЛИ ПРЕДСТАВЛЕНИЯ ЧИСЕЛ НА РАЗЛИЧНЫХ КОМПЬЮТЕРНЫХ ПЛАТФОРМАХ 369

Литература 372

Предметный указатель.. 373

ПРЕДИСЛОВИЕ

В 2012 г. языку Си исполняется 40 лет. Как латинский язык явился основой многих европейских языков, так и язык Си стал родоначальником языков Си++, Java, Perl, C#, PHP, JavaScript и т. д. В отличие от мертвой латыни, язык Си – не только живой язык, но и наиболее распространенный и эффективный из универсальных языков программирования. Программы на языке Си исполняются почти на всех компьютерах, он работает со средой программирования UNIX, и сама операционная система UNIX написана на нем. На Си написано множество библиотечных функций и утилит. Сотни тысяч программистов знают язык Си, он неотделим от общечеловеческой культуры программирования. Любой программист должен владеть языком Си, чтобы профессионально работать в области информационных технологий и понимать своих коллег. В учебных программах и стандартах высших и средних специальных учебных заведений для большинства естественно-научных специальностей предусмотрено изучение программирования на языке Си.

Язык программирования Си создан в 1972 г. сотрудником фирмы Bell Laboratories Деннисом Ритчи (Dennis M. Ritchie) при разработке операционной системы UNIX. Язык проектировался как инструмент для системного программирования с ориентацией на разработку хорошо структурированных программ. Удачное сочетание лаконичности конструкций и богатства выразительных возможностей позволило языку Си быстро распространиться и стать наиболее популярным языком прикладного и системного программирования. Компиляторы языка Си работают почти на всех типах современных ЭВМ в операционных системах Windows, UNIX-подобных ОС (FreeBSD, Linux), Solaris, Mac OS и др.

В отличие от многих предшествующих языков (Ада, Алгол-60, Алгол-68 и т. д.), которые вступали в силу после принятия соответствующих национальных и международных стандартов, язык Си вначале был создан как рабочий инструмент, не претендующий на широкое применение. Стандарта на язык Си до 1989 г. не сущест-

вовало, и в качестве формального описания разработчики компиляторов использовали первое издание книги Б. Кернигана и Д. Ритчи, вышедшее в США в 1978 г. (переведена на русский язык в 1985 г. [1]). Роль неформального стандарта языка Си сохранилась за этой книгой и в настоящее время. Не случайно в литературе и документации по компиляторам ссылка на эту работу обозначается специальным сокращением K&R.

Второе издание книги Б. Кернигана и Д. Ритчи [2] описывает язык Си в стандартизованном Американским институтом национальных стандартов виде (стандарт ANSI языка Си). В настоящее время, кроме стандарта ANSI C, разработаны международный стандарт ISO C (International Standard Organization C) и стандарт 1999 года **C99** – современный стандарт языка программирования Си. Определен в ISO/IEC 9899:1999, современная версия – ISO/IEC 9899:1999/ Cor 3:2007 от 2007-11-15. Эти версии стандарта близки друг к другу, и на различиях между стандартами нет необходимости останавливаться до возникновения разногласий в толковании той или иной конструкции языка либо при оценке стандартности конкретного компилятора. Эти ситуации выходят за рамки курса по программированию на языке Си. В случае необходимости получения справок по стандартам языка Си следует обращаться к специальным публикациям. Неформальное применение книги K&R в качестве стандарта до 1989 г. и последующая ее переработка авторами в соответствии с принятым стандартом ANSI привели к тому, что ее и сейчас можно рассматривать как достоверный источник при получении справок по языку Си.

Настоящий учебник предназначен для изучения программирования на стандартном языке Си. Ориентация сделана как на изложение синтаксиса и семантики конструкций языка, так и на их практическое использование при решении типовых задач программирования. После описания в главе 1 основных понятий языка Си рассмотрены средства представлений базовых конструкций структурного программирования, возможности которых в главе 2 иллюстрируются на простых вычислительных задачах. Глава 3 содержит подробное описание препроцессорных средств компилятора языка Си, которые активно используются при последующем изучении методов и приемов программирования на языке Си. Следующая глава и посвящена незаменимым в системном программировании понятиям – объектам и адресам (указателям). Аппарат указателей используется затем при обработке массивов и строк. Центральное

место занимает глава 5, посвященная функциям. Здесь возможности функций рассмотрены подробно и с нужной полнотой. Особое внимание уделено взаимосвязи функций с указателями, а также классам памяти, которые вводятся в связи с организацией многофайловых программ, включающих много функций. Глава 6 рассматривает структурированные данные (структуры и объединения). Особенности работы с файлами, а также средства ввода-вывода показаны на типовых задачах в главе 7. В главе 8 приводятся сведения о подготовке и выполнении программ в среде семейства операционных систем UNIX.

Приводимые в учебнике программы сопровождаются результатами, полученными на ЭВМ. Более подробно с практическими приемами программирования на языке Си читатель может познакомиться, обратившись к «Практикуму по программированию на Си» [10].

Целью настоящего учебника является изложение методики и принципов корректного, структурированного программирования на языке Си. Программы, иллюстрирующие конструкции и возможности языка, написаны максимально понятно для читателя. Авторы нигде не гнались за эффективностью кода в ущерб его структурированности и простоты. Возможности современных компиляторов языка Си таковы, что они позволяют генерировать весьма эффективный код по тексту хорошо структурированной программы без специальных ухищрений программиста, направленных на повышение быстродействия или незначительную экономию памяти.

Книга написана на основе дисциплин, которые авторы в течение ряда лет преподавали в МИЭМе на факультете прикладной математики, на факультете автоматики и вычислительной техники и факультете повышения квалификации инженеров. Материал курса соответствует учебной программе дисциплины «Алгоритмические языки и программирование». Изучение указанной дисциплины, в частности языка Си, служит основой для курсов по математическому обеспечению ЭВМ и сетей, по операционным системам, построению компиляторов и системному программированию.

Авторы надеются, что книга поможет ликвидировать разрыв между техническими руководствами по реализации языка Си и потребностями в методическом обеспечении учебного процесса. Для чтения книги достаточно знать основы информатики. Поэтому пособие можно использовать как в вузе, так и в курсах информатики школ, гимназий, лицеев и техникумов. Необходимым условием освоения

материала книги является выполнение приведенных в ней примеров на любой ЭВМ, снабженной транслятором с языка Си.

Повышению качества рукописи способствовали замечания рецензентов.

Любые конструктивные замечания и предложения по улучшению учебника авторы с благодарностью примут и учтут в дальнейшем. Нам можно писать по адресу издательства.

Глава 1
БАЗОВЫЕ ПОНЯТИЯ ЯЗЫКА

Начиная изучать новый для вас алгоритмический язык программирования, необходимо выяснить следующие вопросы:

1. Каков алфавит языка и как правильно записывать его лексемы[1]?
2. Какие типы данных приняты в языке и как они определяются (описываются)?
3. Какие операции над данными допустимы в языке, как строятся с их помощью выражения и как они выполняются?
4. Какова структура программы, в какой последовательности размещаются операторы, описание и определения?
5. Как выводить (представлять пользователю) результаты работы программы?
6. Как реализованы оператор присваивания, условные операторы и операторы перехода?
7. Как вводить исходные данные для программы?
8. Какие специальные конструкции для организации циклов есть в языке?
9. Каков аппарат подпрограмм (процедур) и (или) подпрограмм-функций?

Затем следует приступать к составлению программ, углубляя в ходе программирования знание языка. Изложение материала в данном пособии почти соответствует описанной схеме изучения алгоритмических языков. Введя основные средства языка Си, будем рассматривать конкретные программы, а затем, переходя к новым классам задач, введем все конструкции языка и те средства, которые не упоминаются в перечисленных выше вопросах.

---

[1] Лексема – единица текста программы, которая при компиляции воспринимается как единое целое и по смыслу не может быть разделена на более мелкие элементы.

В начале первой главы рассмотрим алфавит, идентификаторы, константы, типы данных и операции языка. Этот базовый материал необходим для всех следующих глав. Не освоив перечисленных понятий, невозможно начинать программирование.

Традиционно перед изложением синтаксиса языка программирования авторы пособий дают неформальное введение, где на примерах иллюстрируют основные принципы построения программ на предлагаемом языке. Однако язык Си невелик, и его лексические основы можно рассмотреть весьма подробно уже в самом начале изучения. Поэтому начнем с алфавита и лексем.

1.1. Алфавит, идентификаторы, служебные слова

Алфавит. В алфавит языка Си входят:

- прописные и строчные буквы латинского алфавита (A, B, ..., Z, a, b, ..., z);
- цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- специальные знаки: " , { } | [ ] ( ) + – / % \ ; ' . : ? < = > _ ! & * # ~ ^;
- неизображаемые символы («обобщенные пробельные символы»), используемые для отделения лексем друг от друга (например, пробел, табуляция, переход на новую строку).

В комментариях, строках и символьных константах могут использоваться и другие литеры (например, русские буквы).

Комментарий формируется как последовательность знаков (символов), ограниченная слева знаками /*, а справа – знаками */. Например:

```
/* Это комментарий */
```

В стандартном языке Си комментарии запрещено вкладывать друг в друга, то есть запись:

```
/* текст-1 /* текст-2 */ текст-3 */
```

ошибочна – «текст-3» не считается комментарием.

В современных версиях языка Си (C89, C9x) можно использовать «комментарий в строке», начинающийся с двух символов «//» и продолжающийся до конца строки.

В языке Си шесть классов лексем: свободно выбираемые и используемые идентификаторы, служебные (ключевые) слова, константы, строки (строковые константы), операции (знаки операций), разделители (знаки пунктуации).

Идентификатор. Последовательность букв, цифр и символов подчеркивания «_», начинающаяся с буквы или символа подчеркивания, считается идентификатором языка Си. Примеры идентификаторов:

```
KOM_16,    size88,    _MIN,    TIME,    time
```

Прописные и строчные буквы различаются, то есть два последних идентификатора различны.

Идентификаторы могут иметь любую длину, но компилятор учитывает не более 31 символа от начала идентификатора. В некоторых компиляторах это ограничение еще более жесткое, и учитываются только первые 8 символов любого идентификатора. В этом случае идентификаторы NUMBER_OF_ROOM и NUMBER_OF_TEST в программе будут неразличимы.

Служебные (ключевые) слова. Идентификаторы, зарезервированные в языке, то есть такие, которые нельзя использовать в качестве свободно выбираемых программистом имен, называют служебными словами. Служебные слова определяют типы данных, классы памяти, квалификаторы типа, модификаторы, псевдопеременные и операторы. В стандарте языка определены следующие служебные слова:

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

Служебные слова в конструкциях языка используются по-разному.

Для обозначения типов данных они служат спецификаторами и квалификаторами типов. Последние уточняют свойства обозначенных типов данных.

К *спецификаторам типов* относятся:

- **char** – символьный;
- **double** – вещественный двойной точности с плавающей точкой;

- **enum** – перечисляемый тип (перечисление) – определение целочисленных констант, для каждой из которых вводятся имя и значение;
- **float** – вещественный с плавающей точкой;
- **int** – целый;
- **long** – целый увеличенной длины (длинное целое);
- **long long** – целый тип длиной не менее 64 бит;
- **short** – целый уменьшенной длины (короткое целое);
- **struct** – структура (структурный тип);
- **signed** – знаковый, то есть целое со знаком (старший бит считается знаковым);
- **union** – объединение (объединяющий тип);
- **unsigned** – беззнаковый, то есть целое без знака (старший бит не считается знаковым);
- **void** – отсутствие значения;
- **typedef** – вводит синоним обозначения типа (определяет сокращенное наименование для обозначения типа).

Квалификаторы типа:

- **const** – квалификатор объекта, имеющего постоянное значение, то есть доступного только для чтения;
- **volatile** – квалификатор объекта, значение которого может измениться без явных указаний программиста.

Квалификаторы типа информируют компилятор о необходимости и (или) возможности особой обработки объектов в процессе оптимизации кода программы.

Для обозначения классов памяти используются:

- **auto** – автоматический;
- **extern** – внешний;
- **register** – регистровый;
- **static** – статический.

Для построения операторов используются:

- **break** – завершить, прервать (например, цикл или переключатель);
- **case** – определяет вариант в операторе **switch**;
- **continue** – завершить текущую итерацию цикла (продолжить цикл, перейдя к следующей итерации);
- **default** – определяет действия при отсутствии нужного варианта в операторе **switch**;
- **do** – выполнять (заголовок оператора цикла с постусловием);

- **else** – входит в оператор **if**, определяя альтернативную ветвь;
- **for** – для (заголовок оператора параметрического цикла);
- **goto** – перейти (безусловный переход);
- **if** – «если» – обозначение условного оператора;
- **return** – возврат (из функции);
- **sizeof** – операция определения размера операнда (в байтах);
- **switch** – переключатель;
- **while** – «пока» (заголовок цикла с предусловием или завершение цикла **do**).

Конструкции языка, в которых используются служебные слова, будем определять по мере необходимости. Можно было бы не перечислять всех служебных слов, а вводить их по мере изложения языка, однако их запрещено использовать в качестве имен, выбираемых программистом, и поэтому для предупреждения возможных ошибок список служебных слов нужен уже на данном этапе.

Добавим еще одно соглашение, обычно соблюдаемое авторами компиляторов и стандартных библиотек языка Си. Идентификаторы, начинающиеся с одного или двух символов подчеркивания «_», зарезервированы для использования в библиотеках и компиляторах. Поэтому такие идентификаторы не рекомендуется выбирать в качестве имен в прикладной программе на языке Си. Следующее соглашение относительно имен относится уже не к стандарту и не к реализациям, а отображает стиль оформления текста программы. Рекомендуется при программировании имена констант записывать целиком заглавными буквами.

1.2. Литералы

По определению, константа представляет значение, которое не может быть изменено. Синтаксис языка определяет пять типов констант: символы, константы перечисляемого типа, вещественные числа, целые числа и нулевой указатель («null-указатель»). Все константы, кроме нулевого указателя, отнесены в языке Си к арифметическим.

Символы, или символьные константы. Для изображения отдельных знаков, имеющих индивидуальные внутренние коды, используются символьные константы. Каждая символьная константа – это лексема, которая состоит из изображения символа и ограничивающих апострофов. Например: 'A', 'a' , 'B' , '8' , '0' , '+' , ';' и т. д.

Внутри апострофов можно записать любой символ, изображаемый на дисплее или принтере в текстовом режиме. Однако в ЭВМ используются и коды, не имеющие графического представления на экране дисплея, клавиатуре или принтере. Примерами таких кодов служит код перехода курсора дисплея на новую строку или код возврата каретки (возврат курсора к началу текущей строки). Для изображения в программе соответствующих символьных констант используются комбинации из нескольких символов, имеющих графическое представление. Каждая такая комбинация начинается с символа '\' (обратная косая черта – backslash). Такие наборы литер, начинающиеся с символа '\', в литературе по языку Си называют управляющими последовательностями. Ниже приводится их список:

'\n' – перевод строки;
'\t' – горизонтальная табуляция;
'\r' – возврат каретки (курсора) к началу строки;
'\\' – обратная косая черта \;
'\'' – апостроф (одиночная кавычка);
'\"' – кавычка (символ двойной кавычки);
'\0' – нулевой символ;
'\a' – сигнал-звонок;
'\b' – возврат на одну позицию (на один символ);
'\f' – перевод (прогон) страницы;
'\v' – вертикальная табуляция;
'\?' – знак вопроса.

Обратите внимание на то, что перечисленные константы изображаются двумя и более литерами, а обозначают они одну символьную константу, имеющую индивидуальный двоичный код. Управляющие последовательности являются частным случаем эскейп-последовательностей (ESC-sequence), к которым также относятся лексемы вида '\ddd', '\xhh' или '\Xhh'.

'\ddd' – восьмеричное представление любой символьной константы. Здесь d – восьмеричная цифра (от 0 до 7). Например, '\017' или '\233'.

'\xhh' или '\Xhh' – шестнадцатеричное представление любой символьной константы. Здесь h – шестнадцатеричная цифра (от 0 до F). Например, '\x0b', '\x1A' и т. д.

В приложении 1 приведены таблицы числовых кодов символов. Зная значения кодов, можно изображать в виде эскейп-последова-

тельности любой символ, как воспроизводимый в текстовом режиме на дисплее (принтере), так и неизображаемый (управляющий).

Символьная константа (символ) имеет целый тип, то есть символы можно использовать в качестве целочисленных операндов в выражениях.

Целые константы. Синтаксисом языка определены целые константы: десятичные, шестнадцатеричные и восьмеричные. Основание определяется префиксом в записи константы. Для десятичных констант префикс не используется. Десятичные целые определены как последовательности десятичных цифр, начинающиеся не с нуля (если это не число нуль):

44 684 0 1024

Последовательность цифр, начинающаяся с 0 и не содержащая десятичных цифр старше 7, воспринимается как восьмеричная константа:

016 – восьмеричное представление десятичного целого 14.

Последовательность шестнадцатеричных цифр (0, 1, ..., 9, A, B, C, D, E, F), перед которой записаны символы 0x или 0X, считается шестнадцатеричной константой:

0x16 – шестнадцатеричное представление десятичного целого 22;
0XFF – шестнадцатеричное представление десятичного целого 255.

Каждая конкретная реализация языка вводит свои ограничения на предельные значения констант. Например, компилятор Turbo C в отношении целых констант соответствует стандарту и допускает целые десятичные от 0 до 32767, а длинные целые (см. ниже тип **long**) – от 0 до 2147483647.

Вещественные константы. Для представления вещественных (нецелых) чисел используются константы, представляемые в памяти ЭВМ в форме с плавающей точкой. Каждая вещественная константа состоит из следующих частей: целая часть (десятичная целая константа); десятичная точка; дробная часть (десятичная целая константа); признак показателя «e» или «E»; показатель десятичной степени (десятичная целая константа, возможно, со знаком). При записи констант с плавающей точкой могут опускаться целая или дробная часть (но не одновременно); десятичная точка или символ экспоненты с показателем степени (но не одновременно). Примеры констант с плавающей точкой:

44. 3.14159 44e0 .314159E1 0.0

Предельные значения и типы арифметических констант. Машинное представление (код) программы на языке Си предполагает, что каждая константа, введенная в программе, занимает в ЭВМ некоторый участок памяти. Размеры этого участка памяти и интерпретация его содержимого определяются типом соответствующей константы. В приложении 2 приведены допустимые стандартом предельные значения для разных типов данных. Почти все компиляторы отводят символьным константам (символам) по одному байту (восемь бит). Тем самым вводится ограничение на все разнообразие символьных констант – их внутренние коды должны находиться в диапазоне от 0 до 255. В языках многих стран мира не весь набор символов (букв и знаков) может быть представлен с помощью одного байта. В настоящее время ведется систематическая международная работа по созданию многобайтового универсального кода (unicode), в рамках которого можно представлять символы почти всех алфавитов. Однако рассмотрение многобайтовых кодов в рамках настоящего пособия представляется нам преждевременным.

Для целых и вещественных констант каждая реализация компилятора с языка Си может определять свои ограничения. Кроме того, необходимо учитывать платформу (аппаратная и программная составляющие), в среде которой реализован компилятор с языка Си, и платформу, в среде которой будет эксплуатироваться созданный программный продукт (см. приложение 4). В табл. 1.1 приведены пределы, исходя из которых компиляторы, реализованные на современных персональных компьютерах (ПК), выбирают типы целых констант. Например, все целые константы в диапазоне от 0 до 32767 имеют тип **int**, то есть будут представлены в памяти участками в 2 байта (16 бит).

Таблица 1.1. Целые константы и выбираемые для них типы

Диапазоны значений констант			Тип данных
десятичные	**восьмеричные**	**шестнадцатеричные**	
от 0 до 32767	от 00 до 077777	от 0x0000 до 0x7FFF	int
–	от 0100000 до 0177777	от 0x8000 до 0xFFFF	unsigned int
от 32768 до 2147483647	от 020000 до 017777777777	от 0x10000 до 0x7FFFFFFF	long
от 2147483648 до 4294967295	от 020000000000 до 037777777777	от 0x80000000 до 0xFFFFFFFF	unsigned long
до 9223372036854775807	до 0377777777777777777777	до 0x7FFFFFFFFFFFFFFF	long long
> 9223372036854775807	>037777777777777777777	> 0x7FFFFFFFFFFFFFF	ошибка

В табл. 1.2 приведены сведения о представлении данных вещественных типов.

Таблица 1.2. Данные вещественных типов

Тип данных	Размер, бит	Диапазон абсолютных величин
float	32	от 3.4E–38 до 3.4E+38
double	64	от 1.7E–308 до 1.7E+308
long double	80	от 3.4E–4932 до 1.1E+4932

Вещественная константа 3.141592653589793 будет воспринята как имеющая тип **double**, и ей будет выделено 8 байт (64 бита). Тот же тип выбирается для константы 3.14, так как по умолчанию всем вещественным константам присваивается тип **double**.

Если программиста не устраивает тип, который компилятор приписывает константе, то тип можно явно указать в записи константы с помощью суффиксов: **F** (или **f**) – **float** (для вещественных), **U** (или **u**) – **unsigned** (для целых), **L** (или **l**) – **long** (для целых и вещественных). Например:

3.14159F – константа типа **float** (выделяется 4 байта);
3.14L – константа типа **long double** (выделяется 10 байт).

С помощью суффикса **U** (или **u**) можно представить целую константу в виде беззнакового целого. Например:

50000U – константа типа **unsigned int**.

Константе 50000U выделяются 2 байта (вместо четырех, как было бы при отсутствии суффикса (см. табл. 1.1)). В этом случае, то есть для **unsigned int**, знаковый бит используется для представления одного из разрядов кода числа, и диапазон значений становится от 0 до 65535.

Суффикс **L** (или **l**) позволяет выделить целой константе 4 байта (32 бита):

500L – константа типа **long**, которой выделяется 4 байта;
0L – целая константа типа **long** длиной 4 байта.

Совместное использование в любом порядке суффиксов **U** (или **u**) и **L** (или **l**) позволяет приписать целой константе тип **unsigned long,** и она займет в памяти 32 разряда (бита), причем знаковый разряд будет использоваться для представления разряда кода (а не знака). Примеры:

0LU – целая константа типа **unsigned long** длиной 4 байта;

2424242424UL – константа типа **unsigned long**;
123LL – целая константа типа **long long**.

Нулевой указатель. Null-указатель, называемый нулевым указателем, – это единственная неарифметическая константа. Ее роль и функциональные возможности станут ясны при изучении аппарата указателей (см. следующие главы). В конкретных реализациях null-указатель может быть представлен либо как 0, либо как 0L, либо как именованная константа **NULL**. Здесь нужно отметить, что значение константы **NULL** не обязано быть нулем и имеет право не совпадать с кодом символа '0'.

Константы перечисляемого типа. Целочисленные именованные константы можно вводить с помощью перечисления:

enum *тип_перечисления {список_именованных_констант}*,

где **enum** – служебное слово, вводящее перечисление; *тип_перечисления* – его название – необязательный произвольный идентификатор; *список_именованных_констант* – разделенная запятыми последовательность идентификаторов или именованных констант вида: *имя_константы=значение_константы.*

Примеры:

```
enum {ONE=1, TWO, THREE, FOUR};
enum DAY {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY};
enum BOOLEAN {NO, YES};
```

Если в списке нет ни одного элемента со знаком '=', то значения констант начинаются с 0 и увеличиваются на 1 слева направо. Таким образом, NO равно 0, YES равно 1, SUNDAY имеет значение 0, и FRIDAY имеет значение 5. Именованная константа со знаком '=' получает соответствующее значение (ONE=1), а следующие за ней именованные константы без явных значений увеличиваются на 1 каждая. В нашем примере TWO равно 2, THREE равно 3, FOUR равно 4.

Строки, или строковые константы. Формально строки (в соответствии со стандартом) не относятся к константам языка Си, а представляют собой отдельный тип его лексем. Для них в литературе используется еще одно название – «строковые литералы». Строковая константа определяется как последовательность символов (см. выше символьные константы), заключенная в двойные кавычки (не в апострофы):

```
"Образец строки"
```

Среди символов строки могут быть эскейп-последовательности, то есть сочетания знаков, соответствующие неизображаемым символам, или символам, задаваемым их внутренними кодами. В этом случае, как и в представлениях отдельных символьных констант, их изображения начинаются с обратной косой черты ' \' :

```
"\n Текст \n разместится \n в 3-х строках дисплея"
```

Представления строковых констант в памяти ЭВМ подчиняются следующим правилам. Все символы строки размещаются подряд, и каждый символ (в том числе представленный эскейп-последовательностью) занимает ровно 1 байт. В конце записи строковой константы компилятор помещает символ '\0'.

Таким образом, количество байтов, выделяемое в памяти ЭВМ для представления значения строки, ровно на 1 больше, чем число символов в записи этой строковой константы:

```
"Эта строка занимает в памяти ЭВМ 43 байта."
"Строка в 18 байт."
```

При работе с символьной информацией нужно помнить, что длина константы 'F' равна 1 байту, а длина строки "F" равна 2 байтам.

При записи строковых констант возможно размещение одной константы в нескольких строках текстового файла с программой. Для этого используется следующее правило.

Если в последовательности символов (литер) константы встречается литера '\', за которой до признака '\n' конца строки текстового файла размещены только пробелы, то эти пробелы вместе с символом '\' и окончанием '\n' удаляются, и продолжением строковой константы считается следующая строка текста. Например, следующий текст представляет одну строковую константу:

```
"Шалтай-Болтай \
     сидел на стене."
```

В программе эта константа будет эквивалентна такой:

```
"Шалтай-Болтай      сидел на стене."
```

Начальные (левые) пробелы в продолжении константы на новой строке не удаляются, а считаются входящими в строковую константу.

Две строковые константы, между которыми нет других разделителей, кроме обобщенных пробельных символов (пробел, табуляция, конец строки и т. д.), воспринимаются как одна строковая константа. Таким образом,

```
"Шалтай-Болтай" " свалился во сне."
```

Воспринимается как одна константа:

```
"Шалтай-Болтай свалился во сне."
```

Тем же правилам подчиняются и строковые константы, размещенные на разных строках. Как одна строка будет воспринята последовательность

```
"Вся королевская "
                          "конница, "
    "вся королевская "
                          "рать"
```

Эти четыре строковые константы эквивалентны одной:

```
"Вся королевская конница, вся королевская рать"
```

Обратите внимание, что в результирующую строку здесь не включаются начальные пробелы перед каждой константой-продолжением.

1.3. Переменные и именованные константы

Переменная как объект. Одним из основных понятий языка Си является объект – именованная область памяти. Частный случай объекта – переменная. Отличительная особенность переменной состоит в возможности связывать с ее именем различные значения, совокупность которых определяется типом переменной. При задании значения переменной в соответствующую ей область памяти помещается код этого значения. Доступ к значению переменной наи-

более естественно обеспечивает ее имя, а доступ к участку памяти возможен только по его адресу. О взаимосвязях имен и адресов будет подробно говориться в главе, посвященной указателям и работе с памятью ЭВМ. Для целей первых глав будет вполне достаточно интерпретировать понятие переменной как пару «имя – значение».

Определение переменных. Каждая переменная перед ее использованием в программе должна быть определена, то есть для переменной должна быть выделена память. Размер участка памяти, выделяемой для переменной, и интерпретация содержимого зависят от типа, указанного в определении переменной.

В соответствии с типами значений, допустимых в языке Си, рассмотрим символьные, целые и вещественные переменные автоматической памяти. О классах памяти (один из которых – класс автоматической памяти) будем подробно говорить позже. Сейчас достаточно ввести только переменные автоматической памяти, которые существуют в том блоке, где они определены. В наиболее распространенном случае таким блоком является текст основной (main) функции программы.

Простейшая форма определения переменных:

тип список_имен_переменных,

где *имена переменных* – это выбранные программистом идентификаторы, которые в списке разделяются запятыми; *тип* – один из уже упоминаемых (в связи с константами) типов.

Определены целочисленные типы (перечислены в порядке неубывания длины внутреннего представления):

- ❑ **char** – целый длиной не менее 8 бит;
- ❑ **short int** – короткий целый (допустима аббревиатура **short**);
- ❑ **int** – целый;
- ❑ **long** – длинный целый.

Каждый из целочисленных типов может быть определен либо как знаковый **signed**, либо как беззнаковый **unsigned** (по умолчанию **signed**).

Различие между этими двумя типами – в правилах интерпретации старшего бита внутреннего представления. Спецификатор **signed** требует, чтобы старший бит внутреннего представления воспринимался как знаковый; **unsigned** означает, что старший бит внутреннего представления входит в код представляемого числового значения, которое считается в этом случае беззнаковым. Выбор знакового или беззнакового представления определяет предельные

значения, которые можно представить с помощью описанной переменной. Например, на современном ПК переменная типа **unsigned int** позволяет представить числа от 0 до 65535, а переменной типа **signed int** (или просто **int**) соответствуют значения в диапазоне от –32768 до +32767. Чтобы глубже понять различие между целой величиной и целой величиной без знака, следует обратить внимание на результат выполнения унарной операции "–" (минус) над целой величиной и целой величиной без знака. Для целой величины результат очевиден и тривиален. Результатом при использовании целой величины без знака является

2^n – (*значение_величины_без_знака*),

где n – количество разрядов, отведенное для представления величины без знака.

По умолчанию при отсутствии в качестве префикса ключевого слова **unsigned** любой целый тип считается знаковым (**signed**). Таким образом, употребление совместно со служебными словами **char**, **short**, **int**, **long** префикса **signed** излишне. Допустимо отдельное использование обозначений (спецификаторов) «знаковости». При этом

signed эквивалентно **signed int**;
unsigned эквивалентно **unsigned int**.

Примеры определений целочисленных переменных:

```
char symbol, cc;
unsigned char code;
int number, row;
unsigned long long_number;
```

Обратите внимание на необходимость символа «точка с запятой» в конце каждого определения.

Стандартом языка введены следующие вещественные типы:

- **float** – вещественный одинарной точности;
- **double** – вещественный удвоенной точности;
- **long double** – вещественный максимальной точности.

Значения всех вещественных типов в ЭВМ представляются с «плавающей точкой», то есть с мантиссой и порядком, как было рассмотрено при определении констант (§1.2). Примеры определений вещественных переменных:

```
float x, X, cc3, pot_8;
double e, Stop, B4;
```

Предельные значения переменных. Предельные значения констант (и соответствующих переменных) разработчики компиляторов вправе выбирать самостоятельно, исходя из аппаратных возможностей компьютера. Однако при такой свободе выбора стандарт языка требует, чтобы для значений типа **short** и **int** было отведено не менее 16 бит, для **long** – не менее 32 бит. При этом размер **long** должен быть не менее размера **int**, а **int** – не менее **short**. Предельные значения арифметических констант и переменных для большинства компиляторов, реализованных на современных ПК, приведены в табл. 1.3.

Таблица 1.3. Основные типы данных

Тип данных	Размер, бит	Диапазон значений
unsigned char	8	0 … 255
char	8	–128 … 127
enum	16	–32768 … 32767
unsigned int	16	0 … 65535
short int (short)	16	–32768 … 32767
unsigned short	16	0 … 65535
int	16	–32768 … 32767
unsigned long	32	0 … 4294967295
long	32	–2147483648 … 2147483647
long long	64	–9223372036854775808 … 9223372036854775807
float	32	3.4E–38 … 3.4E+38
double	64	1.7E–308 … 1.7E+308
long double	80	3.4E–4932 … 1.1E+4932

Предельные значения вещественных переменных совпадают с предельными значениями соответствующих констант (см., например, табл. 1.2).

Предельные значения целочисленных переменных совпадают с предельными значениями соответствующих констант (см. табл. 1.1). Таблица 1.3 содержит и предельные значения для тех типов, которые не включены в табл. 1.1.

Требования стандарта отображают таблицы приложения 2.

Инициализация переменных. В соответствии с синтаксисом языка переменные автоматической памяти после определения по умол-

чанию имеют неопределенные значения. Надеяться на то, что они равны, например, 0, нельзя. Однако переменным можно присваивать начальные значения, явно указывая их в определениях:

тип имя_ переменной=начальное_значение;

Этот прием назван ***инициализацией***. В отличие от присваивания, которое осуществляется в процессе выполнения программы, инициализация выполняется при выделении для переменной участка памяти. Примеры определений с инициализацией:

```
float pi=3.1415, cc=1.23;
unsigned int year=1997;
```

Именованные константы. В языке Си, кроме переменных, могут быть определены константы, имеющие фиксированные названия (имена). В качестве имен констант используются произвольно выбираемые программистом идентификаторы, не совпадающие с ключевыми словами и с другими именами объектов. Традиционно принято, что для обозначений констант выбирают идентификаторы из больших букв латинского алфавита и символов подчеркивания. Такое соглашение позволяет при просмотре большого текста программы на языке Си легко отличать имена переменных от названий констант.

Первая возможность определения именованных констант была проиллюстрирована в §1.2, посвященном константам. Это перечисляемые константы, вводимые с использованием служебного слова **enum**.

Вторую возможность вводить именованные константы обеспечивают определения такого вида:

const *тип имя_константы=значение_константы;*

Здесь **const** – квалификатор типа, указывающий, что определяемый объект имеет постоянное значение, то есть доступен только для чтения; *тип* – один из типов объектов; *имя_константы* – идентификатор; *значение_константы* должно соответствовать ее типу. Примеры:

```
const double E=2.718282;
const long M=99999999;
const F=765;
```

В последнем определении тип константы не указан, по умолчанию ей приписывается тип **int**.

Третью возможность вводить именованные константы обеспечивает препроцессорная директива

#define *имя_константы значение_константы*

Обратите внимание на отсутствие символа «точка с запятой» в конце директивы. Подробному рассмотрению директивы **#define** будут посвящены два параграфа главы 3. Здесь мы только упоминаем о возможности с ее помощью определять именованные константы. Кроме того, отметим, что в конкретные реализации компиляторов с помощью директив **#define** включают целый набор именованных констант с фиксированными именами (см. главу 3 и приложение 2).

Отличие определения именованной константы

```
const double E=2.718282;
```

от определения препроцессорной константы с таким же значением

```
#define EULER 2.718282
```

состоит внешне в том, что в определении константы E явно задается ее тип, а при препроцессорном определении константы EULER ее тип определяется «внешним видом» значения константы. Например, следующее определение

```
#define NEXT 'Z'
```

вводит обозначение NEXT для символьной константы 'Z'. Это соответствует такому определению:

```
const char NEXT = 'Z';
```

Однако различия между обычной именованной константой и препроцессорной константой, вводимой директивой **#define**, гораздо глубже и принципиальнее. До начала компиляции текст программы на языке Си обрабатывается специальным компонентом транслятора – препроцессором. Если в тексте встречается директива

```
#define EULER 2.718282
```

а ниже ее в тексте используется имя константы EULER, например в таком виде:

```
double mix = EULER;
d = alfa*EULER;
```

то препроцессор заменит каждое обозначение EULER на ее значение и сформирует такой текст:

```
double mix = 2.718282;
d = alfa*2.718282;
```

Далее текст от препроцессора поступает к компилятору, который уже «и не вспомнит» о существовании имени EULER, использованного в препроцессорной директиве **#define**. Константы, определяемые на препроцессорном уровне с помощью директивы **#define**, очень часто используются для задания размеров массивов, что будет продемонстрировано позже.

Итак, основное отличие констант, определяемых препроцессорными директивами **#define**, состоит в том, что эти константы вводятся в текст программы до этапа ее компиляции. Специальный компонент транслятора – препроцессор – обрабатывает исходный текст программы, подготовленный программистом, и делает в этом тексте замены и подстановки. Пусть в исходном тексте встречается директива:

```
#define ZERO 0.0
```

Это означает, что каждое последующее использование в тексте программы имени ZERO будет заменяться на 0.0.

Рисунок 1.1 иллюстрирует некоторые принципы работы препроцессора. Его основное отличие от других компонентов транслятора – обработка программы выполняется только на уровне ее текста. На входе препроцессора – текст с препроцессорными директивами, на выходе препроцессора – модифицированный текст без препроцессорных директив. Этот выходной модифицированный текст изменен по сравнению с входным текстом за счет выполнения препроцессорных директив, но сами препроцессорные директивы в выходном тексте отсутствуют. Полностью все препроцессорные директивы будут рассмотрены позже в главе 3. В связи с именованными конс-

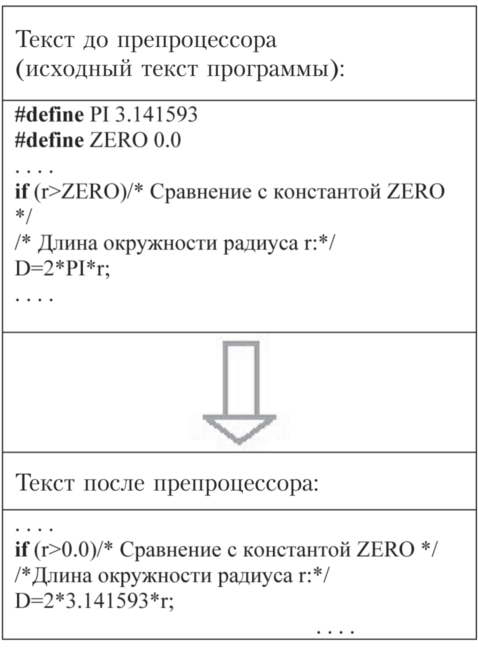


Рис. 1.1. *Обработка текста программы препроцессором*

тантами здесь рассматривается только одна из возможностей директивы **#define** – простая подстановка.

Имена PI и ZERO (см. рис. 1.1) после работы препроцессора заменены в тексте программы на определенные в двух директивах **#define** значения (3.141593 и 0.0).

Обратите внимание, что подстановка не выполняется в комментариях и в строковых константах. В примере на рис. 1.1 идентификатор ZERO остался без изменений в комментарии (/* Сравнение с константой ZERO */).

Именно с помощью набора именованных препроцессорных констант стандарт языка Си рекомендует авторам компиляторов определять предельные значения всех основных типов данных. Для этого в языке определен набор фиксированных имен, каждое из которых является именем одной из констант, определяющих то или иное предельное значение. Например:

- **FLT_MAX** – максимальное число с плавающей точкой типа **float**;

- **CHAR_BIT** – количество битов в байте;
- **INT_MIN** – минимальное значение для данных типа **int**.

Общий список стандартных препроцессорных именованных констант для арифметических данных дан в приложении 2.

Чтобы использовать в программе указанные именованные препроцессорные константы, недостаточно записать их имена в программе. Предварительно в текст программы необходимо включить препроцессорную директиву такого вида:

#include <*имя_заголовочного_файла*>,

где в качестве *имени_заголовочного_файла* подставляются:
limits.h – для данных целых типов;
float.h – для вещественных данных.

В заголовочный файл **limits.h** авторы компилятора поместили набор препроцессорных директив, среди которых есть такие (приведены значения в шестнадцатеричном виде для Turbo C):

```
#define  CHAR_BIT  8
#define  SHRT_MAX  0x7FFF
#define  LONG_MAX  0x7FFFFFFFL
.  .  .  .
```

В заголовочном файле **float.h** находятся директивы, определяющие константы, связанные с представлением данных вещественных типов. Например:

```
#define  FLT_MIN  1.17549435E-38F
#define  DBL_MIN  2.2250738585072014E-308
#define  DBL_EPSILON  2.2204460492503131E-16
.  .  .  .
```

Значения этих предопределенных на препроцессорном уровне констант в соответствии со стандартом языка в конкретных компиляторах могут быть несколько иными, отличными от тех, что приведены в таблицах приложения 2.

Подробно о возможностях препроцессорных средств будет говориться в главе 3. Сейчас достаточно знать, что, записав в тексте своей программы директиву

#include <limits.h>

можно использовать в программе стандартные именованные константы **CHAR_BIT**, **SHRT_MIN** и т. д. (см. приложение 2), а уж их

значениями будут те числа, которые включили в директивы **#define** авторы конкретного компилятора и конкретной библиотеки.

Если включить в программу директиву

#include <float.h>

то станут доступными именованные константы предельных значений числовых данных вещественных типов (см. приложение 2).

Такой подход к определению предельных значений с помощью препроцессорных констант, сохраняемых в библиотечных файлах, позволяет писать программы, не зависящие от реализации, что обеспечивает их достаточную мобильность. Программист использует в программе стандартные имена (обозначения) констант, а их значения определяются версией реализации, то есть конкретным компилятором и его библиотеками.

1.4. Операции

В этом параграфе достаточно лаконично определяются все операции языка Си и приводится таблица их приоритетов. Хотя в ближайших параграфах (где вводятся выражения и основные операторы языка) нам понадобятся не все знаки операций и нередко будет достаточно интуитивного представления об их старшинстве и ассоциативности, но материал этого параграфа слишком важен, чтобы говорить о нем позже. В последующих главах применение операций языка будет рассмотрено подробно, но к таблице приоритетов стоит обращаться и при дальнейшем чтении, и при написании программ.

Знаки операций. Для формирования и последующего вычисления выражений используются операции. Для изображения одной операции в большинстве случаев используется несколько символов. В табл. 1.4 приведены все знаки операций, определенные стандартом языка. Операции в таблице разбиты на группы в соответствии с их рангами.

За исключением операций "**[ ]**", "**( )**" и "**?:**", все знаки операций распознаются компилятором как отдельные лексемы. В зависимости от контекста одна и та же лексема может обозначать разные операции, то есть один и тот же знак операции может употребляться в различных выражениях и по-разному интерпретироваться в зависимости от контекста. Например, бинарная операция **&** – это поразрядная конъюнкция, а унарная операция **&** – это операция получения адреса.

Операции ранга 1 имеют наивысший приоритет. Операции одного ранга имеют одинаковый приоритет, и если их в выражении несколько, то они выполняются в соответствии с правилом ассоциативности либо слева направо (→), либо справа налево (←). Если один и тот же знак операции приведен в таблице дважды (например, знак *), то первое появление (с меньшим по номеру, то есть старшим по приоритету, рангом) соответствует унарной операции, а второе – бинарной.

Опишем кратко возможности отдельных операций.

Таблица 1.4. Приоритеты (ранги) операций

Ранг	Операции	Ассоциативность
1	( ) [ ] -> .	→
2	! ~ + - ++ -- & * (*тип*) sizeof	←
3	* / % (мультипликативные бинарные)	→
4	+ - (аддитивные бинарные)	→
5	<< >> (поразрядного сдвига)	→
6	< <= >= > (отношения)	→
7	== != (отношения)	→
8	& (поразрядная конъюнкция «И»)	→
9	^ (поразрядное исключающее «ИЛИ»)	→
10	| (поразрядная дизъюнкция «ИЛИ»)	→
11	&& (конъюнкция «И»)	→
12	|| (дизъюнкция «ИЛИ»)	→
13	?: (тернарная операция)	←
14	= *= /= %= += -= &= ^= |= <<= >>=	←
15	, (операция «запятая»)	→

Унарные (одноместные) операции. Для изображения одноместных префиксных и постфиксных операций используются следующие символы:

- **&** – операция получения адреса операнда (*ранг 2*);
- ***** – операция обращения по адресу, то есть раскрытия ссылки, иначе операция *разыменования* (доступа по адресу к значению того объекта, на который указывает операнд). Операндом должен быть указатель (*ранг 2*);
- **–** – унарный минус, изменяет знак арифметического операнда (*ранг 2*);
- **+** – унарный плюс, введен для симметрии с унарным минусом (*ранг 2*);

- **~** – поразрядное инвертирование внутреннего двоичного кода целочисленного аргумента – побитовое отрицание (*ранг 2*);
- **!** – логическое отрицание (НЕ) значения операнда (*ранг 2*). Применяется к скалярным операндам. Целочисленный результат **0** (если операнд ненулевой, то есть истинный) или **1** (если операнд нулевой, то есть ложный). Напомним, что в качестве логических значений в языке используют целые числа: **0** – ложь и не нуль, то есть **(!0)** – истина. Отрицанием любого ненулевого числа будет **0**, а отрицанием нуля будет **1**. Таким образом: **!1** равно **0**; **!2** равно **0**; **!(-5)** равно **0**; **!0** равно **1**;
- **++** – увеличение на единицу (инкремент или автоувеличение – *ранг 2*); имеет две формы:
  - *префиксная операция* – увеличение значения операнда на 1 до его использования. Ассоциативность справа в соответствии со стандартом;
  - *постфиксная операция* – увеличение значения операнда на 1 после его использования. Ассоциативность слева в соответствии со стандартом.

  Операнд для операции **++** (и для операции **--**) не может быть константой либо произвольным выражением. Записи **++5** или **84++** будут неверными. **++(j+k)** – также неверная запись. Операндами унарных операций **++** и **--** должны быть всегда модифицируемые именующие выражения (*L-value*, *left value*, *l*-значение, леводопустимое выражение). Термины «леводопустимое выражение» и «*l*-значение» происходят от объяснения действия операции присваивания **E = D**, в которой операнд **E** слева от знака операции присваивания может быть только модифицируемым *l*-значением. Примером модифицируемого *l*-значения служит имя переменной, которой выделена память. Таким образом, *l*-значение – ссылка на область памяти, значение которой доступно изменениям;
- **--** – уменьшение на единицу (декремент или автоуменьшение – *ранг 2*) – унарная операция, операндом которой должно быть леводопустимое выражение, то есть не константа и не выражение:
  - *префиксная операция* – уменьшение на 1 значения операнда до его использования;
  - *постфиксная операция* – уменьшение на 1 значения операнда после его использования;

- **sizeof** – операция (*ранг 2*) вычисления размера (в байтах) для объекта того типа, который имеет операнд. Разрешены два формата операции:
  - **sizeof** *выражение*;
  - **sizeof***(тип)*;

  sizeof не вычисляет значения выражения, а только определяет его тип, для которого затем вычисляется размер.

Бинарные (двуместные) операции делятся на следующие группы:

- аддитивные;
- мультипликативные;
- сдвигов;
- поразрядные;
- операции отношений;
- логические;
- присваивания;
- выбора компонента структурированного объекта;
- операция «запятая»;
- скобки в качестве операций.

Аддитивные операции:

- **+** – бинарный плюс – сложение арифметических операндов или сложение указателя с целочисленным операндом (*ранг 4*);
- **–** – бинарный минус – вычитание арифметических операндов или вычитание указателей (*ранг 4*).

Мультипликативные операции:

- * – умножение операндов арифметического типа (*ранг 3*);
- / – деление операндов арифметического типа (*ранг 3*). При целочисленных операндах абсолютное значение результата округляется до целого. Например, **20/3** равно **6**, **–20/3** равно **–6**, **(–20)/3** равно **–6**, **20/(–3)** равно **–6**;
- % – получение остатка от деления целочисленных операндов (деление по модулю – *ранг 3*). При неотрицательных операндах остаток положительный. В противном случае остаток определяется реализацией. Например:
  - **13%4** равняется **1**, **(–13)%4** равняется **–1**;
  - **13%(–4)** равно **+1**, а **(–13)%(–4)** равняется **–1**.

  При ненулевом делителе для целочисленных операндов всегда выполняется соотношение: **(a/b)*b + a%b** равно **a**.

Операции сдвига (определены только для целочисленных операндов). Формат выражения с операцией сдвига:

операнд_левый операция_сдвига операнд_правый

<< – сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда (*ранг 5*);

>> – сдвиг вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда (*ранг 5*).

Поразрядные операции:

- **&** – поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов (*ранг 8*);
- | – поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов (*ранг 10*);
- ^ – поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов (*ранг 9*).

Результат выполнения операций сдвига и поразрядных операций:

- 4<<2 равняется 16;
- 5>>1 равняется 2;
- 6&5 равняется 4;
- 6 | 5 равняется 7;
- 6^5 равняется 3.

Напоминаем, что двоичный код для 4 равен 100, для 5 – это 101, для 6 – это 110 и т. д. При сдвиге влево на две позиции код 100 становится равным 10000 (десятичное значение равно 16). Остальные результаты операций сдвига и поразрядных операций могут быть прослежены аналогично.

Обратите внимание, что сдвиг влево на n позиций эквивалентен умножению значения на 2^n, а сдвиг кода вправо уменьшает соответствующее значение в 2^n раз с отбрасыванием дробной части результата. (Поэтому **5>>1** равно 2.)

Выражения с поразрядными операциями. Поразрядные операции позволяют конструировать выражения, в которых обработка операндов выполняется на битовом уровне (поразрядно).

Операции отношений (сравнения):

< меньше, чем (*ранг 6*);
> больше, чем (*ранг 6*);
<= меньше или равно (*ранг 6*);
>= больше или равно (*ранг 6*);
== равно (*ранг 7*);
!= не равно (*ранг 7*).

Операнды операций отношений должны быть арифметического типа или могут быть указателями. Результат целочисленный: 0 (ложь) или 1 (истина). Последние две операции (операции сравнения на равенство) имеют более низкий приоритет по сравнению с остальными операциями отношений. Таким образом, выражение **(x < B == A < x)** есть 1 тогда и только тогда, когда значение **x** находится в интервале от **A** до **B** и **A<B**. (Вначале вычисляются **x < B** и **A < x**, а к результатам применяется операция сравнения на равенство **==**.)

***Логические бинарные операции*:**

- && – конъюнкция (И) арифметических операндов или отношений (*ранг 11*). Целочисленный результат 0 (ложь) или 1 (истина);
- || – дизъюнкция (ИЛИ) арифметических операндов или отношений (*ранг 12*). Целочисленный результат 0 (ложь) или 1 (истина).

  (Вспомните о существовании унарной операции отрицания '**!**'.)

Результаты отношений и логических операций:

- 3<5 равняется 1;
- 3>5 равняется 0;
- 3==5 равняется 0;
- 3!=5 равняется 1;
- 3!=5 || 3==5 равняется 1;
- 3+4>5 && 3+5>4 && 4+5>3 равняется 1.

Операции присваивания (ранг 14)

В качестве левого операнда в операциях присваивания может использоваться только модифицируемое именующее выражение (*l*-значение), то есть ссылка на некоторую именованную область памяти, значение которой доступно изменениям.

Перечислим операции присваивания, отметив, что существуют одна простая операция присваивания и ряд составных операций:

- = – *простое присваивание*: присвоить значение выражения-операнда из правой части операнду левой части. Пример: **P = = 10.3 – 2*x**;
- *= – *присваивание после умножения*: присвоить операнду левой части произведение значений обоих операндов. **P *= 2** эквивалентно **P = P * 2**;
- /= – *присваивание после деления*: присвоить операнду левой части частное от деления значения левого операнда на значение правого. **P /= 2.2 – d** эквивалентно **P = P / (2.2 – d)**;

- %= – *присваивание после деления по модулю*: присвоить операнду левой части остаток от целочисленного деления значения левого операнда на значение правого операнда. **N %= 3** эквивалентно **N = N % 3**;
- += – *присваивание после суммирования*: присвоить операнду левой части сумму значений обоих операндов. **A += B** эквивалентно **A = A + B**;
- –= – *присваивание после вычитания*: присвоить операнду левой части разность значений левого и правого операндов. **X –= 4.3 – Z** эквивалентно **X = X – (4.3 – Z)**;
- <<= – *присваивание после сдвигов разрядов влево*: присвоить целочисленному операнду левой части значение, полученное сдвигом влево его битового представления на количество разрядов, равное значению правого целочисленного операнда. Например, **a <<= 4** эквивалентно **a = a << 4**;
- >>= – *присваивание после сдвигов разрядов вправо*: присвоить целочисленному операнду левой части значение, полученное сдвигом вправо его битового представления на количество разрядов, равное значению правого целочисленного операнда. Например, **a >>= 4** эквивалентно **a = a >> 4**;
- &= – *присваивание после поразрядной конъюнкции*: присвоить целочисленному операнду левой части значение, полученное поразрядной конъюнкцией (И) его битового представления с битовым представлением целочисленного операнда правой части. **e &= 44** эквивалентно **e = e & 44**;
- |= – *присваивание после поразрядной дизъюнкции*: присвоить целочисленному операнду левой части значение, полученное поразрядной дизъюнкцией (ИЛИ) его битового представления с битовым представлением целочисленного операнда правой части. **a |= b** эквивалентно **a = a | b**;
- ^= – *присваивание после исключающего поразрядного «ИЛИ»*: присвоить целочисленному операнду левой части значение, полученное применением поразрядной операции исключающего ИЛИ к битовым представлениям значений обоих операндов. **z ^= x + y** эквивалентно **z = z ^ (x + y)**.

Обратите внимание, что для всех составных операций присваивания форма присваивания **E1 op= E2** эквивалентна **E1 = E1 op (E2)**, где **op** – обозначение операции.

Операции выбора компонентов структурированного объекта:

- **.** (точка) – прямой выбор (выделение) компонента структурированного объекта, например объединения или структуры (*ранг 1*). Формат применения операции:
  имя_структурированного_объекта **.** *имя_компонента*
- **–>** – косвенный выбор (выделение) компонента структурированного объекта, адресуемого указателем (*ранг 1*). При использовании операции требуется, чтобы с объектом был связан указатель (указателям посвящена глава 4). В этом случае формат применения операции имеет вид:
  указатель_на_структурированный_объект **–>** *имя_компонента*

Так как операции выбора компонентов структурированных объектов используются со структурами и объединениями, то необходимые пояснения и примеры приведем позже, введя перечисленные понятия и, кроме того, аккуратно определив указатели.

Запятая в качестве операции (ранг 15)

Несколько выражений, разделенных запятыми «**,**», вычисляются последовательно слева направо. В качестве результата сохраняются тип и значение самого правого выражения. Таким образом, операция «запятая» группирует вычисления слева направо. Тип и значение результата определяются самым правым из разделенных запятыми операндов (выражений). Значения всех левых операндов игнорируются. Например, если переменная x имеет тип **int**, то значением выражения (x=3, 3*x) будет 9, а переменная x примет значение 3.

Скобки в качестве операций

Круглые **( )** *и квадратные* **[ ]** *скобки* играют роль бинарных операций (*ранг 1*) при вызове функций и индексировании элементов массивов. Для программиста, начинающего использовать язык Си, мысль о том, что скобки в ряде случаев являются бинарными операциями, часто даже не приходит в голову. И это даже тогда, когда он практически в каждой программе обращается к функциям или применяет индексированные переменные. Итак, отметим, что скобки могут служить бинарными операциями, особенности и возможности которых достойны внимания.

Круглые скобки обязательны в обращении к функции:

имя_функции(список_аргументов),

где операндами служат *имя_функции* и *список_аргументов*. Результат вызова определяется (вычисляется) в теле функции, структуру которого задает ее определение.

В выражении

имя_массива[*индекс*]

операндами для операции [ ] служат *имя_массива* и *индекс*. Подробнее с индексированными переменными мы познакомимся на примерах в главе 2 и более подробно в следующих главах.

Тернарная (условная трехместная) операция (ранг 13). В отличие от унарных и бинарных операций, *тернарная операция* используется с тремя операндами. В изображении условной операции применяются два символа **'?'** и **':'** и три выражения-операнда:

выражение_1 ***?*** *выражение_ 2* ***:*** *выражение_3*

Первым вычисляется значение *выражения_1*. Если оно истинно, то есть не равно нулю, то вычисляется значение *выражения_2*, которое становится результатом. Если при вычислении *выражения_1* получится 0, то в качестве результата берется значение *выражения_3*. Классический пример:

x < 0 ? –x : x;

Выражение возвращает абсолютную величину переменной **x**.

Операция явного преобразования типа. Операция преобразования (приведения) типа (*ранг 2*) имеет следующий формат:

(имя_типа) операнд

Такое выражение позволяет преобразовывать значение операнда к заданному типу. В качестве операнда используется унарное выражение, которое в простейшем случае может быть переменной, константой или любым выражением, заключенным в круглые скобки. Например, преобразования **(long)8** (внутреннее представление результата имеет длину 4 байта) и **(char)8** (внутреннее представление результата имеет длину 1 байт) изменяют длину внутреннего представления целых констант, не меняя их значений.

В этих преобразованиях константа не меняла значения и оставалась целочисленной. Однако возможны более глубокие преобразования, например **(long double)6** или **(float)4** не только изменяют длину константы, но и структуру ее внутреннего представления. В результатах будут выделены порядок и мантисса, значения будут вещественными.

Примеры:

```
long i = 12L;      /* Определение переменной */
float brig;         /* Определение переменной */
brig = (float)i;   /* Явное приведение типа */
```

brig получает значение 12L, преобразованное к типу **float**.

Преобразования типов арифметических данных нужно применять аккуратно, так как возможно изменение числовых значений. При преобразовании больших целочисленных констант к вещественному типу (например, к типу **float**) возможна потеря значащих цифр (потеря точности). Если вещественное значение преобразуется к целому, то возможна ошибка при выходе полученного значения за диапазон допустимых значений для целых. В этом случае результат преобразования не всегда предсказуем и целиком зависит от реализации.

1.5. Разделители

Этот параграф может быть опущен при первом чтении, так как смысл почти всех разделителей становится очевиден при разборе той или иной конструкции языка. Однако полнота изложения сведений о лексемах и их назначениях требует систематического рассмотрения разделителей именно здесь, что мы и делаем. В дальнейшем этот раздел можно использовать для справок. В некоторых примерах данного параграфа пришлось использовать понятия, вводимые в следующих главах (например, структурный тип или прототип функции).

Разделители, или *знаки пунктуации*, входят в число лексем языка:

[ ]   ( )   { }   ,   ;   :   ...   *   =   #

Квадратные скобки. Для ограничения индексов одно- и многомерных массивов используются квадратные скобки [ ]. Примеры:

int A[5]; A – одномерный массив из пяти элементов;
int x, e[3][2]; e – двумерный массив (матрица) размером 3×2.

Круглые скобки. Назначение круглых скобок ( ):

1) выделяют выражения-условия (в операторе «если»):

```
if (x < 0) x = -x;
/*абсолютная величина арифметической переменной*/
```

2) входят как обязательные элементы в определение и описание (в прототип) любой функции, где выделяют соответственно список параметров и список спецификаций параметров:

```
float F(float x, int k) /* Определение функции*/
{ тело_функции }
float F(float, int); /* Описание функции - ее прототип */
```

3) круглые скобки обязательны при определении указателя на функцию:

```
int (*pfunc)( ); /* Определение указателя pfunc на функцию */
```

4) группируют выражения, изменяя естественную последовательность выполнения операций:

```
y = (a + b) / c; /* Изменение приоритета операций */
```

5) входят как обязательные элементы в операторы циклов:

```
for (i=0, j=1; i<j; i+=2, j++) тело_цикла;
while ( i<j ) тело_цикла;
do тело_цикла while ( k>0 );
```

6) в макроопределениях настоятельно рекомендуется применение круглых скобок, обрабатываемых препроцессором.

Фигурные скобки. Для обозначения соответственно начала и конца составного оператора или блока используют фигурные скобки { }. Пример использования составного оператора в условном операторе:

```
if (d > x) { d--; x++; }
```

Пример блока – тело любой функции:

```
float absx (float x)
{
 return x>0.0?x:-x;
}
```

Обратите внимание на отсутствие точки с запятой после закрывающейся скобки **'}'**, обозначающей конец составного оператора или блока.

Фигурные скобки используются для выделения списка компонентов в определениях структурных и объединяющих типов:

```
/* Определение структурного типа cell: */
struct cell
{
   char *b;
   int ee;
   double U[6];
};
/* Определение объединяющего типа mix: */
union mix
{
   unsigned int ii;
   char cc[2];
};
```

Обратите внимание на необходимость точки с запятой после определения каждого типа.

Фигурные скобки используются при инициализации массивов и структур при их определении:

```
/* Инициализация массива: */
int month [ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
/* Инициализация структуры stock типа mixture */
struct mixture
{
   int ii;
   double dd;
   char cc; }
stock = { 666, 3.67, '\t' };
```

В примере mixture – имя структурного типа с тремя компонентами разных типов, stock – имя конкретной структуры типа mixture. Компоненты ii, dd, cc структуры stock получают значения при инициализации из списка в фигурных скобках. (Подробно о структурах см. в главе 6.)

Запятая. Запятая может быть использована в качестве операции, а может применяться как разделитель. В последнем случае она разделяет элементы списков. Списками определяют начальные значения элементов массивов и компонентов структур при их инициализации (примеры только что даны).

Другой пример списков – списки параметров аргументов в функциях.

Кроме того, запятая используется в качестве разделителя в заголовке оператора цикла:

```
for (x=p1,y=p2,i=2; i<n; z=x+y, x=y, y=z, i++);
```

(В данном примере после выполнения цикла значением переменной z будет величина, равная *n*-му члену последовательности чисел Фибоначчи, определенной по значениям первых двух **p1** и **p2**.)

Запятая как разделитель используется также в описаниях и определениях объектов (например, переменных) одного типа:

```
int i, n;
float x, y, z, p1, p2;
```

Следует обратить внимание на необходимость с помощью круглых скобок отделять запятую-операцию от запятой-разделителя. Например, для элементов следующего массива m используется список с тремя начальными значениями:

```
int i=1, m[ ]={ i, (i=2,i*i), i };
```

В данном примере запятая в круглых скобках выступает в роли знака операции. Операция присваивания «**=**» имеет более высокий приоритет, чем операция «запятая». Поэтому вначале **i** получает значение 2, затем вычисляется произведение **i*i**, и этот результат служит значением выражения в скобках. Однако значением переменной **i** остается 2. Значениями m[0], m[1], m[2] будут соответственно 1, 4, 2.

Точка с запятой. Каждый оператор, каждое определение и каждое описание в программе на языке Си завершает точка с запятой '**;**'. Любое допустимое выражение, за которым следует '**;**', воспринимается как оператор. Это справедливо и для пустого выражения, то есть отдельный символ «точка с запятой» считается пустым оператором. Пустой оператор иногда используется как тело цикла. Примером может служить цикл **for**, приведенный выше для иллюстрации особенностей использования запятой в качестве разделителя. (Вычисляется *n*-й член последовательности чисел Фибоначчи.)

Примеры операторов-выражений:

```
i++;     /* Результат - только изменение значения переменной i */
F(z,4);  /* Результат определяется телом функции с именем F */
```

Двоеточие. Для отделения метки от помечаемого ею оператора используется двоеточие '**:**':

метка: оператор;

Многоточие. Это три точки '**...**' без пробелов между ними. Оно используется для обозначения переменного числа аргументов у функции при ее определении и описании (при задании ее прототипа). При работе на языке Си программист постоянно использует библиотечные функции со списком аргументов переменной длины для форматных ввода и вывода. Их прототипы выглядят следующим образом:

```
int printf(char * format, ...);
int scanf (char * format, ...);
```

Здесь с помощью многоточия указана возможность при обращении к функциям использовать разное количество аргументов (не меньше одного, так как аргумент, заменяющий параметр **format**, должен быть указан всегда и не может опускаться).

Подготовка своих функций с переменным количеством аргументов на языке Си требует применения средств адресной арифметики, например макросов, предоставляемых заголовочным файлом **stdarg.h**. О возможностях упомянутых макросов подробно говорится в главе 5.

Звездочка. Как уже упоминалось, звездочка '*****' используется в качестве знака операции умножения и знака операции разыменования (получения доступа через указатель). В описаниях и определениях звездочка означает, что описывается (определяется) указатель на значение использованного в объявлении типа:

```
/*Указатель на величину типа int*/
int * point;
/* Указатель на указатель на объект типа char */
char ** refer;
```

Обозначение присваивания. Как уже упоминалось, для обозначения операции присваивания используется символ '**=**'. Кроме того, в определении объекта он используется при его инициализации:

```
/* инициализация структуры */
struct {char x, int y} A={ 'z', 1918 };
/* инициализация переменной */
int F = 66;
```

Признак препроцессорных директив. *Символ* '#' (знак номера или диеза в музыке) используется для обозначения директив (команд) препроцессора. Если этот символ является первым отличным от пробела символом в строке программы, то строка воспринимается как директива препроцессора. Этот же символ используется в качестве одной из препроцессорных операций (см. главу 3).

Без одной из препроцессорных директив обойтись практически невозможно. Это директива

#include <stdio.h>

которая включает в текст программы средства связи с библиотечными функциями ввода-вывода.

1.6. Выражения

Введя константы, переменные, разделители и знаки операций, охарактеризовав основные типы данных и рассмотрев переменные, можно конструировать выражения. Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей, в качестве которых чаще всего выступают круглые скобки ( ). Назначение любого выражения – формирование некоторого значения. В зависимости от типа формируемых значений определяются типы выражений. Если значениями выражения являются целые и вещественные числа, то говорят об арифметических выражениях.

Арифметические выражения. В арифметических выражениях допустимы следующие операции:

- + – сложение (или унарная операция +);
- – – вычитание (или унарная операция изменения знака);
- * – умножение;
- / – деление;
- % – деление по модулю (то есть получение остатка от целочисленного деления первого операнда на второй).

Операндами для перечисленных операций служат константы и переменные арифметические типы, а также выражения, заключенные в круглые скобки.

Примеры выражений с двумя операндами:

```
a+b         12.3-x         3.14159*Z         k/3         16%i
```

Нужно быть аккуратным, применяя операцию деления '/' к целочисленным операндам. Например, как мы уже упоминали выше, за

счет округления результата значением выражения 5/3 будет 1, а соответствует ли это замыслам программиста, зависит от смысла той конкретной конструкции, в которой это выражение используется.

Чтобы результат выполнения арифметической операции был вещественным, необходимо, чтобы вещественным был хотя бы один из операндов. Например, значением выражения 5.0/2 будет 2.5, что соответствует смыслу обычного деления.

Операции *****, **/**, **%** (см. табл. 1.4) имеют один ранг (3), операции **+**, **-** также ранг (4), но более низкий. Арифметические операции одного ранга выполняются слева направо. Для изменения порядка выполнения операций обычным образом используются скобки. Например, выражение (d+b)/2.0 позволяет получить среднее арифметическое операндов d и b.

Как уже говорилось, введены специфические унарные операции ++ (инкремент) и –– (декремент) для изменения на 1 операнда, который в простейшем случае должен быть переменной (леводопустимым значением). Каждая из этих операций может быть префиксной и постфиксной:

- выражение **++**m увеличивает на 1 значение m, и это полученное значение используется как значение выражения ++m (префиксная форма);
- выражение ––k уменьшает на 1 значение k, и это новое значение используется как значение выражения ––k (префиксная форма);
- выражение i**++** (постфиксная форма) увеличивает на 1 значение i, однако значением выражения i++ является предыдущее значение i (до его увеличения);
- выражение j–– (постфиксная форма) уменьшает на 1 значение j, однако значением выражения j–– является предыдущее значение j (до его уменьшения).

Например, если n равно 4, то при вычислении выражения n++*2 результат равен 8, а n примет значение 5. При n, равном 4, значением выражения ++n*2 будет 10, а n станет равно 5.

Внешнюю неоднозначность имеют выражения, в которых знак унарной операции ++ (или ––) записан непосредственно рядом со знаком бинарной операции +:

x+++b   или   z–––d

В этих случаях трактовка выражений однозначна и полностью определяется рангами операций (бинарные аддитивные + и – имеют ранг 4; унарные ++ и –– имеют ранг 2). Таким образом:

x+++b эквивалентно (x++)+b
z−−−d эквивалентно (z−−)−d

Отношения и логические выражения. Отношение определяется как пара арифметических выражений, соединенных (разделенных) знаком операции отношения. Знаки операций отношения (уже были введены выше):

== равно;	**!=** не равно;
< меньше, чем;	**<=** меньше или равно;
> больше, чем;	**>=** больше или равно.

Примеры отношений:

```
a−b>6.3
(x-4)*3==12
6<=44
```

Логический тип в языке Си отсутствует, поэтому принято, что отношение имеет ненулевое значение (обычно 1), если оно истинно, и равно 0, если оно ложно. Таким образом, значением отношения 6<=44 будет 1.

Операции **>**, **>=**, **<**, **<=** имеют один ранг 6 (см. табл. 1.4). Операции сравнения на равенство **= =** и **!=** также имеют одинаковый, но более низкий ранг 7, чем остальные операции отношений. Арифметические операции имеют более высокий ранг, чем операции отношений, поэтому в первом примере для выражения a–b не нужны скобки.

Логических операций в языке Си три:

- ! – отрицание, то есть логическое НЕ (*ранг 2*);
- && – конъюнкция, то есть логическое И (*ранг 11*);
- || – дизъюнкция, то есть логическое ИЛИ (*ранг 12*).

Они перечислены по убыванию старшинства (ранга). Как правило, логические операции применяются к отношениям. До выполнения логических операций вычисляются значения отношений, входящих в логическое выражение. Например, если **a**, **b**, **c** – переменные, соответствующие длинам сторон треугольника, то для них должно быть истинно, то есть не равно 0, следующее логическое выражение:

```
a+b>c  &&  a+c>b  &&  b+c>a
```

Несколько операций одного ранга выполняются слева направо, причем вычисления прерываются, как только будет определена истинность (или ложность) результата, то есть если в рассмотренном

примере a+b окажется не больше c, то остальные отношения не рассматриваются – результат ложен.

Так как значением отношения является целое (0 или 1), то ничто не противоречит применению логических операций к целочисленным значениям. При этом принято, что любое ненулевое положительное значение воспринимается как истинное, а ложной считается только величина, равная нулю. Значением !5 будет 0, значением 4 && 2 будет 1 и т. д.

Присваивание. Как уже говорилось, символ «=» в языке Си обозначает бинарную операцию, у которой в выражении должно быть два операнда – левый (модифицируемое именующее выражение – обычно переменная) и правый (обычно выражение). Если z – имя переменной, то

z = 2.3 + 5.1

есть выражение со значением 7.4. Одновременно это значение присваивается и переменной z. Только в том случае, когда в конце выражения с операцией присваивания помещен символ «;», это выражение становится оператором присваивания. Таким образом,

z = 2.3 + 5.1;

есть оператор присваивания переменной z значения, равного 7.4.

Тип и значение выражения с операцией присваивания определяются значением выражения, помещенного справа от знака «=». Однако этот тип может не совпадать с типом переменной из левой части выражения. В этом случае при определении значения переменной выполняется преобразование (приведение) типов (о правилах приведения см. ниже в этом параграфе).

Так как выражение справа от знака «=» может содержать, в свою очередь, операцию присваивания, то в одном операторе присваивания можно присвоить значения нескольким переменным, то есть организовать «множественное» присваивание, например:

```
c = x = d = 4.0 + 2.4;
```

Здесь значение 6.4 присваивается переменной d, затем 6.4 как значение выражения с операцией присваивания «d=4.0+2.4» присваивается x и, наконец, 6.4 как значение выражения «x=d» присваивается c. Естественное ограничение – слева от знака «=» в каждой из операций присваивания может быть только леводопустимое выражение (в первых главах книги – имя переменной).

В языке Си существует целый набор «составных операций присваивания» (*ранг 14* в табл. 1.4). Как уже говорилось в §1.4, каждая из составных операций присваивания объединяет некоторую бинарную логическую или арифметическую операцию и собственно присваивание. Операция составного присваивания может использоваться следующим образом:

имя_переменной **ор**=*выражение;*

где **ор** – одна из операций *, /, %, +, –, &, ^, |, <<, >>. Если рассматривать конструкцию «**ор=**» как две операции, то вначале выполняется **ор,** а затем «=». Например:

```
x*=2;        z+=4;        i/=x+4*z;
```

При выполнении каждого из этих операторов операндами для операции **ор** служат переменная из левой части и выражение из правой. Результат присваивается переменной из левой части.

Таким образом, первый пример можно рассматривать как обозначение требования «удвоить значение переменной x»; второй пример – «увеличить на 4 значение переменной z»; третий пример – «уменьшить значение переменной i в (x+4*z) раз». Этим операторам эквивалентны такие операторы присваивания:

```
x=x*2;        z=z+4;        i=i/(x+4*z);
```

В последнем из них пришлось ввести скобки для получения правильного результата. Обратите внимание на то, что использовать операции составного присваивания можно только в тех случаях, когда одна переменная используется в обеих частях. Более того, для некоторых операций эта переменная должна быть обязательно первым (левым) операндом. Например, не удастся заменить составными следующие простые операторы присваивания:

```
a=b/a;        x=z%x.
```

Приведение типов. Рассматривая операцию деления, мы отметили, что при делении двух целых операндов результат получается целым. Например, значением выражения 5/2 будет 2, а не 2.5. Для получения вещественного результата нужно выполнять деление не целых, а вещественных операндов, например, записав 5.0/2.0, получим значение 2.5.

Если операндами являются безымянные константы, то заменить целую константу (как мы только что сделали) на вещественную совсем не трудно. В том случае, когда операндом является именованная константа, переменная или выражение в скобках, необходимо для решения той же задачи использовать операцию явного приведения (преобразования) типа. Например, рассмотрим такой набор определений и операторов присваивания:

```
int n=5, k=2;
double d;
int m;
d=(double) n/ (double) k;
m=n/k;
```

В этом фрагменте значением d станет величина 2.5 типа **double**, а значением переменной m станет целое значение 2.

Операция деления является только одной из бинарных операций. Почти для каждой из них операнды могут иметь разные типы. Однако не всегда программист должен в явном виде указывать преобразования типов. Если у бинарной операции операнды имеют разные типы (а должны в соответствии с синтаксисом выражения иметь один тип), то компилятор выполняет преобразование типов автоматически, то есть приводит оба операнда к одному типу. Например, для тех же переменных значение выражения d+k будет иметь тип **double** за счет неявного преобразования, выполняемого автоматически без указания программиста. Рассмотрим правила, по которым такие приведения выполняются.

Правила преобразования типов. При вычислении выражений некоторые операции требуют, чтобы операнды имели соответствующий тип, а если требования к типу не выполнены, принудительно вызывают выполнение нужных преобразований. Та же ситуация возникает при инициализации, когда тип инициализирующего выражения приводится к типу определяемого объекта. Напомним, что в языке Си присваивание является бинарной операцией, поэтому сказанное относительно преобразования типов относится и ко всем формам присваивания, однако при присваиваниях значение выражения из правой части всегда приводится к типу переменной из левой части, независимо от соотношения этих типов.

Правила преобразования в языке Си для основных типов определены стандартом языка. Эти стандартные преобразования включают перевод «низших» типов в «высшие».

Среди преобразований типов выделяют:

- преобразования в *арифметических выражениях*;
- преобразования при присваиваниях;
- преобразования *указателей*.

Преобразование типов указателей будет рассмотрено в главе 4. Здесь рассмотрим преобразования типов при арифметических операциях и особенности преобразований типов при присваиваниях.

При преобразовании типов нужно различать преобразования, изменяющие внутреннее представление данных, и преобразования, изменяющие только интерпретацию внутреннего представления. Например, когда данные типа **unsigned int** переводятся в тип **int**, менять их внутреннее представление не требуется – изменяется только интерпретация. При преобразовании значений типа **double** в значение типа **int** недостаточно изменить только интерпретацию, необходимо изменить длину участка памяти для внутреннего представления и кодировку. При таком преобразовании из **double** в **int** возможен выход за диапазон допустимых значений типа **int**, и реакция на эту ситуацию существенно зависит от конкретной реализации. Именно поэтому для сохранения мобильности программ в них рекомендуется с осторожностью применять неявные преобразования типов.

Рассмотрим последовательность выполнения преобразования операндов в арифметических выражениях.

1. Все короткие целые типы преобразуются в типы не меньшей длины в соответствии с табл. 1.5. Затем оба значения, участвующие в операции, принимают одинаковый тип в соответствии со следующими ниже правилами.
2. Если один из операндов имеет тип **long double**, то второй тоже будет преобразован в **long double**.
3. Если п. 2 не выполняется и один из операндов есть **double**, другой приводится к типу **double**.
4. Если пп. 2–3 не выполняются и один из операндов имеет тип **float**, то второй приводится к типу **float**.
5. Если пп. 2–4 не выполняются (оба операнда целые) и один операнд **unsigned long int**, то оба операнда преобразуются к типу **unsigned long int**.
6. Если пп. 2–5 не выполняются и один операнд есть **long**, другой преобразуется к типу **long**.
7. Если пп. 2–6 не выполняются и один операнд **unsigned**, то другой преобразуется к типу **unsigned**.

8. Если пп. 2–7 не выполнены, то оба операнда принадлежат типу **int**.

Таблица 1.5. Правила стандартных арифметических преобразований

Исходный тип	Преобразованный тип	Правила преобразований
char	**int**	Расширение нулем или знаком в зависимости от умолчания для **char**
unsigned char	**int**	Старший байт заполняется нулем
signed char	**int**	Расширение знаком
short	**int**	Сохраняется то же значение
unsigned short	**unsigned int**	Сохраняется то же значение
enum	**int**	Сохраняется то же значение
Битовое поле	**int**	Сохраняется то же значение

Используя арифметические выражения, следует учитывать приведенные правила и не попадать в «ловушки» преобразования типов, так как некоторые из них приводят к потерям информации, а другие изменяют интерпретацию битового (внутреннего) представления данных.

На рис. 1.2 стрелками отмечены «безопасные» арифметические преобразования, гарантирующие сохранение точности и неизменность численного значения.

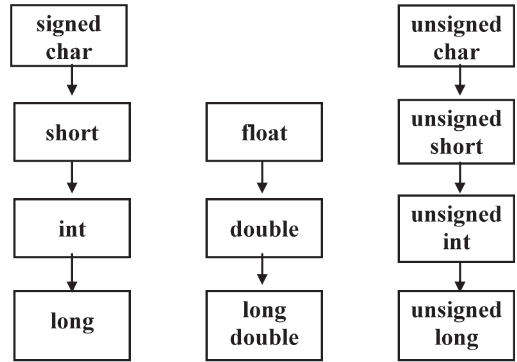


Рис. 1.2. *Арифметические преобразования типов, гарантирующие сохранение значимости*

При преобразованиях, которые не отнесены схемой (рис. 1.2) к безопасным, возможны существенные информационные потери. Для оценки значимости таких потерь рекомендуется проверить обратимость преобразования типов. Преобразование целочисленных значений в вещественные осуществляется настолько точно, насколько это предусмотрено аппаратурой. Если конкретное целочисленное значение не может быть точно представлено как вещественное, то младшие значащие цифры теряются и обратимость невозможна.

Приведение вещественного значения к целому типу выполняется за счет отбрасывания дробной части. Преобразование целой величины в вещественную также может привести к потере точности.

Операция поразрядного отрицания (дополнения или инвертирования битов) обозначается символом «**~**» и является унарной (одноместной), то есть действует на один операнд, который должен быть целого типа. Значение операнда в виде внутреннего битового представления обрабатывается таким образом, что формируется значение той же длины (того же типа), что и операнд. В битовом представлении результата содержатся 1 во всех разрядах, где у операнда 0, и 0 в тех разрядах, где у операнда 1. Например:

```
unsigned char E='\0301', F;
F=~E;
```

Значением F будет восьмеричный код '\076' символа '>' (см. приложение 1). Действительно, битовые представления значений E и F можно изобразить так:

11000001 – для значения переменной E, то есть для '\0301';

00111110 – для значения переменной F, то есть для '\076'.

За исключением дополнения, все остальные поразрядные операции бинарные (двухместные).

Операции сдвигов **>>** (вправо) и **<<** (влево) должны иметь целочисленные операнды. Над битовым представлением значения левого операнда выполняется действие – сдвиг. Правый операнд определяет величину поразрядного сдвига. Например:

5<<2 будет равно 20;

5>>2 будет равно 1.

Битовые представления тех же операций сдвига можно изобразить так:

101<<2 равно 10100, то есть 20;

101>>2 равно 001, то есть 1.

При сдвиге влево на N позиций двоичное представление левого операнда сдвигается, а освобождающиеся слева разряды заполняются нулями. Такой сдвиг эквивалентен умножению значения операнда на 2^N.

Сдвиг вправо на N позиций несколько сложнее. Тут следует отметить две особенности. Первое – это исчезновение младших разрядов, выходящих за разрядную сетку. Вторая особенность – отсутствие стандарта на правило заполнения освобождающихся левых разрядов. В стандарте языка сказано, что когда левый операнд есть целое значение с отрицательным знаком, то при сдвиге вправо заполнение освобождающихся левых разрядов определяется ***реализацией***. Здесь возможны два варианта: освобождающиеся разряды заполняются значениями знакового разряда (арифметический сдвиг вправо) или освобождающиеся слева разряды заполняются нулями (логический сдвиг вправо).

При положительном левом операнде сдвиг вправо на N позиций эквивалентен уменьшению значения левого операнда **в раз** с отбрасыванием дробной части результата. (Поэтому 5>>2 равно 1.)

К автору: во сколько раз?

Операция «поразрядное исключающее ИЛИ». Эта операция имеет очень интересные возможности. Она применима к целым операндам. Результат формируется при поразрядной обработке битовых кодов операндов. В тех разрядах, где оба операнда имеют одинаковые двоичные значения (1 и 1 или 0 и 0), результат принимает значение 1. В тех разрядах, где биты операндов не совпадают, результат равен 0. Пример использования:

```
char a='A';  /* внутренний код 01000001 */
char z='Z';  /* внутренний код 01011010 */
a=a^z;        /* результат: 11100100 */
z=a^z;        /* результат: 01000001 */
a=a^z;        /* результат: 01011010 */
```

Переменные a и z «обменялись» значениями без использования вспомогательной переменной!

Поразрядная дизъюнкция (поразрядное ИЛИ) применима к целочисленным операндам. В соответствии с названием она позволяет получить 1 в тех разрядах результата, где не одновременно равны 0 биты обоих операндов. Например:

5 | 6 равно 7 (для 5 – код 101, для 6 – код 110);

10 | 8 равно 10 (для 10 – код 1010, для 8 – код 1000).

Поразрядная конъюнкция (поразрядное И) применима к целочисленным операндам. В битовом представлении результата только те биты равны 1, которым соответствуют единичные биты обоих операндов. Примеры:

5&6 равно 4 (для 5 – код 101, для 6 – код 110);

10&8 равно 8 (для 10 – код 1010, для 8 – код 1000).

Условное выражение. Как уже говорилось в §1.4, операция, вводимая двумя лексемами '?' и ':' (она имеет *ранг 13*), является уникальной. Во-первых, в нее входит не одна, а две лексемы, во-вторых, она трехместная, то есть должна иметь три операнда. С ее помощью формируется условное выражение, имеющее такой вид:

операнд_1 **?** *операнд_2* **:** *операнд_3*

Все три операнда – выражения. *Операнд_1* – это арифметическое выражение и чаще всего отношение либо логическое выражение. Типы *операнда_2* и *операнда_3* могут быть разными (но они должны быть одного типа или должны автоматически приводиться к одному типу).

Первый операнд является условием, в зависимости от которого вычисляется значение выражения в целом. Если значение первого операнда отлично от нуля (условие истинно), то вычисляется значение *операнда_2*, и оно становится результатом. Если значение первого операнда равно 0 (то есть условие ложно), то вычисляется значение *операнда_3*, и оно становится результатом.

Примеры применения условного выражения мы уже приводили в §1.4.

Контрольные вопросы

1. Какие типы данных приняты в языке и как они определяются (описываются)?
2. Какие операции над данными допустимы в языке, как строятся с их помощью выражения и как они выполняются?
3. Дайте определение служебного слова.
4. Как используются служебные слова для обозначения типов данных?
5. Перечислите типы констант.
6. Какой тип имеет целочисленная константа без суффикса?
7. Совпадают ли коды символов '\0' и '0'?
8. Перечислите суффиксы, определяющие тип целой константы.

9. Перечислите суффиксы, определяющие тип вещественной константы.
10. Объясните назначения эскейп-последовательностей.
11. Чем различаются знаковые и беззнаковые целые?
12. Каковы размеры участков памяти, выделяемых для представления арифметических констант?
13. Из каких частей состоит вещественная константа?
14. Как в языке Си определяется понятие объекта?
15. Что такое «переменная»?
16. Приведите форму определения переменных.
17. Перечислите арифметические операции в порядке возрастания их рангов.
18. Объясните различия между префиксной и постфиксной формами операций декремента и инкремента.
19. Объясните возможности применения запятой в качестве операции.
20. Приведите примеры использования поразрядных операций и операций сдвигов.
21. Знаки каких бинарных операций могут использоваться в составных операциях присваивания?
22. Какого типа должны быть операнды тернарной (условной) операции?
23. К каким операндам применимы операции ++ и −−?
24. В чем особенность деления целочисленных операндов?
25. Назовите правила выполнения операции %.
26. Перечислите арифметические преобразования, гарантирующие сохранение значимости.

Глава 2
ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА СИ

2.1. Структура и компоненты простой программы

В первой главе мы рассмотрели лексемы языка, способы определения констант и переменных, правила записи и вычисления выражений. Несколько слов было сказано об операции присваивания, об операторе присваивания и о том, что каждое выражение превращается в оператор, если в конце выражения находится разделитель «точка с запятой». В этой главе перейдем собственно к программированию, то есть рассмотрим операторы, введем элементарные средства ввода-вывода, опишем структуру однофайловой программы и на несложных примерах вычислительного характера продемонстрируем особенности программирования на языке Си.

Текст программы и препроцессор. Каждая программа на языке Си есть последовательность препроцессорных директив, описаний и определений глобальных объектов и функций. Препроцессорные директивы (в главе 1 мы упоминали директивы **#include** и **#define**) управляют преобразованием текста программы до ее компиляции. Определения вводят функции и объекты. Объекты необходимы для представления в программе обрабатываемых данных. Функции определяют потенциально возможные действия программы. Описания уведомляют компилятор о свойствах и именах тех объектов и функций, которые определены в других частях программы (например, ниже по ее тексту или в другом файле).

Программа на языке Си должна быть оформлена в виде одного или нескольких текстовых файлов. Текстовый файл разбит на строки. В конце каждой строки есть признак ее окончания (плюс управ-

ляющий символ перехода к началу новой строки). Просматривая текстовый файл на экране дисплея, мы видим последовательность строк, причем признаки окончания строк невидимы, но по ним производится разбивка текста редактором.

Определения и описания программы на языке Си могут размещаться в строках текстового файла достаточно произвольно (в свободном формате). Для препроцессорных директив существуют ограничения. Во-первых, препроцессорная директива обычно размещается в одной строке, то есть признаком ее окончания является признак конца строки текста программы. Во-вторых, символ '**#**', вводящий каждую директиву препроцессора, должен быть первым отличным от пробела символом в строке с препроцессорной директивой.

Подробному изучению возможностей препроцессора и всех его директив будет посвящена глава 3. Сейчас достаточно рассмотреть только основные принципы работы препроцессора и изложить общую схему подготовки исполняемого модуля программы, написанной на языке Си. Исходная программа, подготовленная на языке Си в виде текстового файла, проходит три обязательных этапа обработки (рис. 2.1):

- препроцессорное преобразование текста;
- компиляция;
- компоновка (редактирование связей или сборка).

Только после успешного завершения всех перечисленных этапов формируется исполняемый машинный код программы.

Задача препроцессора – преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Каждая препроцессорная директива начинается с символа '**#**'. В этой главе нам будет достаточно директивы **#include**.

Препроцессор «сканирует» исходный текст программы в поиске строк, начинающихся с символа '#'. Такие строки воспринимаются препроцессором как команды (директивы), которые определяют действия по преобразованию текста. Директива **#include** определяет, какие текстовые файлы нужно включить в этом месте текста программы.

Директива **#include <...>** предназначена для включения в текст программы текста файла из каталога «заголовочных файлов», поставляемых вместе со стандартными библиотеками компилятора. Каждая библиотечная функция, определенная стандартом язы-

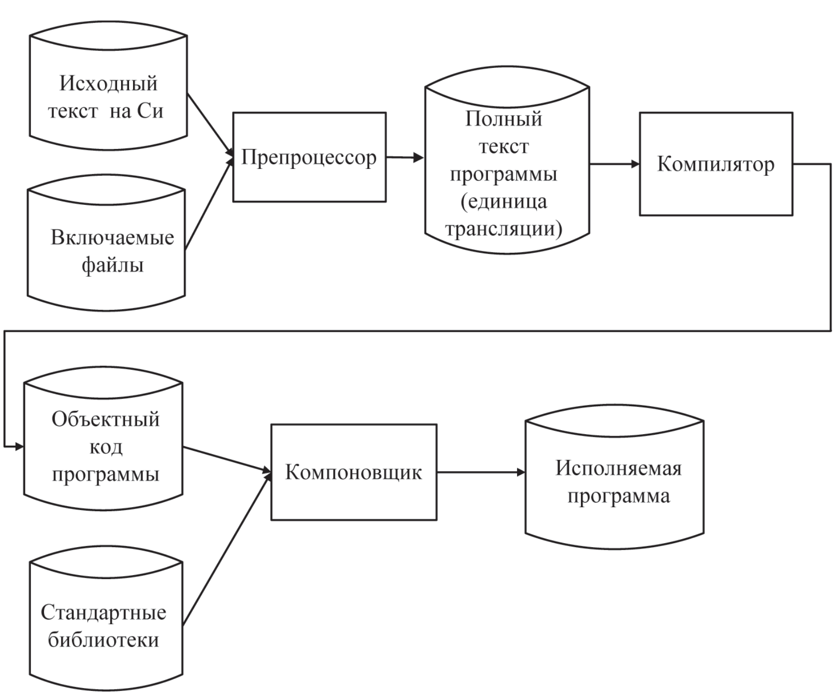


Рис. 2.1. *Схема подготовки исполняемой программы*

ка Си, имеет соответствующее описание (прототип библиотечной функции плюс определения типов, переменных, макроопределений и констант) в одном из заголовочных файлов. Список заголовочных файлов для стандартных библиотек определен стандартом языка.

Важно понимать, что употребление в программе препроцессорной директивы

#include < *имя_заголовочного_файла* **>**

не подключает к программе соответствующую стандартную библиотеку. Препроцессорная обработка выполняется на уровне исходного текста программы. Директива **#include** только позволяет вставить в текст программы описания из указанного заголовочного файла. Подключение к программе кодов библиотечных функций (см. рис. 2.1) осуществляется лишь на этапе редактирования связей (этап компоновки), то есть после компиляции, когда уже получен машинный код программы. Доступ к кодам библиотечных функций нужен лишь на этапе компоновки. Именно поэтому компилировать про-

грамму и устранять синтаксические ошибки в ее тексте можно без стандартной библиотеки, но обязательно с заголовочными файлами.

Здесь следует отметить еще одну важную особенность. Хотя в заголовочных файлах содержатся описания всех стандартных функций, в код программы включаются только те функции, которые используются в программе. Выбор нужных функций выполняет компоновщик на этапе, называемом «редактирование связей».

Термин «заголовочный файл» (*header file*) в применении к файлам, содержащим описания библиотечных функций стандартных библиотек, не случаен. Он предполагает включение этих файлов именно в начало программы. Мы настоятельно рекомендуем, чтобы до обращения к любой функции она была определена или описана в том же файле, где помещен текст программы. Описание или определения функций должны быть «выше» по тексту, чем вызовы функций. Именно поэтому заголовочные файлы нужно помещать в начале текста программы, то есть заведомо раньше обращений к соответствующим библиотечным функциям.

Хотя заголовочный файл может быть включен в программу не в ее начале, а непосредственно перед обращением к нужной библиотечной функции, такое размещение директив **#include <...>** не рекомендуется.

Структура программы. После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы. Теперь программа представляет собой набор описаний и определений. Если не рассматривать (в этой главе) определений глобальных объектов и описаний, то программа будет набором определений функций.

Среди этих функций всегда должна присутствовать функция с фиксированным именем **main**. Именно эта функция является главной функцией программы, без которой программа не может быть выполнена. Имя этой главной функции для всех программ одинаково (всегда **main**) и не может выбираться произвольно. Таким образом, исходный текст программы в простом случае (когда программа состоит только из одной функции) имеет такой вид:

```
директивы_препроцессора
int main( )
{
  определения_объектов;
  операторы;
return 0;
}
```

Заголовочные файлы, с которыми всегда приходится иметь дело, рекомендуется помещать в начале текста программы. Именно эта особенность отмечена в предложенном формате программы.

Перед именем каждой функции программы следует помещать сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип **void**. Функция **main( )** является той функцией программы, которая запускается на исполнение по командам операционной системы. Возвращаемое функцией **main( )** значение также передается операционный системе. Если программист не предполагает, что операционная система будет анализировать результат выполнения его программы, то проще всего указать, что возвращаемое значение отсутствует, то есть имеет тип **void**.

Каждая функция (в том числе и **main**) в языке Си должна иметь набор параметров. Этот набор может быть пустым, тогда в скобках после имени функции помещается служебное слово **void** либо скобки остаются пустыми. В отличие от обычных функций, главная функция **main( )** может использоваться как с параметрами, так и без них. Состав списка параметров функции **main( )** и их назначение будут рассмотрены в главе 5. Сейчас только отметим, что параметры функции **main( )** позволяют организовать передачу данных из среды выполнения в исполняемую программу, минуя средства, предоставляемые стандартной библиотекой ввода-вывода.

Вслед за заголовком **void main( )** размещается тело функции. Тело функции – это *блок*, последовательность определений, описаний и исполняемых операторов, заключенная в фигурные скобки. Определения и описания в блоке будем размещать до исполняемых операторов. Каждое определение, описание и каждый оператор завершаются символом **';'** (точка с запятой).

Определения вводят объекты, необходимые для представления в программе обрабатываемых данных. Примером таких объектов служат именованные константы и переменные разных типов. Описания уведомляют компилятор о свойствах и именах объектов и функций, определенных в других частях программы. Операторы определяют действия программы на каждом шаге ее выполнения.

Чтобы привести пример простейшей осмысленной программы на языке Си, необходимо ввести оператор, обеспечивающий вывод данных из ЭВМ, например на экран дисплея. К сожалению (или как особенность языка), такого оператора в языке Си НЕТ! Все возможности обмена данными с внешним миром программа на языке Си реализует с помощью библиотечных функций ввода-вывода.

Для подключения к программе описаний средств ввода-вывода из стандартной библиотеки компилятора используется директива **#include <stdio.h>**.

Название заголовочного файла **stdio.h** является аббревиатурой: *std* – *standard* (стандартный), *i* – *input* (ввод), *o* – *output* (вывод), *h* – *head* (заголовок).

Функция форматированного вывода. Достаточно часто для вывода информации из ЭВМ в программах используется функция **printf( )**. Она переводит данные из внутреннего кода в символьное представление и выводит полученные изображения символов результатов на экран дисплея. При этом у программиста имеется возможность форматировать данные, то есть влиять на их представление на экране дисплея.

Возможность форматирования условно отмечена в самом имени функции с помощью литеры f в конце ее названия (*print formatted*).

Оператор вызова функции **printf( )** можно представить так:

printf (*форматная_строка, список_аргументов*);

Форматная строка ограничена двойными кавычками (см. строковые константы, §1.2) и может включать произвольный текст, управляющие символы и спецификации преобразования данных. *Список аргументов* (с предшествующей запятой) может отсутствовать. Именно такой вариант использован в классической первой программе на языке Си [1, 2]:

```
#include <stdio.h>
void main( )
{
 printf ("\n Здравствуй, Мир!\n");
}
```

Директива **#include** <**stdio.h**> включает в текст программы описание (прототип) библиотечной функции **printf( )**. (Если удалить из текста программы эту препроцессорную директиву, то появятся сообщения об ошибках и исполнимый код программы не будет создан. Среди параметров функции **printf( )** есть в этом примере только форматная строка (список аргументов отсутствует). В форматной строке два управляющих символа '\n' – «перевод строки». Между ними текст, который выводится на экран дисплея:

```
Здравствуй, Мир!
```

Первый символ '\n' обеспечивает вывод этой фразы с начала новой строки. Второй управляющий символ '\n' переведет курсор к началу следующей строки, где и начнется вывод других сообщений (не связанных с программой) на экран дисплея.

Итак, произвольный текст (не спецификации преобразования и не управляющие символы) непосредственно без изменений выводится на экран. Управляющие символы (перевод строки, табуляция и т. д.) позволяют влиять на размещение выводимой информации на экране дисплея.

Спецификации преобразования данных предназначены для управления формой внешнего представления значений аргументов функции **printf( )**. Обобщенный формат спецификации преобразования имеет вид: *%флажки ширина_поля.точность модификатор спецификатор*

Среди элементов спецификации преобразования обязательными являются только два – символ '%' и *спецификатор*.

В задачах вычислительного характера этой главы будем использовать спецификаторы:

- **d** – для целых десятичных чисел (тип **int**);
- **u** – для целых десятичных чисел без знака (тип **unsigned**);
- **f** – для вещественных чисел в форме с фиксированной точкой (типы **float** и **double**);
- **e** – для вещественных чисел в форме с плавающей точкой (с мантиссой и порядком) – для типов **double** и **float**;
- **g** – наиболее компактная запись из двух вариантов: с плавающей или фиксированной точкой.

В список аргументов функции **printf( )** включают объекты, значения которых должны быть выведены из программы. Это выражения и их частные случаи – переменные и константы. Количество аргументов и их типы должны соответствовать последовательности спецификаций преобразования в форматной строке. Например, если вещественная переменная summa имеет значение 2102.3, то при таком вызове функции

```
printf(«\n summa=%f», summa);
```

на экран с новой строки будет выведено:

```
summa=2102.3
```

После выполнения операторов

```
float c, e;
int k;
c=48.3; k=-83; e=16.33;
printf ("\nc=%f\tk=%d\te=%e", c, k, e);
```

на экране получится такая строка:

```
c=48.299999      k=-83         e=1.63300e+01
```

Здесь обратите внимание на управляющий символ '\t' (табуляция). С его помощью выводимые значения в строке результата отделены друг от друга.

Для вывода числовых значений в спецификации преобразования весьма полезны *«ширина_поля»* и *«точность»*.

Ширина_поля – целое положительное число, определяющее длину (в позициях на экране) представления выводимого значения.

Точность – целое положительное число, определяющее количество цифр в дробной части внешнего представления вещественного числа (с фиксированной точкой) или его мантиссы (при использовании формы с плавающей точкой).

Пример с теми же переменными:

```
printf ("\nc=%5.2\tk=%5d\te=%8.2f\te=%11.4e", c, k, e, e);
```

Результат на экране:

```
c=48.30     k=   -83     e=     16.33     e=  1.6330e+01
```

В качестве *модификаторов* в спецификации преобразования используются символы:

h – для вывода значений типа **short int**;
l – для вывода значений типа **long**;
L – для вывода значений типа **long double**.

Примеры на использование модификаторов пока приводить не будем.

Хотя в разделе, посвященном символам и строковым константам (§1.2), упоминалось о возможностях записи управляющих последовательностей и эскейп-последовательностей внутри строк, остановимся еще раз на этом вопросе в связи с форматной строкой. При

необходимости вывести на экран (на печать) парные кавычки или апострофы их представляют с помощью соответствующих последовательностей: \» или \', то есть заменяют парами литер. Обратная косая черта '\' для однократного вывода на экран должна быть дважды включена в форматную строку.

При необходимости вывести символ % в форматную строку его включают дважды: % %.

Применимость вещественных данных. Даже познакомившись с различиями в диапазонах представления вещественных чисел, начинающий программист не сразу осознает различия между типами **float**, **double** и **long double**. Прежде всего бросается в глаза разное количество байтов, отводимых в памяти для вещественных данных перечисленных типов. На современных ПК:

- для **float** – 4 байта;
- для **double** – 8 байт;
- для **long double** – 10 байт.

По умолчанию все константы, не относящиеся к целым типам, принимают тип **double**. У программиста это соглашение часто вызывает недоумение – а не лучше ли всегда работать с вещественными данными типа **float** и только при необходимости переходить к **double** или **long double**? Ведь значения больше 1E+38 и меньше 1E–38 встречаются довольно редко.

Следующая программа (предложена С. М. Лавреновым) иллюстрирует опасности, связанные с применением данных типа **float** даже в несложных арифметических выражениях:

```
#include <stdio.h>
void main( )
{
 float a, b, c, t1, t2, t3;
 a=95.0;
 b=0.02;
 t1=(a+b)*(a+b);
 t2=-2.0*a*b-a*a;
 t3=b*b;
 c=(t1+t2)/t3;
 printf("\nc=%f\n", c);
}
```

Результат выполнения программы:

```
c=2.441406
```

Если в той же программе переменной **a** присвоить значение 100.0, то результат будет еще хуже:

```
c=0.000000.
```

Таким образом, запрограммированное с использованием переменных типа **float** несложное алгебраическое выражение

$$\frac{(a+b)^2-(a^2+2ab)}{b^2}$$

никак «не хочет» вычисляться и принимать свое явное теоретическое единичное значение.

Если заменить в программе только одну строку, то есть так определить переменные:

```
double a, b, c, t1, t2, t3;
```

значение выражения вычисляется совершенно точно:

```
c=1.000000
```

Приведенный пример и общие положения вычислительной математики заставляют существенно ограничить применение переменных типа **float**. Тип **float** можно выбирать для представления исходных данных или окончательных результатов, получаемых в программе. Однако применение данных типа **float** в промежуточных вычислениях (особенно в итерационных алгоритмах) следует ограничить и всегда использовать **double** либо **long double**.

К сожалению, ни **double**, ни **long double** не снимают полностью проблем конечной точности представления вещественных чисел в памяти ЭВМ. Существенное различие в порядках значений операндов арифметических выражений может привести к подобным некорректным результатам и при использовании типов **double** и **long**.

Выделение лексем из текста программы. В главе 1 мы ввели понятие лексемы, перечислили их группы (идентификаторы, знаки операций и т. д.) и определили состав каждой из групп лексем. Первая задача, которую решает компилятор, – это лексический анализ текста программы. В результате лексического анализа из сплошного текста выделяются лексические единицы (лексемы). Программисту полезно знать, какими правилами при этом руководствуется компилятор.

Компилятор просматривает символы (литеры) текста программы слева направо. При этом его первая задача – выделить лексемы языка. За очередную лексическую единицу принимается ***наибольшая*** последовательность литер, которая образует лексему. Таким образом, из последовательности int_line компилятор не станет выделять как лексему служебное слово **int,** а воспримет всю последовательность как введенный пользователем идентификатор.

В соответствии с тем же принципом выражение d+++b трактуется как d++ +b, а выражение b-->c эквивалентно (b--)>c.

Следующая программа иллюстрирует сказанное:

```
#include <stdio.h>
void main()
{ int n=10,m=2;
   printf("\nn+++m=%d",n+++m);
   printf("\nn=%d, m=%d",n,m);
   printf("\nm-->n=%d",m-->n);
   printf("\nn=%d, m=%d",n,m);
   printf("\nn-->m=%d",n-->m);
   printf("\nn=%d, m=%d",n,m);
}
```

Результат выполнения программы:

```
n+++m=12
n=11,m=2
m-->n=0
n=11,m=1
n-->m=1
n=10,m=1
```

Результаты вычисления выражений n+++m, n-->m, m-->n полностью соответствуют правилам интерпретации выражений на основе таблицы рангов операций (см. табл. 1.4). Унарные операции **++** и **--** имеют ранг 2. Аддитивные операции **+** и **-** имеют ранг 4. Операции отношений имеют ранг 6.

2.2. Элементарные средства программирования

Группы операторов языка Си. Если вспомнить вопросы, перечисленные в начале главы 1, то окажется, что мы уже получили ответы на многие из них. Введены алфавит языка и его лексемы; приведены

основные типы данных, константы и переменные; определены все операции; рассмотрены правила построения арифметических выражений, отношений и логических выражений; описана структура программы; рассмотрены средства вывода из ЭВМ арифметических значений с помощью функции **printf( )**; определен оператор присваивания.

В этом и следующих параграфах второй главы мы ответим на остальные вопросы, сформулированные в главе 1, и этого будет достаточно, чтобы писать на языке Си программы для решения задач вычислительного характера.

Вернемся вновь к структуре простой программы, состоящей только из одной функции с именем **main( )**.

директивы_препроцессора
int main( )
{ *определения_объектов;*
  операторы;
return 0;
}

Как мы уже договорились, пока нам будет достаточно препроцессорной директивы **#include <...>**. В качестве определяемых объектов будем вводить переменные и константы базовых типов. А вот об операторах в теле функции нужно говорить подробно.

Каждый оператор определяет действия программы на очередном шаге ее выполнения. У оператора (в отличие от выражения) нет значения. По характеру действий различают два типа операторов: операторы преобразования данных и операторы управления работой программы.

Наиболее типичные операторы преобразования данных – это операторы присваивания и произвольные выражения, завершенные символом «точка с запятой»:

```
i++;      /*Арифметическое выражение - оператор*/
x*=i;     /*Оператор составного присваивания*/
i=x-4*i;  /*Оператор простого присваивания*/
```

Так как вызов функции является выражением с операцией «круглые скобки» и операндами «имя функции», «список аргументов», к операторам преобразования данных можно отнести и оператор вызова или обращения к функции:

имя_функции (список_ аргументов);

Мы уже использовали обращение к библиотечной функции **printf( )**, параметры которой определяли состав и представление на экране дисплея выводимой из программы информации. С точки зрения процесса преобразования информации, функция **printf( )** выполняет действия по перекодированию данных из их внутреннего представления в последовательность кодов, пригодных для вывода на экран дисплея.

Операторы управления работой программы:

- составные операторы;
- операторы выбора;
- операторы циклов;
- операторы перехода.

К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Отличие блока от составного оператора – наличие определений в теле блока. Например, приведенный ниже фрагмент программы – составной оператор:

```
{
 n++;
 summa+=(float)n;
}
```

а этот фрагмент – блок:

```
{
 int     n=0;
 n++;
 summa+=(float)n;
}
```

Наиболее часто блок употребляется в качестве тела функции.

Операторы выбора – это условный оператор **(if)** и переключатель **(switch)**.

Операторы циклов в языке Си трех видов – с предусловием **(while)**, с постусловием **(do)** и параметрический **(for)**.

Операторы перехода выполняют безусловную передачу управления: **goto** (безусловный переход), **continue** (завершение текущей итерации цикла), **break** (выход из цикла или переключателя), **return** (возврат из функции).

Перечислив операторы управления программой, перейдем к подробному рассмотрению тех из них, которых будет достаточно для программирования простейших алгоритмов.

Условный оператор имеет 2 формы: сокращенную и полную. Сокращенная форма:

if *(выражение_условие) оператор;*

где в качестве *выражения_условия* могут использоваться: арифметическое выражение, отношение и логическое выражение. Оператор, включенный в условный, выполняется только в случае истинности (то есть при ненулевом значении) *выражения_условия*. Пример:

```
if (x < 0 && x > -10) x=-x;
```

Полная форма условного оператора:

if (*выражение_условие*)
оператор_1;
else
оператор_2;

Здесь в случае истинности *выражения_условия* выполняется только *оператор_1*. *Если* значение *выражения_условия* ложно, то выполняется только *оператор_2*. Например:

```
if (x > 0)
b=x;
else
b=-x;
```

Выполнение условного оператора иллюстрируют схемы на рис. 2.2.

Обратим внимание на то, что в условных операторах в качестве любого из операторов (после условия или после **else**) может использоваться составной оператор. Например, при решении алгебраического уравнения 2-й степени $ax^2 + bx + c = 0$ действительные корни имеются только в случае, если дискриминант $(b^2 - 4ac)$ неотрицателен.

Следующий фрагмент программы иллюстрирует использование условного оператора при определении действительных корней x_1, x_2 квадратного уравнения:

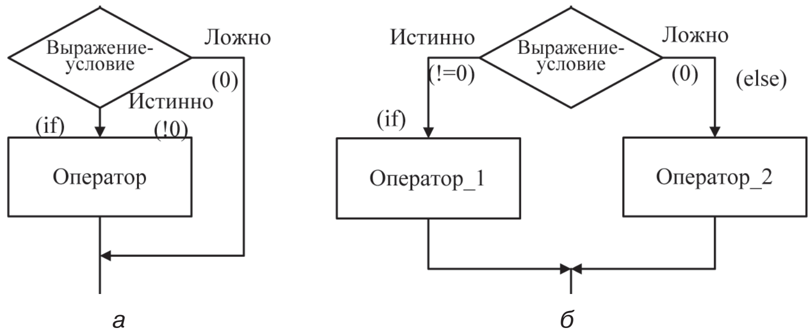


Рис. 2.2. *Схемы условных операторов (выражение-условие – условие после **if**): а – сокращенная форма; б – полная форма*

```
. . .
d=b*b - 4*a*c;     /* d - дискриминант   */
if (d>=0.0)
{
  x1=(-b+sqrt(d))/2/a;
  x2=(-b-sqrt(d))/2/a;
  printf("\n Корни: x1=%e,   x2=%e", x1, x2);
}
else
printf("\n Действительные корни отсутствуют.");
```

Во фрагменте предполагается, что переменные d, b, a, x1, x2 – вещественные (типа **float** либо **double**). До приведенных операторов переменные a, b, c получили конкретные значения, для которых выполняются вычисления. В условном операторе после **if** находится составной оператор, после **else** – только один оператор – вызов функции **printf( )**. При вычислении корней используется библиотечная функция **sqrt( )** из стандартной библиотеки компилятора. Ее прототип находится в заголовочном файле **math.h** (см. приложение 3).

Метки и пустой оператор. Метка – это идентификатор, помещаемый слева от оператора и отделенный от него двоеточием «:». Например:

```
COH: X+=-8;
```

Чтобы можно было поставить метку в любом месте программы (или задать пустое тело цикла), в язык Си введен пустой оператор,

изображаемый только одним символом «;». Таким образом, можно записать такой помеченный пустой оператор:

```
МЕТКА:;
```

Оператор перехода. Оператор безусловного перехода имеет следующий вид:

goto *идентификатор;*

где *идентификатор* – одна из меток программы. Например:

```
goto COH; или goto МЕТКА;
```

Введенных средств языка Си вполне достаточно для написания примитивных программ, которые не требуют ввода исходных данных. Алгоритмы такого сорта довольно редко применяются, но для иллюстрации некоторых особенностей разработки и выполнения программ рассмотрим следующую задачу.

Программа оценки машинного нуля. В вычислительных задачах при программировании итерационных алгоритмов, завершающихся при достижении заданной точности, часто нужна оценка «машинного нуля», то есть числового значения, меньше которого невозможно задавать точность данного алгоритма. Абсолютное значение «машинного нуля» зависит от разрядной сетки, применяемой ЭВМ, от принятой в конкретном трансляторе точности представления вещественных чисел и от значений, используемых для оценки точности. Следующая программа оценивает абсолютное значение «машинного нуля» относительно единицы, представленной переменной типа **float**:

```
1  /* Оценка "машинного нуля" */
2  #include <stdio.h>
3  void main( )
4  { int k;      /*k   - счетчик итераций*/
5     float e,e1; /*e1  - вспомогательная переменная*/
6     e=1.0;      /*e   - формируемый результат*/
7     k=0;
8  M:e=e/2.0;
9     e1=e+1.0;
10    k=k+1;
11    if (e1>1.0) goto M;
12    printf ("\n число делений на 2: %6d\n", k);
```

```
13    printf ("машинный нуль: %e\n", e);
14 }
```

В строках программы слева помещены порядковые номера, которых нет в исходном тексте. Номера добавлены только для удобства ссылок на операторы. Строка 1 – комментарий с названием программы. Комментарии в строках 4, 5, 6 поясняют назначение переменных. Объяснить работу программы проще всего с помощью трассировочной таблицы (табл. 2.1).

Таблица 2.1. Трассировочная таблица

Шаг выполнения	Номер строки	Значения переменных		
		e	k	e1
1	6	**1.0**	–	–
2	7	1.0	**0**	–
3	8	**0.5**	0	–
4	9	0.5	0	**1.5**
5	10	0.5	**1**	1.5
6	11	0.5	1	1.5
7	8	**0.25**	1	1.5
8	9	0.25	1	**1.25**
9	10	0.25	**2**	1.25
10	11	0.25	2	1.25
11	8	**0.125**	2	1.25
...	...	...	...	...

Во втором столбце таблицы указаны номера строк с исполняемыми операторами. Значения переменных даны после выполнения соответствующего оператора. Только что измененное значение переменной в таблице выделено. После подготовительных присваиваний (строки 6, 7) циклически выполняются операторы 8–11 до тех пор, пока истинно отношение e1>1.0, проверяемое в условном операторе. При каждой итерации значение переменной e уменьшается вдвое, и, наконец, прибавление (в строке 9) к 1.0 значения e не изменит результата, то есть e1 станет равно 1.0.

При использовании компилятора gcc ОС FreeBSD получен следующий результат:

```
Число делений на 2:   24
Машинный нуль:   5.960464e-08
```

При использовании в строке 5 для определения переменных e, e1 типа **double**, то есть при использовании двойной точности, получен иной результат:

```
Число делений на 2:   53
Машинный нуль:   1.110223e-16
```

Оба результата не хуже значений, приведенных в приложении 2, для предельных констант **FLT_EPSILON** и **DBL_EPSILON**.

Ввод данных. Для ввода данных с клавиатуры ЭВМ в программе будем использовать функцию (описана в заголовочном файле **stdio.h**):

scanf (*форматная_строка, список_аргументов*);

Функция **scanf( )** выполняет «чтение» кодов, вводимых с клавиатуры. Это могут быть как коды видимых символов, так и управляющие коды, поступающие от вспомогательных клавиш и от их сочетаний. Функция **scanf( )** воспринимает коды, преобразует их во внутренний формат и передает программе. При этом программист может влиять на правила интерпретации входных кодов с помощью спецификаций форматной строки. (Возможность форматирования условно отмечена в названии функции с помощью литеры **f** в конце имени.)

И форматная строка, и список аргументов для функции **scanf( )** обязательны. В форматную строку для функции **scanf( )** входят спецификации преобразования вида:

% * *ширина_поля модификатор спецификатор*

Среди элементов спецификации преобразования обязательны только % и *спецификатор*. Для ввода числовых данных используются спецификаторы:

d – для целых десятичных чисел (тип **int**);
u – для целых десятичных чисел без знака (тип **unsigned int**);
f – для вещественных чисел (тип **float**);
e – для вещественных чисел (тип **float**).

Ширина_поля – целое положительное число, позволяющее определить, какое количество байтов (символов) из входного потока соответствует вводимому значению. Этим элементом мы сейчас не будем пользоваться.

Звездочка '*' в спецификации преобразования позволяет пропустить во входном потоке байты соответствующего вводимого значе-

ния. (Сейчас, когда уже забыли о подготовке данных на перфокартах и перфолентах, звездочка при вводе почти не используется. Она может быть полезной при чтении данных из файлов, когда нужно пропускать те или иные значения.)

В качестве *модификаторов* используются символы:

h – для ввода значений типа **short int (hd)**;
l – для ввода значений типа **long int (ld)** или **double (lf, le)**;
L – для ввода значений типа **long double (Lf, Le)**;
ll – для ввода значений типа **long long**.

В ближайших программах нам не потребуются ни '*', ни модификаторы. Информацию о них приводим только для полноты сведений о спецификациях преобразования данных при вводе.

В отличие от функции **printf( )**, аргументами для функции **scanf( )** могут быть только адреса объектов программы, в частном случае – адреса ее переменных. Не расшифровывая понятия адреса (адресам и указателям будет посвящена глава 4), напомним, что в языке Си имеется специальная унарная операция & получения адреса объекта:

& *имя_объекта*

Выражение для получения адреса переменной будет таким:

& *имя_переменной*

Итак, для обозначения адреса перед именем переменной записывают символ &. Если name – имя переменной, то &name – ее адрес.

Например, для ввода с клавиатуры значений переменных n, z, x можно записать оператор:

```
scanf ("%d%f%f",&n,&z,&x);
```

В данном примере спецификации преобразования в форматной строке не содержат сведений о размерах полей и точностях вводимых значений. Это разрешено и очень удобно при вводе данных, диапазон значений которых определен не строго. Если переменная n описана как целая, z и x – как вещественные типа **float**, то после чтения с клавиатуры последовательности символов 18 18 –0.431 переменная n получит значение 18, z – значение 18.0, x – значение –0.431.

При чтении входных данных функция **scanf( )** (в которой спецификации не содержат сведений о длинах вводимых значений) воспринимает в качестве разделителей полей данных «обобщенные

пробельные символы» – собственно пробелы, символы табуляции, символы новых строк. Изображения этих символов на экране отсутствуют, но у них имеются коды, которые «умеет» распознавать функция **scanf( )**. При наборе входной информации на клавиатуре функция **scanf( )** начинает ввод данных после нажатия клавиши **Enter**. До этого набираемые на клавиатуре символы помещаются в специально выделенную операционной системой область памяти – в буфер клавиатуры и одновременно отображаются на экране в виде строки ввода. До нажатия клавиши **Enter** разрешено редактировать (исправлять) данные, подготовленные в строке ввода.

Рассмотрим особенности применения функции **scanf( )** для ввода данных и принципы построения простых программ на основе следующих несложных задач вычислительного характера.

Вычисление объема цилиндра. В предыдущих задачах не требовалось вводить исходные данные. После выполнения программы на экране появлялся результат. Более общий случай – программа, требующая ввода исходных данных:

```
1  /*Вычисление объема прямого цилиндра*/
2  #include <stdio.h>
3  void main( )
4  {
5   double h, r, v;
6   const float PI = 3.14159;
7   /*h - высота цилиндра, r  - радиус цилиндра*/
8   /*v - объем цилиндра,  PI - число "пи"    */
9   printf("\n Радиус цилиндра r= ");
10  scanf("%lf", &r);
11  printf("Высота цилиндра h= ");
12  scanf("%lf", &h);
13  v = h * PI * r * r;
14  printf("Объем цилиндра: %10.4f" ,v);
15 }
```

В тексте программы несколько особенностей. Определена константа PI, то есть со значением 3.14159 связано имя PI, которое до конца выполнения программы будет именовать только это значение.

Перед каждым вводом помещены (строки 9, 11) обращения к функции **printf( )**, выводящей на экран запрос-подсказку, вслед за которой на экране отображается набираемое на клавиатуре вводимое значение. Функция **scanf( )** считывает только это значение, как только будет нажата клавиша **Enter**, что воспринимается как

признак конца строки ввода. Поэтому очередное выполнение функции **printf( )** выводит данные на следующую строку. В результатах выполнения программ <Enter> обозначает нажатие пользователями клавиши **Enter**. Текст на экране при выполнении программы может быть таким:

```
радиус цилиндра r= 2.0  <Enter>
высота цилиндра h= 4.0  <Enter>
объем цилиндра: 50.2654
```

Здесь пользователь ввел 2.0 для r и 4.0 для h. Другой вариант:

```
радиус цилиндра r= 4.0  <Enter>
высота цилиндра h= 2.0  <Enter>
объем цилиндра: 100.5309
```

Еще раз обратите внимание на использование в функции **scanf( )** не имен переменных, а их адресов &r, &h. Кроме того, обратите внимание на спецификации преобразования **%lf**. Если бы переменные h и r имели тип **float**, то в форматных строках функций **scanf( )** нужно было бы применять спецификации **%f** или **%e**.

Сумма членов ряда Фибоначчи. Ряд Фибоначчи определен, если известны первые два его члена f_1, f_2, так как очередной член $f_i = f_{i-1} + f_{i-2}$ для i > 2. Необходимо вычислить сумму заданного количества (k) первых членов ряда Фибоначчи, если известны первые два: $p = F_1$ и $r = F_2$. Следующая программа решает эту задачу:

```
1      /*Вычисление суммы членов ряда Фибоначчи*/
2      #include <stdio.h>
3      void main( )
4      {
5      int k,i;/*k-число членов; i-номер члена */
6      float s,p,r,f;     /* s - искомая сумма */
7      /*Члены: p -первый; r - второй; f - i-й*/
8  M1: printf("\n Введите число членов ряда k=");
9      scanf("%d",&k);
10     if( k > 2 ) goto M2;
11     printf("\n Ошибка! k должно быть > 2 !");
12     goto M1;
13 M2: printf("\n Первый член ряда p=");
14     scanf("%f",&p);
15     printf("\n Второй член ряда r=");
```

```
16      scanf("%f",&r);
17      i = 3;
18      s = p + r;
19 M:   f = p + r;
20      s = s + f;
21      p = r;
22      r = f;
23      i = i + 1;
24      if ( i <= k ) goto M;
25      printf("\n Сумма членов ряда: %10.3f", s);
26      }
```

Обратите внимание на строки 10–12, где выполняется проверка введенного значения k. Программа может правильно работать только при k>2, поэтому лишь в этом случае выполняется переход из строки 10 к метке M2 (строка 13). В противном случае печатается сообщение об ошибке (в строке 11), и после перехода к метке M1 запрашивается новое значение k. (Для полного понимания работы программы целесообразно составить трассировочную таблицу, выбрав подходящие значения k, p, r.)

Результат (текст на экране) может быть, например, таким:

```
Введите число членов ряда k=-4    <Enter>
Ошибка! k должно быть > 2 !
Введите число членов ряда k=10    <Enter>
Первый член ряда p=4.0
Второй член ряда r=5.0
Сумма членов ряда:    660.000
```

Особенность и недостаток программы состоят в том, что она никогда не закончит вычислений, если не ввести допустимого значения k>2.

2.3. Операторы цикла

Три формы операторов цикла. Как и в других языках программирования, в языке Си существуют специальные средства для организации циклов (операторы циклов), позволяющие упростить их программирование. В большинстве языков программирования оператор цикла состоит из двух элементов – заголовка и тела. Тело включает операторы, выполняемые в цикле, заголовок организует циклическое выполнение операторов тела. В соответствии

с названием заголовок размещается непосредственно перед телом цикла. В языке Си равноправно используются три разных оператора цикла, обозначаемых соответственно служебными словами **while, for, do**.

Циклы **while** и **for** построены по схеме:

заголовок_цикла
тело_цикла

Цикл **do** имеет другую структуру – тело цикла как бы обрамлено (сверху и снизу) конструкциями, организующими циклическое выполнение тела. Поэтому говорить о заголовке цикла **do** в языке Си, по-видимому, некорректно.

Введем форматы перечисленных операторов цикла и приведем примеры их использования. Предварительно отметим, что во всех трех операторах цикла тело цикла – это либо отдельный, либо составной оператор, то есть последовательность операторов, заключенная в операторные скобки {}. Тело цикла может быть и пустым оператором, то есть изображаться только как «;».

Цикл **while** (цикл с предусловием) имеет вид:

while (*выражение_условие*)
тело_цикла

В качестве *выражения_условия* чаще всего используется отношение или логическое выражение. Если оно истинно, то есть не равно 0, то тело цикла выполняется до тех пор, пока *выражение_условие* не станет ложным.

Обратите внимание, что проверка истинности выражения осуществляется до каждого выполнения тела цикла (до каждой итерации). Таким образом, для заведомо ложного *выражения_условия* тело цикла не выполнится ни разу. *Выражение_условие* может быть и арифметическим выражением. В этом случае цикл выполняется, пока значение *выражения_условия* не равно 0.

Цикл **do** (цикл с постусловием) имеет вид:

do
тело_цикла
while (*выражение_условие*);

Выражение_условие логическое или арифметическое, как и в цикле **while**. В цикле **do** тело цикла всегда выполняется по крайней мере один раз. После каждого выполнения тела цикла проверяется истинность *выражения_условия* (на неравенство 0), и, если оно ложно (то

есть равно 0), цикл заканчивается. В противном случае тело цикла выполняется вновь.

Цикл **for** (называемый параметрическим) имеет вид:

for (*выражение_1; выражение_условие; выражение_3*)
тело_цикла

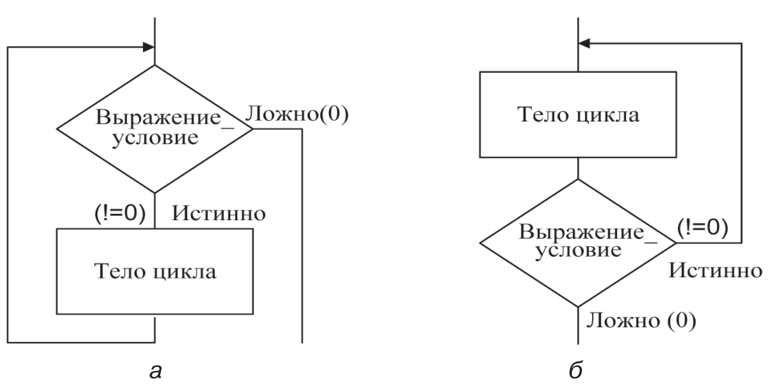
Первое и третье выражения в операторе **for** могут состоять из нескольких выражений, разделенных запятыми. *Выражение_1* определяет действия, выполняемые до начала цикла, то есть задает начальные условия для цикла. *Выражение_условие* – обычно логическое или арифметическое. Оно определяет условия окончания или продолжения цикла. Если оно истинно (то есть не равно 0), то выполняется тело цикла, а затем вычисляется *выражение_3*. *Выражение_3* обычно задает необходимые для следующей итерации изменения параметров или любых переменных тела цикла. После выполнения *выражения_3* вычисляется истинность *выражения_условия,* и все повторяется... Таким образом, *выражение_1* вычисляется только один раз, а *выражение_условие* и *выражение_3* вычисляются после каждого выполнения тела цикла. Цикл продолжается до тех пор, пока не станет ложным *выражение_условие*. Любое из трех, любые два или все три выражения в операторе **for** могут отсутствовать, но разделяющие их символы «;» должны присутствовать всегда. Если отсутствует *выражение_условие*, то считается, что оно истинно и нужны специальные средства для выхода из цикла.

Схемы организации циклов **while, do** и **for** даны на рис. 2.3.

Проиллюстрируем особенности трех типов цикла на примере вычисления приближенного значения

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

для заданного значения x. Вычисления будем продолжать до тех пор, пока очередной член ряда остается больше заданной точности. Обозначим точность через eps, результат – b, очередной член ряда – r, номер члена ряда – i. Для получения i-го члена ряда нужно (i–1)-й член умножить на x и разделить на i, что позволяет исключить операцию возведения в степень и явное вычисление факториала. Опустив определения переменных, операторы ввода и проверки исходных данных, а также вывода результатов, запишем три фрагмента программ.



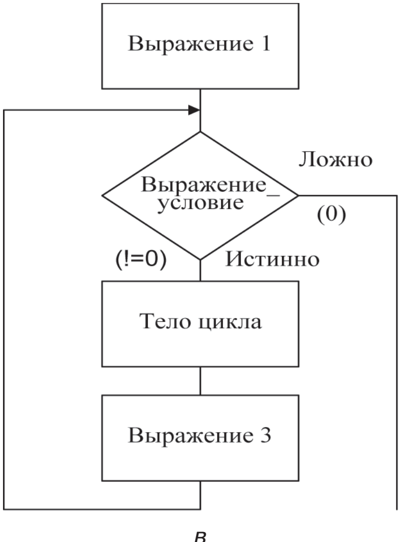


Рис. 2.3. *Схемы организации циклов* ***while****,* ***do****,* ***for****:*
а – цикл с предусловием ***while****; б – цикл с постусловием* ***do****;*
в – параметрический цикл ***for***

```
/* Цикл с предусловием */
i = 2;
b = 1.0;
r = x;
while( r > eps || r < -eps )
```

```
{
 b=b+r;
 r=r*x/i;
 i++;
}
```

Так как проверка точности проводится до выполнения тела цикла, то для окончания цикла абсолютное значение очередного члена должно быть меньше или равно заданной точности.

```
/* Цикл с постусловием */
i=1;
b=0.0;
r=1.0;
do {
      b=b+r;
      r=r*x/i;
      i++;
    }
while( r >= eps || r <= -eps );
```

Так как проверка точности осуществляется после выполнения тела цикла, то условие окончания цикла – абсолютное значение очередного члена строго меньше заданной точности. Соответствующие изменения внесены и в операторы, выполняемые до цикла.

```
/* Параметрический цикл */
i=2;
b=1.0;
r=x;
for( ; r > eps || r < -eps ; )
{
 b=b+r;
 r=r*x/i;
 i=i+1;
}
```

Условие окончания параметрического цикла такое же, как и в цикле **while**.

Все три цикла записаны по возможности одинаково, чтобы подчеркнуть сходство циклов. Однако в данном примере цикл **for** имеет существенные преимущества. В заголовок цикла (в качестве *выражения_1*) можно ввести инициализацию всех переменных:

```
for ( i=2, b=1.0, r=x; r>eps || r<-eps ; )
{
 b=b+r;
 r=r*x/i;
 i=i+1;
}
```

В *выражение_3* можно включать операцию изменения счетчика членов ряда:

```
for( i=2, b=1.0, r=x; r>eps || r<-eps; i++)
{
 b=b+r;
 r=r*x/i;
}
```

Можно еще более усложнить заголовок, перенеся в него все исполнимые операторы тела цикла:

```
for(i=2, b=1.0, r=x; r>eps || r<-eps;
      b+=r, r*=x/i, i++);
```

В данном случае тело цикла – пустой оператор. Для сокращения *выражения_3* в нем использованы составные операции присваивания и операция ++.

Приближенное значение экспоненты. Приведем полный текст программы для приближенного вычисления экспоненты:

```
1  /*Приближенное значение экспоненты*/
2  #include <stdio.h>
3  void main( )
4  {
5   int i;           /*i - счетчик членов ряда*/
6   double eps,b=1.0,r,x;/*exp - абс. точность*/
7  /*b - значение экспоненты; r - очередной член*/
8  /* x - вводимое значение показателя экспоненты*/
9   printf("\n Введите значение x=");
10  scanf("%lf",&x);
11  do {
12        printf("\n Введите точность eps=");
13        scanf("%lf",&eps);
14      }
```

```
15       while( eps <= 0.0 );
16       for( i=2, r=x; r > eps || r < -eps; i++ )
17       {
18         b=b+r;
19         r=r*x/i;
20     }
21     printf(" Результат: %f\n",b);
22     printf(" Погрешность: %f\n",r);
23     printf(" Число членов ряда: %d\n",i);
24 }
```

В программе переменная b инициализирована (строка 6), то есть ей еще в процессе выделения памяти присваивается конкретное начальное значение (1.0). При вводе значения переменной eps предусмотрена защита от неверного задания точности (строки 11–15), для чего использован цикл **do**. В соответствии с правилами его работы выполняются операторы из строк 12 и 13, а затем проверяется введенное значение точности. Если значение eps недопустимо (eps<=0.0), то операторы тела цикла (строки 12, 13) выполняются снова. Цикл не закончится до тех пор, пока не будет введено допустимое значение точности.

Результаты выполнения программы:

```
Введите значение x=1        <Enter>
Введите точность eps=0.01     <Enter>
Результат: 2.708333
Погрешность: 0.008333
Число членов ряда: 6
```

Другой вариант выполнения программы:

```
Введите значение x=1        <Enter>
Введите точность eps=-0.3     <Enter>
Введите точность eps=0.0001   <Enter>
Результат: 2.718254
Погрешность: 0.000025
Число членов ряда: 9
```

В последнем варианте вначале введено неверное значение eps=–0.3, и ввод пришлось повторить. Отметим, что в качестве погрешности выводится последнее значение учтенного члена ряда, что не вполне математически корректно.

Оператор break. Как видно из предыдущего примера, принятый способ проверки исходных данных с повторным запросом значения точности eps не очень удобен, так как на экране отсутствует сообщение об ошибке. В примере, рассмотренном в предыдущем параграфе (сумма членов ряда Фибоначчи), на экран выводится конкретное указание о сделанной ошибке («Ошибка! k должно быть > 2 !»), что упрощает ее исправление. Однако в этом случае в программе использованы метки и оператор перехода, что считается некорректным с точки зрения структурного программирования.

Добиться такого же результата можно, не нарушая принципов структурного программирования и применяя оператор цикла, если использовать в его теле оператор прерывания **break**. Этот оператор (рис. 2.4) прекращает выполнение оператора цикла и передает управление следующему за ним (за циклом) оператору.

Необходимость в использовании оператора прерывания в теле цикла возникает, когда условие продолжения итераций нужно проверять не в начале цикла (как в циклах **for** и **while**) и не в конце тела цикла (как в цикле **do**), а в середине тела цикла. Наиболее естественна в этом случае такая структура тела цикла:

{
операторы
  if (*условие*) **break**;
  операторы
}

Рассмотрим пример с использованием оператора прерывания **break** в цикле ввода исходных данных.

Сумма отрезка степенного ряда. Введя значения переменных n и g, вычислить сумму $c = \sum_{i=1}^{n} g^i$, где $n > 0$. Задачу решает следующая программа:

```
1  /* Сумма степенного ряда */
2  #include <stdio.h>
3  void main( )
4  {
5   double g,c,p; /* c - сумма, p - член ряда */
6   int i,n;
7   printf("\n Введите значение g=");
8   scanf("%lf",&g);
```

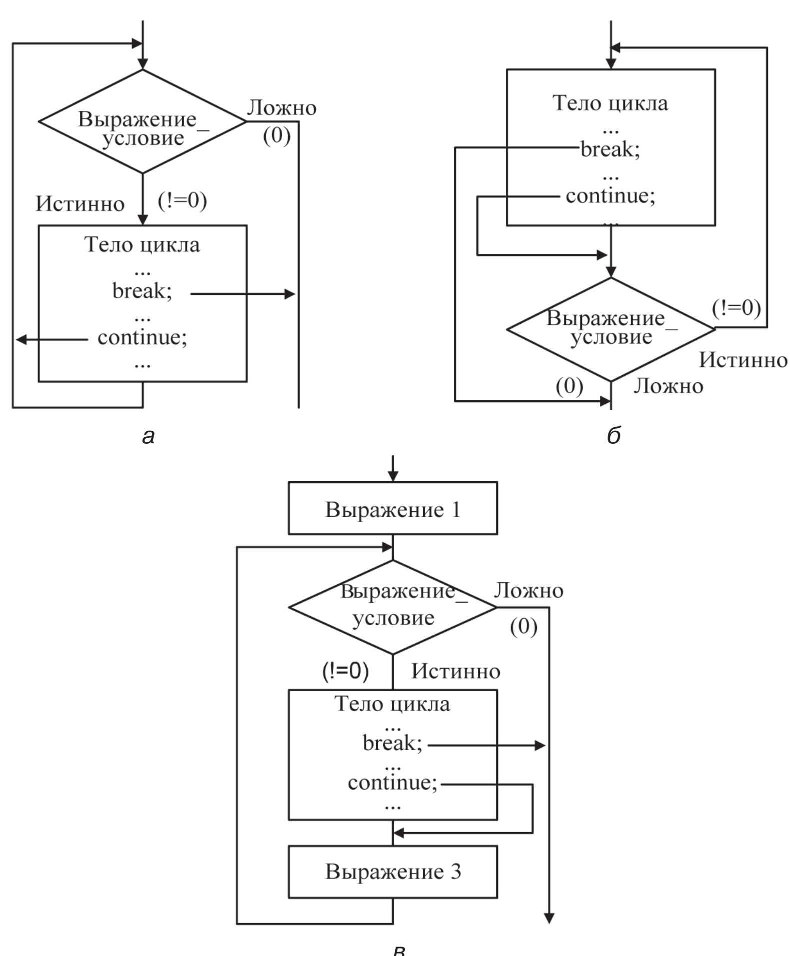


Рис. 2.4. *Схемы выполнения в циклах операторов* ***break*** *и* ***continue****:*
а – цикл с предусловием ***while****; б – цикл с постусловием* ***do****;*
в – параметрический цикл ***for***

```
9    while(1)
10   {
11     printf("\n Введите значение n=");
12     scanf("%d",&n);
13     if( n > 0 ) break;
14     printf("Ошибка! n должно быть >0! \n");
15   }
```

```
16  for( c=0.0, p=1.0, i=1; i <= n; i++ )
17    {
18      p*=g;
19      c+=p;
20    }
21  printf("\n Сумма c=%f",c);
22 }
```

Для защиты от неверных исходных данных использован цикл (строки 9–15), в заголовке которого после **while** записано заведомо истинное выражение 1, то есть цикл может быть прерван только за счет выполнения операторов его тела. В строке 13 оператор **break** выполняется в случае истинности условия «n>0». При этом цикл заканчивается, и следует переход к оператору строки 16. Таким образом, сообщение об ошибке, выводимое функцией **printf( )** из строки 14, печатается для каждого неположительного значения n.

Вычисляя сумму ряда (строки 16–20), очередной его член получаем, умножая предыдущий на g (строка 18). Тем самым устранена необходимость явного возведения в степень. В *выражении_1* оператора **for** формируются начальные значения переменных, изменяемых при выполнении цикла. В строках 18, 19 использованы составные операции присваивания.

Приведем результаты выполнения программы:

```
Введите значение g=8.8         <Enter>
Введите значение n=-15         <Enter>
Ошибка! n должно быть >0!
Введите значение n=15          <Enter>
Сумма c=165816655682177404000
```

Полученное значение суммы довольно трудно прочесть. По-видимому, в этой задаче следует применять печать результата в экспоненциальной форме, то есть в **printf( )** из строки 21 использовать спецификацию преобразования %e. При этом получится такой результат:

```
Введите значение n=15          <Enter>
Сумма c=1.658167e+14.
```

Оператор continue. Еще одну возможность влиять на выполнение операторов тела цикла обеспечивает оператор перехода к следующей итерации цикла **continue** (см. рис. 2.4). Оператор **continue** позволяет в любой точке тела цикла прервать текущую итерацию и перейти

к проверке условий продолжения цикла, определенных в предложениях **for** или **while**. В соответствии с результатами проверки выполнение цикла либо заканчивается, либо начинается новая итерация. Оператор **continue** удобен, когда от итерации к итерации изменяется последовательность выполняемых операторов тела цикла, то есть когда тело цикла содержит ветвления. Рассмотрим пример.

Суммирование положительных чисел. Вводя последовательность чисел, заканчивающуюся нулем, получить сумму и количество только положительных из них. Следующая программа решает эту задачу:

```
#include <stdio.h>
/* Сумма положительных чисел  */
void main( )
{
   double s,x; /*x - очередное число, s - сумма*/
   int k;      /*k - количество положительных  */
printf("\nВведите последоват. чисел с 0 в конце:\n");
   for( x=1.0, s=0.0, k=0; x != 0.0; )
   {
    scanf("%lf",&x);
    if( x <= 0.0 ) continue;
    k++; s+=x;
   }
printf("\n Сумма=%f, колич. положит.=%d",s,k);
}
```

Результат выполнения программы:

```
Введите последовательность чисел с 0 в конце:
6  -3.0  14.0  -5  -4  10  0.0     <Enter>
Сумма=30.000000, колич. Положит.=3
```

Недостаток приведенной программы состоит в том, что нет защиты от неверно введенных данных. Например, не предусмотрены действия, когда в последовательности отсутствует нулевой элемент.

2.4. Массивы и вложение операторов цикла

Массивы и переменные с индексами. Математическим понятием, которое привело к появлению в языках программирования понятия «массив», являются матрица и ее частные случаи: вектор-столбец

или вектор-строка. Элементы матриц в математике принято обозначать с использованием индексов. Существенно, что все элементы матриц либо вещественные, либо целые и т. п. Такая «однородность» элементов свойственна и массиву, определение которого описывает тип элементов, имя массива и его размерность, то есть число индексов, необходимое для обозначения конкретного элемента. Кроме того, в определении указывается количество значений, принимаемых каждым индексом. Например, **int** a[10]; определяет массив из 10 элементов a[0], a[1], ..., a[9]. **float** z[13][[6]; определяет двумерный массив, первый индекс которого принимает 13 значений от 0 до 12, второй индекс принимает 6 значений от 0 до 5. Таким образом, элементы двумерного массива z можно перечислить так:

```
Z[0][0],  Z[0][1],  Z[0][2],...,Z[12][4],  Z[12][5]
```

В соответствии с синтаксисом Си в языке существуют только одномерные массивы, однако элементами одномерного массива, в свою очередь, могут быть массивы. Поэтому двумерный массив определяется как массив массивов. Таким образом, в примере определен массив z из 13 элементов-массивов, каждый из которых, в свою очередь, состоит из 6 элементов типа **float**. Обратите внимание, что нумерация элементов любого массива всегда начинается с 0, то есть индекс изменяется от 0 до n–1, где n – количество значений индекса.

Ограничений на размерность массивов, то есть на число индексов у его элементов, в языке Си теоретически нет. Стандарт языка Си требует, чтобы транслятор мог обрабатывать определения массивов с размерностью до 31. Однако чаще всего используются одномерные и двумерные массивы. Продемонстрируем на простых вычислительных задачах некоторые приемы работы с массивами.

Вычисление среднего и дисперсии. Введя значение n из диапазона (0<n<=100) и значения n первых элементов массива x[0], x[1],...,x[n–1], вычислить среднее и оценку дисперсии значений введенных элементов массива. Задачу решает следующая программа:

```
1  /* Вычисление среднего и дисперсии */
2  #include <stdio.h>
3  void main ( )
4  {           /*n - количество элементов */
5   int i,j,n; /*b - среднее, d - оценка дисперсии; */
6   float a,b,d,x[100],e; /*a,e - вспомогательные*/
7   while (1)
```

```
8   {
9    printf("\n Введите значение n=");
10   scanf("%d", &n);
11   if( n > 0 && n <= 100 ) break;
12   printf("\n Ошибка! Необходимо 0<n<101 ");
13  } /* Конец цикла ввода значения n */
14  printf("\n Введите значения элементов:\n");
15  for( b=0.0,i=0; i<n; i++)
16  {
17   printf("x[%d] =", i);
18   scanf("%f", &x[i]);
19   b+=x[i];/* Вычисление суммы элементов */
20  }
21  b/=n;/* Вычисление среднего */
22  for(j=0,d=0.0; j<n; j++)
23  {
24   a=x[j]-b;
25   d+=a*a;
26  }      /* Оценка дисперсии*/
27  d/=n;
28  printf("\n Среднее =%f, дисперсия=%f",b,d);
29 }
```

В программе определен (строка 6) массив x со 100 элементами, хотя в каждом конкретном случае используются только первые n из них. Ввод значения n сопровождается проверкой допустимости вводимого значения (строки 7–13). В качестве условия после **while** записано заведомо истинное выражение 1, поэтому выход из цикла (оператор **break**) возможен только при вводе правильного значения n, удовлетворяющего неравенству 0<n<101. Следующий цикл (строки 15–20) обеспечивает ввод n элементов массива и получение их суммы (b). Затем в цикле (строки 22–26) вычисляется сумма d квадратов отклонений элементов от среднего значения. Возможен следующий результат работы программы:

```
Введите значение n=5  <Enter>
Введите значение элементов:
x[0] = 4          <Enter>
x[1] = 5          <Enter>
x[2] = 6          <Enter>
x[3] = 7          <Enter>
x[4] = 8          <Enter>
Среднее=6.000000, дисперсия=2.000000
```

Вложенные циклы. В теле цикла разрешены любые исполнимые операторы, в том числе и циклы, то есть можно конструировать вложенные циклы. В математике вложенным циклам соответствуют, например, кратные суммы или произведения.

В качестве примера рассмотрим фрагмент программы для получения суммы такого вида:

$$S = \sum_{j=1}^{10} \left[\prod_{i=1}^{5} a_{ji} + \sum_{i=1}^{5} a_{ji} \right].$$

Введем следующие обозначения: a – двумерный массив, содержащий значения элементов матрицы; p – произведение элементов строки матрицы; c – сумма их значений; s – искомая сумма (результат). Опустив определения переменных и операторы ввода-вывода, запишем текст на языке Си:

```
double  a[10][5];
for(  s=0.0,j=0;  j<10;  j++)
     {
       for(  p=1.0,c=0.0,i=0;  i<5;  i++)
            {
              p*=a[j][i];
              c+=a[j][i];
            }
       s+=c+p;
     }
```

При работе с вложенными циклами следует обратить внимание на правила выполнения операторов **break** и **continue**. Каждый из них действует только в том операторе, в теле которого он непосредственно использован. Оператор **break** прекращает выполнение ближайшего внешнего для него оператора цикла. Оператор **continue** передает управление на ближайшую внешнюю проверку условия продолжения цикла.

Для иллюстрации рассмотрим фрагмент другой программы для вычисления суммы произведений элементов строк той же матрицы:

```
1  double  a[10][5];
2  for  (j=0,s=0.0;  j<10;  j++)
3    {
4       for  (i=0,p=1.0;  i<5;  i++)
5          {
```

```
6           if (a[j][i] == 0.0) break;
7           p*=a[j][i];
8         }
9     if (i <5) continue;
10    s+=p;
11  }
```

Внутренний цикл по i прерывается, если (строка 6) обнаруживается нулевой элемент матрицы. В этом случае произведение элементов столбца заведомо равно нулю, и его не нужно вычислять. Во внешнем цикле (строка 9) проверяется значение i. Если i<5, то есть элемент a[j][i] оказался нулевым, то оператор **continue** передает управление на ближайший оператор цикла (строка 4), и, таким образом, не происходит увеличение s (строка 10) на величину «недосчитанного» значения p. Если внутренний цикл завершен естественно, то i равно 5 и оператор **continue** не может выполняться.

Упорядочение в одномерных массивах. Задаче упорядочения или сортировки посвящены многочисленные работы математиков и программистов. Для демонстрации некоторых особенностей вложения циклов и работы с массивами рассмотрим простейшие алгоритмы сортировки. Необходимо, введя значение переменной 1<n<=100 и значения n первых элементов массива a[0],a[1],...,a[n–1], упорядочить эти первые элементы массива по возрастанию их значений. Текст первого варианта программы:

```
/* Упорядочение элементов массива */
#include <stdio.h>
main( )
{
 int n,i,j;
 double a[100],b;
 while(1)
 {
  printf("\n Введите количество элементов n=");
  scanf("%d",&n);
  if (n > 1 && n <= 100) break;
  printf("Ошибка! Необходимо 1<n<=100!");
 }
 printf("\n Введите значения элементов "
        "массива:\n");
 for(j=0; j<n; j++)
   {
```

```
      printf("a[%d]=", j+1);
      scanf("%lf",&a[j]);
     }
  for(i=0; i<n-1; i++)
     for(j=i+1; j<n; j++)
       if(a[i]>a[j])
          {
           b=a[i];
           a[i]=a[j];
           a[j]=b;
          }
     printf("\n Упорядоченный массив: \n");
     for(j=0; j<n; j++)
        printf("a[%d]=%f\n",j+1,a[j]);
}
```

Результаты выполнения программы:

```
Введите количество элементов n=-15  <Enter>
Ошибка! Необходимо 1<n<=100!
Введите количество элементов n=3    <Enter>
Введите значения элементов массива:
a[1] = 88.8         <Enter>
a[2] = -3.3         <Enter>
a[3] = 0.11         <Enter>
Упорядоченный массив:
a[1] = -3.3
a[2] = 0.11
a[3] = 88.8
```

Обратите внимание, что при заполнении массива и при печати результатов его упорядочения индексация элементов выполнена от 1 до n, как это обычно принято в математике. В программе на Си это соответствует изменению индекса от 0 до (n–1).

В программе реализован алгоритм прямого упорядочения – каждый элемент a[i], начиная с a[0] и заканчивая a[n–2], сравнивается со всеми последующими, и на место a[i] выбирается минимальный. Таким образом, a[0] принимает минимальное значение, a[1] – минимальное из оставшихся и т. д. Недостаток этого алгоритма состоит в том, что в нем фиксированное число сравнений, независимое от исходного расположения значений элементов. Даже для уже упорядоченного массива придется выполнить то же самое количество

итераций (n–1)*n/2, так как условия окончания циклов не связаны со свойствами, то есть с размещением элементов массива.

Алгоритм попарного сравнения соседних элементов позволяет в ряде случаев уменьшить количество итераций при упорядочении. В цикле от 0 до n–2 каждый элемент a[i] массива сравнивается с последующим a[i+1] (0<i<n–1). Если a[i]>a[i+1], то значения этих элементов меняются местами. Упорядочение заканчивается, если оказалось, что a[i] не больше a[i+1] для всех i. Пусть k – количество перестановок при очередном просмотре. Тогда упорядочение можно осуществить с помощью такой последовательности операторов:

```
do  {
     for  (i=0,k=0;  i<n-1;  i++)
        if  (  a[i]  >  a[i+1]  )
           {
            b=a[i];
            a[i]=a[i+1];
            a[i+1]=b;
            k=k+1;
           }
        n--;
     }
while(  k  >  0  );
```

Здесь количество повторений внешнего цикла зависит от исходного расположения значений элементов массива. После первого завершения внутреннего цикла элемент a[n–1] становится максимальным. После второго окончания внутреннего цикла на место a[n–2] выбирается максимальный из оставшихся элементов и т. д. Таким образом, после j-го выполнения внутреннего цикла элементы a[n–j], ...,a[n–1] уже упорядочены, и следующий внутренний цикл достаточно выполнить только для 0<i<(n–j–1). Именно поэтому после каждого окончания внутреннего цикла значение n уменьшается на 1.

В случае упорядоченности исходного массива внешний цикл повторяется только один раз, при этом выполняется (n–1) сравнений, k остается равным 0. Для случая, когда исходный массив упорядочен по убыванию, количество итераций внешнего цикла равно (n–1), а внутренний цикл последовательно выполняется (n–1)*n/2 раз.

Имеется возможность улучшить и данный вариант алгоритма упорядочения (см., например, Кнут Д. Искусство программирования. Т. 3: Сортировка и поиск. – 2-е изд. – М.: Вильямс, 2007), однако

рассмотренных примеров вполне достаточно для знакомства с особенностями применения массивов и индексированных переменных.

Инициализация массивов. При определении массивов возможна их инициализация, то есть присваивание начальных значений их элементам. По существу (точнее, по результату), инициализация – это объединение определения объекта с одновременным присваиванием ему конкретного значения. Использование инициализации позволяет изменить формат определения массива. Например, можно явно не указывать количество элементов одномерного массива, а только перечислить их начальные значения в списке инициализации:

```
double d[ ]={1.0, 2.0, 3.0, 4.0, 5.0};
```

В данном примере длину массива компилятор вычисляет по количеству начальных значений, перечисленных в фигурных скобках. После такого определения элемент d[0] равен 1.0, d[1] равен 2.0 и т. д. до d[4], который равен 5.0.

Если в определении массива явно указан его размер, то количество начальных значений не может быть больше количества элементов в массиве. Если количество начальных значений меньше, чем объявленная длина массива, то начальные значения получат только первые элементы массива (с меньшими значениями индекса):

```
int m[8]={8,4,2};
```

В данном примере определены значения только переменных m[0], m[1] и m[2], равные соответственно 8, 4 и 2. Элементы m[3], ..., m[7] не инициализируются.

Правила инициализации многомерных массивов соответствуют определению многомерного массива как одномерного, элементами которого служат массивы, размерность которых на единицу меньше, чем у исходного массива. Одномерный массив инициализируется заключенным в фигурные скобки списком начальных значений. В свою очередь, начальное значение, если оно относится к массиву, также представляет собой заключенный в фигурные скобки список начальных значений. Например, присвоить начальные значения вещественным элементам двумерного массива **a**, состоящего из трех «строк» и двух «столбцов», можно следующим образом:

```
double a[3][2]={{10,20}, {30,40}, {50,60}};
```

Эта запись эквивалентна последовательности операторов присваивания: a[0][0]=10; a[0][1]=20; a[1][0]=30; a[1][1]=40; a[2][0]=50; a[2][1]=60;. Тот же результат можно получить с одним списком инициализации:

```
double a[3][2]={10,20,30,40,50,60};
```

С помощью инициализации можно присваивать значения не всем элементам многомерного массива. Например, чтобы инициализировать только элементы первого столбца матрицы, ее можно описать так:

```
double z[4][6]={{1}, {2}, {3}, {4}};
```

Следующее описание формирует «треугольную» матрицу в целочисленном массиве из 5 строк и 4 столбцов:

```
int x[5][4]={{1}, {2,3}, {4,5,6}, {7,8,9,10} };
```

В данном примере последняя, пятая строка x[4] остается незаполненной. Первые три строки заполнены не до конца. Схема размещения элементов массива изображена на рис. 2.5.

Строка	0				1				2				3				4			
Столбец	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Значение	1	–	–	–	2	3	–	–	4	5	6	–	7	8	9	10	–	–	–	–

Рис. 2.5. *Схема «треугольного» заполнения матрицы*

2.5. Функции

Определение функций. Как уже было отмечено, каждая программа на языке Си – это совокупность функций. В определении функции указываются последовательность действий, выполняемых при обращении к функции, имя функции, тип результата (возвращаемого значения) и совокупность параметров, заменяемых при обращении к функции аргументами.

Действия, выполняемые при обращении к функции, задает ее тело – составной оператор (блок), обязательным элементом которого

служат внешние фигурные скобки { }. Имя функции, тип результата, совокупность параметров и их свойства задают заголовок функции.

В этой главе будем рассматривать функции, возвращающие значения целого или вещественного типа. Таким образом, типом результата может быть **char**, **int**, **long**, **float**, **double** или **long double**. Могут быть добавлены **signed** и **unsigned**. Если тип результата не указан, то по умолчанию предполагается целый тип **int**. Допустимы функции, не возвращающие результата. В этом случае для них должен быть использован тип **void**.

Имя неглавной функции – это произвольно выбираемый идентификатор. Имя функции не должно совпадать со служебными словами, именами библиотечных функций и с другими именами в программе. Список параметров – набор идентификаторов, отделяемых друг от друга запятыми. Спецификация параметров определяет их свойства и во многом подобна определениям и описаниям. В данной главе будем считать, что параметр может быть специфицирован либо как простая переменная целого или вещественного типа, либо как одномерный массив (целый или вещественный).

Структура определения функции:

тип_результата
имя_функции (спецификация_параметров)
{
определения_объектов;
исполняемые_операторы;
}

Пример функции:

```
double f (int a, float b, double d)
{/*тело функции*/}
```

Принципиально важным оператором тела функции является оператор возврата из функции в точку ее вызова:

return *выражение;*

или

return;

Выражение в операторе возврата задает возвращаемое функцией значение. Для функции типа **void**, не возвращающей никакого значения, выражение в операторе **return** отсутствует. В теле такой

функции оператор **return** может вообще отсутствовать. В этом случае компилятор предполагает, что оператор **return** находится в самом конце тела функции.

Применение оператора **return** допустимо и в функции **main( )**. Если программист явно не поместил в функцию **main( )** оператор **return**, то компилятор поместит его в конце текста функции **main( )**. В отличие от «неглавных» функций, откуда возврат выполняется в вызывающую функцию, выполнение оператора **return;** или **return** *выражение;* в функции **main( )** приводит к завершению программы. Управление при таком выходе передается вызывающей программе, например операционной системе, которая может анализировать значение выражения, использованного в операторе возврата.

Приведем примеры определения функций и проиллюстрируем некоторые их особенности.

Функция для вычисления объема цилиндра. Пусть у функции имеются два параметра, значение большего из которых считается радиусом цилиндра, а меньший определяет значение высоты цилиндра.

Пусть g и h – параметры; тогда объем цилиндра равен g*g*h*PI, если g>h, или g*h*h*PI, если g<h, где PI – число π.

Для решения этой задачи определим функцию:

```
float w(float g, float h)
   {
     if ( g >= h )
         return 3.14159*g*g*h;
     else
         return 3.14159*g*h*h;
   }
```

Для возврата из функции и передачи результата в точку вызова в теле функции используются два оператора return.

Функция для вычисления скалярного произведения векторов. Скалярное произведение двух векторов n-мерного линейного пространства вычисляется по формуле

$$S = \sum_{i=1}^{n} a_i b_i \ .$$

Функция для вычисления указанного произведения может быть определена следующим образом:

```
/* Скалярное произведение n-мерных векторов */
float Scalar_Product (int n, float a[ ], float b[ ])
    { int i;        /* Параметр цикла */
      float z;      /* Формируемая сумма */
      for (i=0, z=0.0; i<n; i++)
           z+=a[i]*b[i];
      return z; /* Возвращаемый результат */
    }
```

Первый параметр n специфицирован как целая переменная типа **int**. В спецификации массивов-параметров типа **float** пределы изменения индексов не указаны, что позволяет при обращении к функции использовать вместо a и b в качестве аргументов одномерные массивы такого же типа любых размеров (с любым количеством элементов). Конкретные пределы изменения их индексов задает аргумент, заменяющий параметр **int** n.

Обращение к функции и ее прототип. Как уже говорилось, для обращения к функции используется элементарное (первичное) выражение, называемое «вызов функции»:

имя_функции (список_аргументов)

Значение этого выражения – возвращаемое функцией значение (определяется в теле функции выполненным оператором **return**). Список аргументов – это список выражений, заменяющих параметры функции. Соответствие между параметрами и аргументами устанавливается по порядку их расположения в списках. Если параметров у функции нет, то не должно быть и аргументов при обращении к этой функции. Аргументы передаются из вызывающей программы в функцию по значению, то есть вычисляется значение каждого аргумента, и именно оно используется в теле функции вместо заменяемого параметра. Пример вызова определенной выше функции для вычисления объема цилиндра:

```
w(z-1.0,1e-2)
```

Стандарт языка Си предусматривает обязательное описание функции с помощью прототипа. Прототип имеет формат:

тип_результата имя_функции
(спецификация_ параметров);

Здесь спецификация параметров представляет собой список типов и, возможно, имен параметров функции.

Прототип функции схож с ее заголовком. Но имеются два существенных отличия. Во-первых, прототип всегда заканчивается признаком конца оператора (символ «;»). Во-вторых, в прототипе могут не указываться имена специфицируемых параметров. Прототип может не использоваться только в том случае, когда определение функции находится в том же файле, где размещена вызывающая ее программа, и это определение помещено в тексте выше вызывающей программы. Прототипы введенных выше функций могут быть такими:

float w(**float**, **float**);
Scalar_Product (**int** n, **float** a[ ], **float** b[ ]);

Имена параметров в прототипе функции w( ) не указаны, специфицированы только их типы.

Прототипы функций необходимо размещать наряду с определением объектов в теле функций до исполняемых операторов.

Приведем примеры программ, состоящих более чем из одной функции.

Вычисление биномиального коэффициента. Как известно,

$$C_n^m = \frac{n!}{m!(n-m)!},$$

где n ≥ m ≥ 0; n, m – целые.

Составим программу для вычисления биномиального коэффициента, в которой используем функцию для вычисления факториала:

```
#include <stdio.h>
int fact(int k) /* Вычисление факториала k!*/
{
 int j, i;        /* Вспомогательные переменные */
 for(i=1, j=1; i<=k; i++) /*Цикл вычисления*/
    j*=i;
 return j;
}   /* Конец определения функции */
/* Вычисление биномиального коэффициента: */
void main( )
{
 int n, m, nmc, nm;    /*nm - значение (n-m) */
/* nmc - значение биномиального коэффициента */
```

```
 while (1)
 {
  printf("\nВведите n=");
  scanf("%d",&n);
  printf("Введите m=");
  scanf("%d", &m);
  if (m>=0 && n>=m && n<10) break;
     printf("Ошибка! Необходимо 0<=m<=n<10");
 }
 nm=n-m;
 nmc=fact(n)/fact(m)/fact(nm);
 printf ("\n Биномиальный коэффициент=%d", nmc);
}  /* Конец основной программы */
```

В основной программе прототип функции fact( ) не нужен, так как определение функции находится в том же файле, что и функция **main( )**, вызывающая fact( ), причем определение размещено выше вызова. Пример выполнения программы:

```
Введите n=4                <Enter>
Введите m=5                <Enter>
Ошибка ! Необходимо 0<m<=n<10
Введите n=4                <Enter>
Введите m=2                <Enter>
Биномиальный коэффициент =6
```

Вычисление объема цилиндра с использованием приведенной выше функции w( ):

```
#include <stdio.h>
/* Вычисление объема цилиндра: */
void main( )
{
 float w(float, float);  /* Прототип  функции */
 float a,b;     /* Исходные данные  */
 int j;     /* Счетчик попыток ввода */
 for (j=0; j<5; j++)
  {    /* Цикл ввода данных  */
     printf("\n Введите a=");
     scanf("%f",&a);
     printf(" Введите b=");
     scanf("%f",&b);
     if ( a > 0.0 && b > 0.0 ) break;
```

```
      printf("\n Ошибка, нужно a>0 и b>0!\n");
   }
   if (j == 5)
    {
      printf("\n ОЧЕНЬ ПЛОХО вводите данные!!");
      return; /* аварийное окончание программы*/
    }
    printf("\n Объем цилиндра =%f", w(a,b));
}   /* Конец основной программы */

/*Функция для вычисления объема цилиндра: */
float w(float g, float h)
{
 if ( g >= h )
    return(3.14159*g*g*h);
   else
    return(3.14159*g*h*h);
}
```

В основной программе использован оператор **return**, прерывающий исполнение программы. Оператор **return** выполняется после цикла ввода исходных данных, если количество неудачных попыток ввода (значений a и b) равно 5. Задан прототип функции w( ), то есть задан ее прототип, что необходимо, так как она возвращает значение, отличное от **int**, и определена стандартным образом позже (ниже), чем обращение к ней. Обращение к функции w( ) использовано в качестве аргумента функции **printf( )**.

Пример выполнения программы:

```
Введите a=2.0           <Enter>
Введите b=-44.3         <Enter>
Ошибка, нужно a>0 и b>0
Введите a=2.0           <Enter>
Введите b=3.0           <Enter>
Объем цилиндра=56.548520
```

Вычисление площади треугольника. Для определения площади треугольника по формуле Герона

$$S = \sqrt{p(p-A)(p-B)(p-C)}$$

достаточно задать длины его сторон A, B, C и, вычислив полупериметр $p=(A+B+C)/2$, вычислить значение площади по формуле.

Однако для составления соответствующей программы необходима функция вычисления квадратного корня. Предположив, что такой функции в библиотеке стандартных математических функций нет, составим ее сами. В основу положим метод Ньютона:

$$x_i = (x_{i-1} + z/x_{i-1})/2, \quad i = 1, 2, \ldots$$

где z – подкоренное выражение; x_0 – начальное приближение.

Вычисления будем проводить с фиксированной относительной точностью ε. Для простоты условием прекращения счета будет выполнение неравенства $\left|\dfrac{x_{i-1} - x_i}{x_i}\right| < \varepsilon$. Для вычисления абсолютного значения введем еще одну функцию с именем abs( ) (хотя такая функция, так же как функция для вычисления квадратного корня, есть в стандартной библиотеке). Программа может быть такой:

```
/* Вычисление площади треугольника */
#include <stdio.h>   /*Для средств ввода-вывода*/
#include <stdlib.h> /* Для функции exit( ) */
void main( )
{
 float a,b,c,p,s;
 float sqr(float); /* Прототип функции */
 printf("\n Сторона a= ");
 scanf("%f",&a);
 printf("Сторона b= ");
 scanf("%f",&b);
 printf("Сторона c= ");
 scanf("%f",&c);
 if(a+b <= c || a+c <= b || b+c <= a)
   {
    printf("\n Треугольник построить нельзя!");
    return;    /* Аварийное окончание работы */
   }
p=(a+b+c)/2;         /* Полупериметр */
s=sqr(p*(p-a)*(p-b)*(p-c));
printf("Площадь треугольника: %f",s);
}     /* Конец основной программы */

/* Определение функции вычисления квадратного корня */
float sqr(float x)
{    /* x-подкоренное выражение */
      /*Прототип функции вычисления модуля: */
```

```
  float abs(float);
  double r,q;
  const double REL=0.00001;
              /* REL-относительная точность */
if (x < 0.0)
   {
    printf("\n Отрицательное подкоренное"
              " выражение");
    exit(1);   /* Аварийное окончание программы */
   }
if (x == 0.0) return x ;
/* Итерации вычисления корня: */
r=x;       /* r - очередное приближение */
do {
      q=r; /* q - предыдущее приближение */
      r=(q+x/q)/2;
     }
while (abs((r-q)/r) > REL);
return r;
}    /* Конец определения функции sqr   */
/* Определение функции */
/* для получения абсолютного значения: */
float abs(float z)
{
  if(z > 0) return z;
  else return(-z);
}    /* Конец определения функции abs */
```

В программе используются три функции. Основная функция **main( )** вызывает функцию sqr( ), прототип которой размещен выше вызова. Функция abs( ) не описана в основной программе, так как здесь к ней нет явных обращений. Функция abs( ) вызывается из функции sqr( ), поэтому ее прототип помещен в тело функции sqr( ).

В процессе выполнения программы может возникнуть аварийная ситуация, когда введены такие значения переменных a, b, c, при которых они не могут быть длинами сторон одного треугольника. При обнаружении подобной ситуации выдается предупреждающее сообщение «Треугольник построить нельзя!», и основная функция **main( )** завершается оператором **return**. В функции sqr( ) также есть защита от неверных исходных данных. В случае отрицательного значения подкоренного выражения (x) нужно не только прервать вычисление значения корня, но и завершить выполнение програм-

мы с соответствующим предупреждающим сообщением. Оператор **return** для этого неудобен, так как позволяет выйти только из той функции, в которой он выполнен. Поэтому вместо **return**; при отрицательном значении x в функции sqr( ) вызывается стандартная библиотечная функция **exit( )**, прекращающая выполнение программы. Прототип (описание) функции **exit( )** находится в заголовочном файле **stdlib.h**, который включается в начало текста программы препроцессорной директивой.

Пример результатов выполнения программы:

```
Сторона a=2.0    <Enter>
Сторона b=3.0    <Enter>
Сторона c=4.0    <Enter>
Площадь треугольника: 2.904737
```

Скалярное произведение векторов. Выше была определена функция Scalar_Product( ) для вычисления скалярного произведения векторов, в которой параметрами являлись массивы. Следующая программа использует эту функцию:

```
/* Скалярное произведение векторов */
#include <stdio.h>
#define MAX_INDEX  5
void main( )
{
 /* Прототип функции: */
 float Scalar_Product(int, float[ ], float[ ]);
 int n,i;
 float x[MAX_INDEX],y[MAX_INDEX];
 printf("\n Размерность векторов n= ");
 scanf("%d",&n);
 if(n < 1 || n >MAX_INDEX)
    {
     printf("\n Ошибка в данных!");
     return;    /* Аварийное завершение */
    }
 printf("Введите %d координ. x: ",n);
 for (i=0; i<n; i++)
     scanf("%f",&x[i]);
 printf("Введите %d координ. y: ",n);
 for (i=0; i<n; i++)
     scanf("%f",&y[i]);
 printf("\n Результат: %7.3f", Scalar_Product(n,x,y));
```

```
}
/* Определение функции scalar: */
float Scalar_Product(int n, float a[],float b[])
/* Скалярное произведение n-мерных векторов */
/* n - размерность пространства векторов */
/* a[ ],b[ ] - массивы координат векторов */
{ int i;          /* Параметр цикла */
   double z;       /* Формируемая сумма */
   for (i=0,z=0.0; i < n; i++)
      z += a[i]*b[i];
   return z; /* Возвращаемый результат */
}
```

В начале программы с помощью **#define** введена препроцессорная константа MAX_INDEX. Далее определены массивы, у которых пределы изменения индексов заданы на препроцессорном уровне. Именно эти пределы проверяются после ввода размерности векторов (n). В теле функции **main( )** приведен прототип функции Scalar_Product( ). Обратите внимание, что в прототипе отсутствуют имена параметров. Тот факт, что два параметра являются одномерными массивами, отображен спецификацией **float**[].

Результаты выполнения программы:

```
Размерность векторов n=2        <Enter>
Введите 2 координ. x:   1   3.1 <Enter>
Введите 2 координ. y:   1   2.1 <Enter>
Результат:   7.510
```

Другая попытка выполнить программу:

```
Размерность векторов n=0
Ошибка в данных!
```

Диаметр множества точек. Как еще один пример использования функций с массивами в качестве параметров рассмотрим программу определения диаметра множества точек в многомерном евклидовом пространстве. Напомним, что диаметром называется максимальное расстояние между точками множества, а расстояние в евклидовом пространстве между точками $x = \{ x_i \}$; $y = \{ y_i \}$, $i = 1, ..., n$, определяется как

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

Введем ограничения на размерность пространства: N_MAX<=10 и количество точек K_MAX<=100. Текст программы может быть таким:

```
#include <stdio.h>
#include <math.h>      /* для функции sqrt( ) */
/* Функция для вычисления расстояния между двумя точками */
float distance(float x[ ], float y[ ], int n)
{
 int i;
 float r,s=0.0;
 for(i=0;i<n;i++)
     {
      r=y[i]-x[i];
      s+=r*r;
     }
 s=sqrt(s);
 return s;
}
#define K_MAX 100
#define N_MAX 10
void main( )
{
   float dist, dist_max,d;
   int i, j, i_max, m_max, n, k, m;
   float a[K_MAX][N_MAX];
      /* a[][] - Массив фиксированных размеров */
   while(1)
   {
    printf("\n Количество точек k=");
    scanf("%d", &k);
    printf("Размерность пространства n=");
    scanf("%d", &n);
    if(k>0 && k<=K_MAX && n>0 && n<=N_MAX) break;
      printf("ОШИБКА В ДАННЫХ!");
   }
   for(i=0;i<k;i++)
     { /* Цикл ввода координат точек   */
       printf("Введите %d координ. "
              "точки %d:\n",n,i+1);
       for(j=0;j<n;j++)
       /* Цикл ввода координат одной точки */
          scanf("%f",&a[i][j]);
     }
```

```
   dist_max=0.0;
   i_max=0;
   m_max=0;
   for(i=0;i<k-1;i++)
      { /* Цикл сравнения точек */
         for(m=i+1;m<k;m++)
            {
             dist=distance(a[i],a[m],n);
             if(dist>dist_max)
                {
                 dist_max=dist;
                 i_max=i;
                 m_max=m;
                }
            }
      } /* Конец цикла по i */
   printf("Результат:\nДиаметр=%f",dist_max);
   printf("\n Номера точек : %d,%d",
          i_max+1,m_max+1);
}
```

Результаты выполнения программы:

```
Количество точек к=4              <Enter>
Размерность пространства n=3      <Enter>
Введите 3 координ. точки 1:
1       1       1                 <Enter>
Введите 3 координ. точки 2:
-1      -1      -1                <Enter>
Введите 3 координ. точки 3:
2       2       2                 <Enter>
Введите 3 координ. точки 4:
-2      -2      -2                <Enter>
Результат:
Диаметр = 6.928203
Номера точек: 3, 4
```

В программе особый интерес представляет обращение к функции distance( ), где в качестве аргументов используются индексированные элементы a[i], a[m]. Каждый из них, по определению, есть одномерный массив из n элементов, что и учитывается в теле функции. Для задания размеров массива a[ ][ ] и предельных значений переменных k и n используются препроцессорные константы K_MAX и

N_MAX. Их нельзя определить как переменные, то есть ошибочной будет последовательность:

```
int K_MAX=100; N_MAX=10;
float a[K_MAX][N_MAX];
```

При определении массивов их размеры можно задавать только с помощью константных выражений.

2.6. Переключатели

Основным средством для организации мультиветвления служит оператор-переключатель, формат которого имеет вид:

switch(*выражение*)
{ **case** *константа1: операторы_1;*
case *константа2: операторы_2;*
default: *операторы*;
}

В этом операторе используются три служебных слова: **switch**, **case**, **default**. Первое из них идентифицирует собственно оператор-переключатель. Служебное слово **case** с последующей константой является в некотором смысле меткой. Константы могут быть целыми или символьными и все должны быть различными (чтобы метки были различимы). Служебное слово **default** также обозначает отдельную метку. При выполнении оператора (рис. 2.6а) вычисляется выражение, записанное после **switch**, и его значение последовательно сравнивается с константами, которые помещены вслед за **case**. При первом же совпадении выполняются операторы, помеченные данной меткой. Если выполненные операторы не предусматривают какого-либо перехода (то есть среди них нет ни **goto**, ни **return**, ни **exit( )**, ни **break**), то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель.

Операторы вслед за **default** выполняются, если значение выражения в скобках после **switch** не совпало ни с одной константой после **case**. Метка **default** может в переключателе отсутствовать. В этом случае при несовпадении значения выражения с константами переключатель не выполняет никаких действий. Операторы, помеченные меткой **default**, не обязательно находятся в конце (после других вариантов переключателя). Уточним, что **default** и «**case** *константа*»

не являются метками в обычном смысле. К ним, например, нельзя перейти с помощью оператора **goto**.

На рис. 2.6 приведены схемы переключателя (рис. 2.6*а*) и оператора альтернативного выбора или селектора (рис. 2.6 *б*), отсутствующего в языке Си.

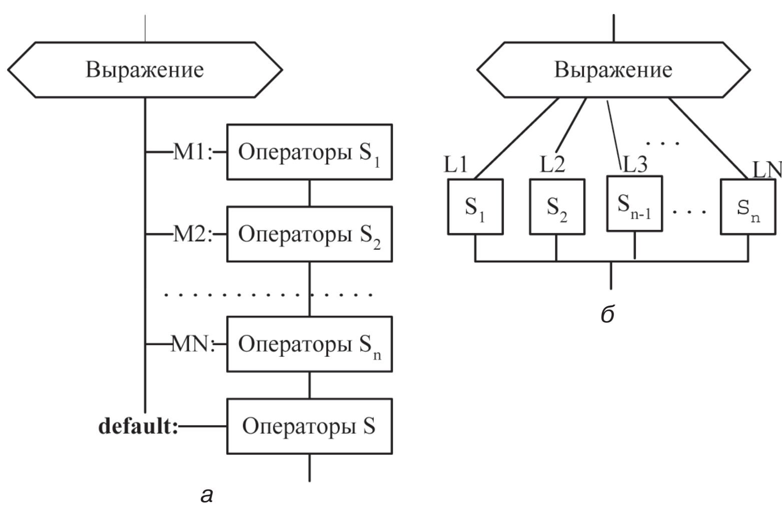


Рис. 2.6. *Переключатель (а) и альтернативный выбор (б):*
а – если выражение равно MK,
то выполняются операторы S_k, $S_{k+1,\ldots,}$ *S, S;*
б – если выражение равно LK, то выполняются только операторы S_k

На рис. 2.7 изображена схема альтернативного выбора, реализованная при помощи переключателя и операторов перехода (go to, return, break), введенных в S_1, S_2, ..., S_n.

Для иллюстрации работы переключателя рассмотрим программу, которая читает десятичную цифру и выводит на экран ее название:

```
#include <stdio.h>
void main( )
{
 int i;
 while(1)
   {
    printf("\n Введите десятичную цифру: ");
    scanf("%d",&i);
```

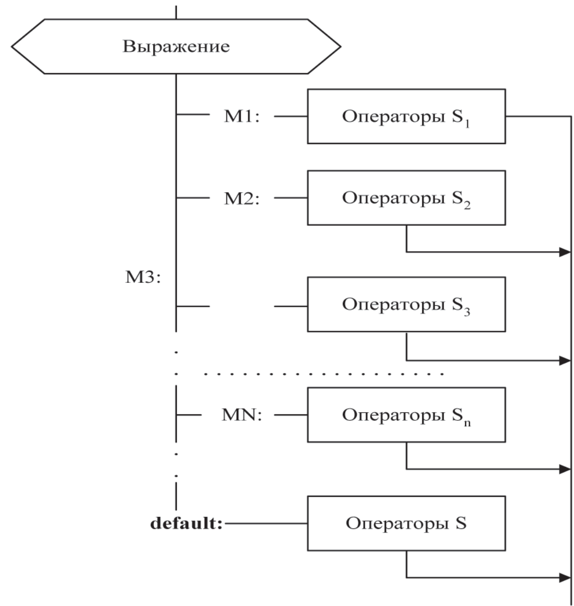


Рис. 2.7. *Альтернативный выбор с использованием переключателя. В число операторов каждой группы S_k добавлен оператор выхода из переключателя*

```
printf("\t\t");
switch( i )
{
 case 1: printf("%d - один \n",i);
 break;
 case 2: printf("%d - два \n",i);
 break;
 case 3: printf("%d - три \n",i);
 break;
 case 4: printf("%d - четыре \n",i);
 break;
 case 5: printf("%d - пять \n",i);
 break;
 case 6: printf("%d - шесть \n",i);
 break;
 case 7: printf("%d - семь \n",i);
 break;
 case 8: printf("%d - восемь \n",i);
 break;
```

```
      case 9: printf("%d - девять \n",i);
      break;
      case 0: printf("%d - нуль \n",i);
      break;
      default: printf("%d - это не цифра! \n",i);
               return;
    } /* Конец переключателя */
  } /* Конец цикла */
}
```

Пример результатов выполнения программы:

```
Введите десятичную цифру: 3            <Enter>
                   3  - три
Введите десятичную цифру: 8            <Enter>
                   8  - восемь
Введите десятичную цифру: -7           <Enter>
                   -7 - это не цифра!
```

Программа прекращает выполнение, как только будет введен символ, отличный от цифры. Завершение программы обеспечивает оператор **return**; который в данном случае передает управление операционной системе, так как выполняет выход из функции **main**( ).

Переключатель вместе с набором операторов **break** реализует в этой программе альтернативный выбор (см. рис. 2.7). Если удалить все операторы **break**, то работа переключателя в этой программе будет соответствовать схеме рис. 2.6*а*.

Несколько «меток» **case** с разными значениями констант могут помечать один оператор внутри переключателя, что позволяет еще больше разнообразить схемы построения операторов **switch**.

Контрольные вопросы

1. Можно ли написать завершенную программу без функции **main**()?
2. Какое расширение должно иметь имя файла с текстом программы на языке Си?
3. Назовите обязательные этапы обработки, которые проходит исходная программа, подготовленная на языке Си в виде текстового файла.
4. Объясните роль препроцессора при подготовке программы на языке Си.

5. Какие требования существуют в отношении препроцессорных директив?
6. Каким образом включают в программу прототипы библиотечных функций?
7. Какова структура простой программы на языке Си?
8. Для чего служат библиотечные функции?
9. Назовите состав тела функции.
10. Для чего служат определения и описания?
11. Как можно задать комментарий в тексте программы?
12. Перечислите группы операторов языка Си.
13. Какие операторы применяются для управления работой программы?
14. Чем отличается блок от составного оператора?
15. В каких местах кода программы может быть поставлена метка?
16. Как обозначается пустой оператор?
17. Перечислите операторы циклов языка Си.
18. Перечислите операторы ветвлений языка Си.
19. В чем сходства и различия операторов **break** и **continue**?
20. Что представляет собой инициализация объекта (например, массива)?
21. Значения каких типов могут возвращать функции?
22. Как задается имя неглавной функции?
23. Дайте определение списка параметров.
24. С помощью какого оператора производится возврат из функции в точку ее вызова?
25. Что такое прототип функции?
26. Где размещают прототипы функций?
27. Каким образом организуется мультиветвление в программе на языке Си?
28. Чем отличаются действия переключателя и оператора альтернативного выбора?

Глава 3
ПРЕПРОЦЕССОРНЫЕ СРЕДСТВА

В предыдущих главах мы познакомились с некоторыми базовыми понятиями и основными средствами (может быть, не самыми эффективными) программирования языка Си. С данной главы мы начнем подробное изучение тех особенностей и возможностей, которыми Си отличается от других языков программирования и которые принесли ему заслуженную популярность и любовь профессиональных программистов.

Несколько нетрадиционно для пособий по языку Си начнем дальнейшее изложение материала с возможностей препроцессора. Это позволит в следующих главах продемонстрировать эффективность препроцессорных средств и их применимость при решении разнородных задач. Откладывать, как часто принято, изучение препроцессора на конец курса по языку Си, по нашему мнению, не совсем удачно. Отметим, что препроцессор обрабатывает почти любые тексты, а не только тексты программ на языке Си. Обработка программ – это основная задача препроцессора, однако он может преобразовывать произвольные тексты, и этой возможностью программисту не следует пренебрегать в своей работе.

Итак, на входе препроцессора – текст с препроцессорными директивами, на выходе препроцессора – текст без препроцессорных директив (см. рис. 1.1).

3.1. Стадии и директивы препроцессорной обработки

В интегрированную среду подготовки программ на Си или в компилятор языка как обязательный компонент входит препроцессор. Назначение препроцессора – обработка исходного текста программы до ее компиляции (см. рис. 2.1).

Стадии препроцессорной обработки. Препроцессорная обработка включает несколько стадий, выполняемых последовательно. Конкретная реализация может объединять несколько стадий, но результат должен быть таким, как если бы они выполнялись в следующем порядке:

- все системно-зависимые обозначения (например, системно-зависимый индикатор конца строки) перекодируются в стандартные коды;
- каждая пара из символов '\' и «конец строки» вместе с пробелами между ними убираются, и тем самым следующая строка исходного текста присоединяется к строке, в которой находилась эта пара символов;
- в тексте (точнее, в тексте каждой отдельной строки) распознаются директивы и лексемы препроцессора, а каждый комментарий заменяется одним символом пустого промежутка;
- выполняются директивы препроцессора и производятся макроподстановки;
- эскейп-последовательности в символьных константах и символьных строках, например '\n' или '\xF2', заменяются на их эквиваленты (на соответствующие числовые коды);
- смежные символьные строки (строковые константы) конкатенируются, то есть соединяются в одну строку;
- каждая препроцессорная лексема преобразуется в лексему языка Си.

Поясним, что понимается под препроцессорными лексемами или лексемами препроцессора (preprocessing token). К ним относятся символьные константы, имена включаемых файлов, идентификаторы, знаки операций, препроцессорные числа, знаки препинания, строковые константы (строки) и любые символы, отличные от пробела. Можно сказать, что к лексемам препроцессора относятся лексемы языка Си, имена файлов и символы, не определенные иным способом.

Знакомство с перечисленными стадиями препроцессорной обработки объясняет, как реализуются некоторые правила синтаксиса языка. Например, становится понятным смысл утверждений: «каждая символьная строка может быть перенесена в файле на следующую строку, если использовать символ '\'» или «две символьные строки, записанные рядом, воспринимаются как одна строка». Отметим, что после «склеивания» строк в соответствии с приведенными

правилами каждая полученная строка обрабатывается препроцессором отдельно.

Рассмотрим подробно стадию обработки директив препроцессора. При ее выполнении возможны следующие действия:

- замена идентификаторов (обозначений) заранее подготовленными последовательностями символов;
- включение в программу текстов из указанных файлов;
- исключение из программы отдельных частей ее текста (условная компиляция);
- макроподстановка, то есть замена обозначения параметризованным текстом, формируемым препроцессором с учетом конкретных параметров (аргументов).

Директивы препроцессора. Для управления препроцессором, то есть для задания нужных действий, используются команды (директивы) препроцессора, каждая из которых помещается на отдельной строке и начинается с символа **#**.

Обобщенный формат директивы препроцессора:

#имя_директивы лексемы_препроцессора

Перед символом '#' и после него в директиве разрешены пробелы. Пробелы также разрешены перед *лексемами_препроцессора*, между ними и после них. Окончанием препроцессорной директивы служит конец текстовой строки (при наличии символа '\', обозначающего перенос строки, окончанием препроцессорной директивы будет признак конца следующей строки текста).

Определены следующие препроцессорные директивы:

- **#define** – определение макроса или препроцессорного идентификатора;
- **#include** – включение текста из файла;
- **#undef** – отмена определения макроса или идентификатора (препроцессорного);
- **#if** – проверка условия-выражения;
- **#ifdef** – проверка определенности идентификатора;
- **#ifndef** – проверка неопределенности идентификатора;
- **#else** – начало альтернативной ветви для #**if**;
- **#endif** – окончание условной директивы #**if**;
- #**elif** – составная директива #**else**/#**if**;
- **#line** – смена номера следующей ниже строки;
- **#error** – формирование текста сообщения об ошибке трансляции;

- **#pragma** – действия, предусмотренные реализацией;
- **#** – пустая директива.

Кроме препроцессорных директив, имеются три препроцессорные операции, которые будут подробно рассмотрены вместе с командой #**define**:

- **defined** – проверка истинности операнда;
- **##** – конкатенация препроцессорных лексем;
- **#** – преобразование операнда в строку символов.

Директива **#define** имеет несколько модификаций. Они предусматривают определение макросов и препроцессорных идентификаторов, каждому из которых ставится в соответствие некоторая символьная последовательность. В последующем тексте программы препроцессорные идентификаторы заменяются на заранее запланированные последовательности символов. Примеры определения констант с помощью **#define** приведены в главе 1.

Директива **#include** позволяет включать в текст программы текст из указанного файла.

Директива **#undef** отменяет действие директивы **#define**, которая определила до этого имя препроцессорного идентификатора.

Директива **#if** и ее модификации **#ifdef**, **#ifndef** совместно с директивами **#else**, **#endif**, **#elif** позволяют организовать условную обработку текста программы. При использовании этих средств компилируется не весь текст, а только те его части, которые выделены с помощью перечисленных директив.

Директива **#line** позволяет управлять нумерацией строк в файле с программой. Имя файла и желаемый начальный номер строки указываются непосредственно в директиве **#line** (подробнее см. §3.6).

Директива **#error** позволяет задать текст диагностического сообщения, которое выводится при возникновении ошибок.

Директива **#pragma** вызывает действия, зависящие от реализации, то есть запланированные авторами компилятора.

Директива **#** ничего не вызывает, так как является пустой директивой, то есть не дает никакого эффекта и всегда игнорируется.

Рассмотрим возможности перечисленных директив и препроцессорных операций при решении типичных задач, поручаемых препроцессору. Одновременно на примерах поясним, что понимается под *препроцессорными лексемами* в обобщенном формате препроцессорных директив.

3.2. Замены в тексте

Директива #define. Как уже иллюстрировалось на примере именованных констант (§1.3 и 2.1), для замены выбранного программистом идентификатора заранее подготовленной последовательностью символов используется директива (обратите внимание на пробелы):

#define *идентификатор строка_замещения*

Директива может размещаться в любом месте обрабатываемого текста, а ее действие в обычном случае распространяется от точки размещения до конца текста. Директива, во-первых, определяет идентификатор как препроцессорный. В результате работы препроцессора вхождения идентификатора, определенного командой **#define**, в тексте программы заменяются строкой замещения, окончанием которой обычно служит признак конца той «физической» строки, где размещена команда **#define**. Символы пробелов, помещенные в начале и в конце строки замещения, в подстановке не используются. Например:

Исходный текст	Результат препроцессорной обработки
`#define begin {`	
`#define end }`	
`void main( )`	`void main( )`
`begin`	`{`
операторы	*операторы*
`end`	`}`

В данном случае программист решил использовать в качестве операторных скобок идентификаторы **begin**, **end**. До компиляции препроцессор заменяет все вхождения этих идентификаторов стандартными скобками **{** и **}**. Соответствующие указания программист дал препроцессору с помощью директив **#define**.

Цепочка подстановок. Если в *строке_замещения* команды **#define** в качестве отдельной лексемы встречается препроцессорный идентификатор, ранее определенный другой директивой **#define**, то выполняется цепочка последовательных подстановок. В качестве примера рассмотрим, как можно определить диапазон (RANGE) возможных значений любой целой переменной типа **int** в следующей программе:

```
#include <limits.h>
#define RANGE ((INT_MAX) - (INT_MIN)+1)
. . . .
/*RANGE - диапазон значений для int */
```

```
. . . .
int RANGE_T = RANGE/8;
. . . .
```

Препроцессор последовательно, строку за строкой, просматривает текст и, обнаружив директиву **#include <limits.h>**, вставляет текст из файла **limits.h**. Там определены константы **INT_MAX** (предельное максимальное значение целых величин), **INT_MIN** (предельное минимальное значение целых величин). Тем самым программа принимает, например, такой вид:

```
. . .
#define INT_MAX   32767
#define INT_MIN   -32768
. . .
#define RANGE    ((INT_MAX)-(INT_MIN)+1).
. . .
/*RANGE - диапазон значений для int*/
. . .
int RANGE_T = RANGE/8;
. . .
```

Обратите внимание, что директива **#include** исчезла из программы, но ее заменил соответствующий текст.

Обнаружив в тексте (добытом из файла **limits.h**) директивы **#define**..., препроцессор выполняет соответствующие подстановки, и программа принимает вид:

```
. . .
#define RANGE ((32767)-(-32768)+1)
/*RANGE - диапазон значений для int*/
. . .
int RANGE_T = RANGE/8;
. . .
```

Подстановки изменили строку замещения препроцессорного идентификатора RANGE в директиве **#define**, размещенной ниже, чем текст, включенный из файла **limits.h**. «Продвигаясь» по тексту программы, препроцессор встречает препроцессорный идентификатор RANGE и выполняет подстановку. Текст программы приобретает следующий вид:

```
. . .
/*RANGE - диапазон значений для int*/
. . .
int RANGE_T = ((32767)-(-32768)+1)/8;
. . .
```

Теперь все директивы **#define** удалены из текста. Получен текст, пригодный для компиляции, то есть создана «единица трансляции». Подстановка строки замещения вместо идентификатора RANGE выполнена в выражении RANGE/8, однако внутри комментария идентификатор RANGE остался без изменений и не изменился идентификатор RANGE_T.

Этот пример иллюстрирует выполнение «цепочки» подстановок и ограничения на замены: замены не выполняются внутри комментариев, внутри строковых констант, внутри символьных констант и внутри идентификаторов (не может измениться часть идентификатора). Например, RANGE_T остался без изменений. Для еще одной иллюстрации перечисленных ограничений рассмотрим такой фрагмент программы:

```
#define  n  24
. . .
char c = '\n'; /* Символьная константа*/
            /* \n - эскейп-последовательность:*/
. . . "\n Строковая константа". . .
c='n'>'\n'?'n':'\n';
int  k=n;
```

В ходе препроцессорной обработки этого текста замена n на 24 будет выполнена только один раз в последнем определении, которое примет вид:

```
int  k=24;
```

Все остальные вхождения символа n в текст программы препроцессор просто «не заметит».

Вернемся к формату директивы **#define**.

Если *строка_замещения* оказывается слишком длинной, то, как уже говорилось, ее можно продолжить в следующей строке текста. Для этого в конце продолжаемой строки помещается символ '\'. В ходе одной из стадий препроцессорной обработки этот символ

вместе с последующим символом конца строки будет удален из текста программы. Пример:

```
#define STRING "\n Game Over! - \
Игра закончена!"
...
printf(STRING);
```

На экран будет выведено:

```
Game Over! - Игра закончена!
```

С помощью команды **#define** удобно выполнять настройку программы. Например, если в программе требуется работать с массивами, то их размеры можно явно определять на этапе препроцессорной обработки:

Исходный текст	**Результат препроцессорной обработки**
`#define K 40`	
`void main( )`	`void main( )`
`{`	`{`
`int M[K][K];`	`int M[40][40];`
`float A[2*K+1],`	`float A[2*40+1],`
`float B[K+3][K-3];`	`float B[40+3][40-3];`
`...`	`...`

При таком описании очень легко изменять предельные размеры сразу всех массивов, изменив только одну константу (строку замещения) в директиве **#define**.

Предусмотренные директивой **#define** препроцессорные замены не выполняются внутри строк, символьных констант и комментариев, то есть не распространяются на тексты, ограниченные кавычками («), апострофами (') и разделителями (/*, */). В то же время строка замещения может содержать перечисленные ограничители, например как это было в замене препроцессорного идентификатора STRING.

Если в программе нужно часто печатать или выводить на экран дисплея значение какой-либо переменной и, кроме того, снабжать эту печать одним и тем же пояснительным текстом, то удобно ввести сокращенное обозначение оператора печати, например:

```
#define PK printf("\n Номер элемента=%d.", N);
```

После этой директивы использование в программе оператора **PK;** будет эквивалентно (по результату) оператору из строки замещения. Например, последовательность операторов

```
int N = 4;
PK;
```

приведет к выводу такого текста:

```
Номер элемента=4.
```

Если в строку замещения входит идентификатор, определенный в другой команде **#define**, то в строке замещения выполняется следующая замена (цепочка подстановок). Например, программа, содержащая команды:

```
#define K 50
#define PE printf ("\n Число элементов K=%d",K);
...
PE;
...
```

выведет на экран такой текст:

```
Число элементов K=50
```

Обратите внимание, что идентификатор K внутри строки замещения, обрамленной кавычками ("), не заменен на 50.

Строку замещения, связанную с конкретным препроцессорным идентификатором, можно сменить, приписав уже определенному идентификатору новое значение другой командой **#define**:

```
#define M 16
    /* Идентификатор M определен как 16 */
#define M 'C'
    /* M определен как символьная константа 'C' */
#define M "C"
    /* M определен как символьная строка */
    /* с двумя элементами: 'C' и '\0' */
```

Замены в тексте можно отменять с помощью команды

#undef *идентификатор*

После выполнения такой директивы идентификатор для препроцессора становится неопределенным, и его можно определять повторно. Например:

```
#define M 16
#undef M
#define M 'C'
#undef M
#define M "C"
```

Директиву **#undef** удобно использовать при разработке больших программ, когда они собираются из отдельных «кусков текста», написанных в разное время или разными программистами. В этом случае могут встретиться одинаковые обозначения разных объектов. Чтобы не изменять исходных файлов, включаемый текст можно «обрамлять» подходящими директивами **#define**, **#undef** и тем самым устранять возможные ошибки. Приведем пример:

```
...
   A = 10;    / Основной текст */
...
#define A X
...
   A = 5;     / Включенный текст */
...
#undef A
...
   B = A;     / Основной текст */
...
```

При выполнении программы переменная B примет значение 10, несмотря на наличие оператора присваивания A = 5; во включенном тексте.

3.3. Включение текстов из файлов

Для включения текста из файла, как мы уже неоднократно показывали, используется команда **#include**, имеющая три формы записи:

#include < *имя_файла* **>** /* Имя в угловых скобках */
#include *«имя_файла»* /* Имя в кавычках */
#include *имя_макроса*
/* Макрос, расширяемый до обозначения файла*/

где *имя_макроса* – это введенный директивой **#define** препроцессорный идентификатор либо макрос, при замене которого после конечного числа подстановок будет получена последовательность символов <*имя_файла*> либо «*имя_файла*». (О макросах см. ниже, в §3.5.)

До сих пор мы в программах и примерах фрагментов программ использовали только первую форму команды **#include**. Существует правило, что если *имя_файла* – в угловых скобках, то препроцессор разыскивает файл в стандартных системных каталогах. Если *имя_файла* заключено в кавычки, то вначале препроцессор просматривает текущий каталог пользователя и только затем обращается к просмотру стандартных системных каталогов.

Начиная работать с языком Си, пользователь сразу же сталкивается с необходимостью использования в программах средств ввода-вывода. Для этого в начале текста программы помещают директиву:

#include <stdio.h>

Выполняя эту директиву, препроцессор включает в программу средства связи с библиотекой ввода-вывода. Поиск файла **stdio.h** ведется в стандартных системных каталогах.

По принятому соглашению суффикс **.h** приписывается тем файлам, которые содержат прототипы библиотечных функций, а также определения и описания типов и констант, используемых при работе с библиотеками компилятора. Эти файлы в литературе по языку Си принято называть заголовочными.

Кроме такого в некоторой степени стандартного файла, каким является **stdio.h**, в заголовок программы могут быть включены любые другие файлы (стандартные или подготовленные специально). Перечень обозначений заголовочных файлов для работы с библиотеками компилятора утвержден стандартом языка. Ниже приведены названия этих файлов, а также краткие сведения о тех описаниях и определениях, которые в них включены. Большинство описаний – прототипы стандартных функций, а определены в основном константы (например, **NULL**, **EOF**), необходимые для работы с библиотечными функциями. Все имена, определенные в стандартных заголовочных файлах, являются зарезервированными именами:

- **assert.h** – диагностика программ;
- **ctype.h** – преобразование и проверка символов;
- **errno.h** – проверка ошибок;
- **float.h** – работа с вещественными данными;
- **limits.h** – предельные значения целочисленных данных;

- **locate.h** – поддержка национальной среды;
- **math.h** – математические вычисления;
- **setjump.h** – возможности нелокальных переходов;
- **signal.h** – обработка исключительных ситуаций;
- **stdarg.h** – поддержка переменного числа параметров;
- **stddef.h** – дополнительные определения;
- **stdio.h** – средства ввода-вывода;
- **stdlib.h** – функции общего назначения (работа с памятью);
- **string.h** – работа со строками символов;
- **time.h** – определение дат и времени.

В конкретных реализациях количество и наименования заголовочных файлов могут быть и другими.

Стандартные заголовочные файлы могут быть нечаянно или нарочно включены в текст программы в любом порядке и по нескольку раз без отрицательных побочных эффектов. Однако действие включаемого заголовочного файла распространяется на текст программы только от места размещения директивы **#include** и до конца текстового файла (и всех включаемых в программу текстов).

Заголовочные нестандартные файлы оказываются весьма эффективным средством при модульной разработке крупных программ, когда связь между модулями, размещаемыми в разных файлах, реализуется не только с помощью параметров, но и через внешние объекты, глобальные для нескольких или всех модулей. Описания таких внешних объектов (переменных, массивов, структур и т. п.) и прототипы функций помещаются в одном файле, который с помощью директив **#include** включается во все модули, где необходимы внешние объекты. В тот же файл можно включить и директиву подключения файла с описаниями библиотеки функций ввода-вывода. Заголовочный файл может быть, например, таким:

```
#include <stdio.h> /* Включение средств обмена */
/* Целые внешние переменные */
extern int ii, jj, ll;
/* Вещественные внешние переменные */
extern float aa, bb;
```

Если в программе используется несколько функций, то часто удобно текст каждой из них хранить в отдельном файле. При подготовке программы в виде одного модуля программист включает в нее тексты всех функций с помощью команд **#include**.

3.4. Условная компиляция

Директивы ветвлений. Условная компиляция обеспечивается в языке Си следующим набором директив, которые, не точно соответствуя названию, управляют не компиляцией, а препроцессорной обработкой текста:

#if *целочисленное_константное_выражение*
#ifdef *идентификатор*
#ifndef *идентификатор*
#else
#endif
#elif

Первые три директивы выполняют проверку условий, две следующие – позволяют определить диапазон действия проверяемого условия. (Директиву **#elif** рассмотрим несколько позже.) Общая структура применения директив условной компиляции такова:

#if ...
текст_1
#else
текст_2
#endif

Конструкция **#else** *текст_2* необязательна. *Текст_1* включается в компилируемый текст только при истинности проверяемого условия (обозначено многоточием после **#if**). Если условие ложно, то при наличии директивы **#else** на компиляцию передается *текст_2*. Если директива **#else** и *текст_2* отсутствуют, то весь текст от **#if** до **#endif** при ложном условии опускается. Различие между формами команд **#if** состоит в следующем.

В первой из перечисленных директив

#if *целочисленное_константное_выражение*

проверяется значение константного выражения, в которое могут входить целые константы и идентификаторы. Идентификаторы могут быть определены на препроцессорном уровне, и тогда их значение определяется подстановками. В противном случае считается, что идентификаторы имеют нулевые значения. Если константное выражение отлично от нуля, то считается, что проверяемое условие истинно. Например, в результате выполнения директив:

#if 5+4
текст_1
#endif

текст_1 всегда будет включен в компилируемую программу.

В директиве

#ifdef *идентификатор*

проверяется, определен ли с помощью директивы **#define** к текущему моменту идентификатор, помещенный после **#ifdef**. Если идентификатор определен, то есть является препроцессорным, то *текст_1* используется компилятором.

В директиве

#ifndef *идентификатор*

проверяется обратное условие – истинным считается неопределенность идентификатора, то есть тот случай, когда идентификатор не был использован в команде **#define** или его определение было отменено командой **#undef**.

Условную компиляцию удобно применять при отладке программ для включения или исключения средств вывода контрольных сообщений. Например:

```
#define DEBUG
...
#ifdef DEBUG
   printf ( " Отладочная печать ");
#endif
```

Таких вызовов функции **printf( )**, появляющихся в программе в зависимости от определенности идентификатора **DEBUG**, может быть несколько, и, убрав либо поместив в скобки комментария /*...*/ директиву **#define DEBUG**, сразу же отключаем все отладочные средства.

Файлы, предназначенные для препроцессорного включения в программу, обычно снабжают защитой от повторного включения. Такое повторное включение может произойти, если несколько файлов, в каждом из которых, в свою очередь, запланировано препроцессорное включение одного и того же файла, объединяются в общий текст программы. Например, подобными средствами защиты снабжены все заголовочные файлы стандартной библиотеки. Схема защиты от повторного включения может быть такой:

```
/* Файл с именем filename */
/* Проверка определенности _FILE_NAME */
#ifndef  _FILE_NAME
.../* Включаемый текст файла filename */
   /* Определение _FILE_NAME */
   #define  _FILE_NAME
#endif
```

Здесь _FILE_NAME – зарезервированный для файла filename препроцессорный идентификатор, который желательно не использовать в других текстах программы.

Для организации мультиветвлений во время обработки препроцессором исходного текста программы введена директива

#elif *целочисленное_константное_выражение*

Требования к *целочисленному_константному_выражению* те же, что и в директиве **#if**.

Структура исходного текста с применением этой директивы такова:

#if *условие*
  текст_для_if
#elif *выражение_1*
  текст_1
#elif *выражение_2*
  текст_2
...
#else
  текст_для_случая_else
#endif

Препроцессор проверяет вначале *условие* в директиве **#if**. Если оно ложно (равно 0), вычисляется *выражение_1*, если при этом оказывается, что и значением *выражения_1* также является 0, вычисляется *выражение_2* и т. д. Если все выражения ложны (равны 0), то в компилируемый текст включается *текст_для_случая_else*. В противном случае, то есть при появлении хотя бы одного истинного выражения (в **#if** или в **#elif**), начинает обрабатываться текст, расположенный непосредственно за этой директивой, а все остальные директивы не рассматриваются.

Таким образом, при использовании директив условной компиляции препроцессор обрабатывает всегда только один из участков текста.

Операция defined. При условной обработке текста (при условной компиляции с использованием директив **#if**, **#elif**) для упрощения записи сложного условия выбора можно использовать унарную препроцессорную операцию

defined *операнд*

где *операнд* – либо идентификатор, либо заключенный в скобки идентификатор, либо обращение к макросу (см. §3.5). Если идентификатор операнда до этого определен с помощью команды **#define** как препроцессорный, то выражение **defined** *операнд* принимает значение 1L, то есть считается истинным. В противном случае его значение равно 0L.

Выражение

#if defined *операнд*

эквивалентно выражению

#ifdef *операнд*

Но в таком простом случае никакие достоинства операции **defined** не проявляются. Поясним с помощью примера другие полезные возможности операции **defined**. Предположим, что некоторый *важный_текст* должен быть передан компилятору только в том случае, если идентификатор Y определен как препроцессорный, а идентификатор N не определен. Директивы препроцессора могут быть записаны следующим образом:

#if defined Y && **!defined** N
важный_текст
#endif

Обработку препроцессор ведет следующим образом. Во-первых, определяется истинность выражений **defined** Y и **defined** N. Получаем два значения, каждое 0L или 1L. К результатам применяется операция && (конъюнкция), и при истинности ее результата *важный_текст* передается компилятору.

Не используя операцию **defined**, то же самое условие можно записать таким способом:

#ifdef Y
#ifndef N
важный_текст
#endif
#endif

Таким образом, из примера видно, что:

#if defined	эквивалентно	**#ifdef**
#if !defined	эквивалентно	**#ifndef**

Стандарт языка Си не включает **defined** в набор ключевых слов. В тексте программы его можно использовать в качестве идентификатора, свободно применяемого программистом для обозначения объектов. **defined** имеет специфическое значение только при формировании выражений-условий, проверяемых в директивах **#if** и **#elif**. Однако идентификатор **defined** запрещено использовать в директивах **#define** и **#undef**.

3.5. Макроподстановки средствами препроцессора

Макрос, по определению, есть средство замены одной последовательности символов на другую. Для выполнения замен должны быть заданы соответствующие макроопределения. Простейшее макроопределение мы уже ввели, рассматривая замены в тексте с помощью директивы

#define *идентификатор строка_замещения*

С помощью директивы **#define** программист может вводить собственные обозначения базовых или производных типов. Например, директива

```
#define REAL long double
```

вводит название (имя) REAL для типа **long double**. Далее в тексте программы можно определять конкретные объекты, используя REAL в качестве обозначения их типа (**long double**):

```
REAL x, array [6];
```

Идентификатор в команде **#define** может определять, как мы видели, имя константы, если *строка_замещения* задает значение этой константы. В более общем случае идентификатор служит обозначением некоторого выражения, например:

```
#define RANGE ((INT_MAX)-(INT_MIN)+1)
```

Идентификаторы, входящие в строку замещения, в свою очередь, могут быть определены как препроцессорные, и их значения будут подставлены вместо них (вместо **INT_MAX** и **INT_MIN** в нашем примере).

Допустимость выполнять с помощью **#define** «цепочки» подстановок расширяет возможности этой директивы, однако она имеет существенный недостаток – строка замещения фиксирована. Большие возможности предоставляет макроопределение с параметрами

#define *имя(список_параметров) строка_замещения*

Здесь *имя* – имя макроса (идентификатор), *список_параметров* – список разделенных запятыми идентификаторов. Между именем макроса и скобкой, открывающей список параметров, не должно быть пробелов.

Для обращения к макросу («для вызова макроса») используется конструкция («макровызов») вида:

имя_макроса (*список_аргументов*)

В списке аргументы разделены запятыми. Каждый аргумент – препроцессорная лексема.

Классический пример макроопределения

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

позволяет формировать в программе выражение, определяющее максимальное из двух значений аргументов. При таком определении вхождение в программу макровызова

```
max(X,Y)
```

заменяется выражением

```
(X<Y? Y:X)
```

а использование конструкции вида

```
max(Z,4)
```

приведет к формированию выражения

```
(Z<4? 4:Z)
```

В первом случае при истинном значении X < Y возвращается значение Y, иначе – значение X. Во втором примере значение переменной Z сравнивается с константой 4 и выбирается большее из значений.

Не менее часто используется определение

```
#define  ABS(X)  (X<0?  -(X):X)
```

С его помощью в программу можно вставлять выражение для определения абсолютных значений переменных. Конструкция

```
ABS(E-Z)
```

заменяется выражением

```
(E-Z<0?  -(E-Z):E-Z)
```

в результате вычисления которого определяется абсолютное значение выражения E–Z. Обратите внимание на скобки. Без них могут появиться ошибки в результатах.

Следует отметить, что последовательные препроцессорные подстановки выполняются в строке замещения, но не действуют на параметры макроса.

Моделирование многомерных массивов. В качестве еще одной области эффективного использования макросов укажем на проблему представления многомерных массивов в языке Си. Массивы мы будем подробно рассматривать в следующих главах, а пока остановимся только на вопросе применения макросредств для удобной адресации элементов матрицы. Напомним общие принципы представления массивов в языке Си.

1. Основным понятием в языке Си является одномерный массив, а возможности формирования многомерных массивов (особенно с переменными размерами) весьма ограничены.
2. Нумерация элементов массивов в языке Си начинается с нуля, то есть для обращения к начальному (первому) элементу массива требуется нулевое значение индекса.

При работе с матрицами обе указанные особенности массивов языка Си создают, по крайней мере, неудобства. Во-первых, при обращении к элементу матрицы нужно указывать два индекса – номер строки и номер столбца элемента матрицы. Во-вторых, нумерацию строк и столбцов матрицы принято начинать с 1.

Применение макросов для организации доступа к элементам массива позволяет программисту обойти оба указанных затруднения, правда, за счет нетрадиционных обозначений индексированных элементов. (Индексы в макросах, представляющих элементы массивов матриц, заключены в круглые, а не в квадратные скобки.) Рассмотрим следующую программу:

```
#define N   4   /* Число строк матрицы */
#define M   5   /* Число столбцов матрицы */
#define A(i,j)   x[M*(i-1) + (j-1)]
#include <stdio.h>
void main ( )
{
 /* Определение одномерного массива */
 double   x[N*M];
 int i,j, k;
 for (k=0; k < N*M; k++)
        x[k]=k;
 for (i=1; i<=N; i++) /* Перебор строк */
 {
   printf ("\n Строка %d:", i);
 /* Перебор элементов строки */
   for (j=1; j<=M; j++)
       printf(" %6.1f", A(i, j));
 }
}
```

Результат выполнения программы:

```
Строка 1:      0.0      1.0      2.0      3.0      4.0
Строка 2:      5.0      6.0      7.0      8.0      9.0
Строка 3:     10.0     11.0     12.0     13.0     14.0
Строка 4:     15.0     16.0     17.0     18.0     19.0
```

В программе определен одномерный массив x[ ], количество элементов в котором зависит от значений препроцессорных идентификаторов N и M.

Значения элементам массива x[ ] присваиваются в цикле с параметром k. Никаких новинок здесь нет. А вот далее для доступа к элементам того же массива x[ ] используются макровызовы вида A(i, j), причем i изменяется от 1 до N, а переменная j изменяется во внутреннем цикле от 1 до M. Переменная i соответствует номеру

строки матрицы, а переменная j играет роль второго индекса, то есть указывает номер столбца. При таком подходе программист оперирует с достаточно естественными обозначениями A(i, j) элементов матрицы, причем нумерация столбцов и строк начинается с 1, как и предполагается в матричном исчислении.

В тексте программы за счет макрорасширений в процессе препроцессорной обработки выполняются замены параметризованных обозначений A(i, j) на x[5*(i–1)+(j–1)], и далее действия выполняются над элементами одномерного массива x[ ]. Но этих преобразований программист не видит и может считать, что он работает с традиционными обозначениями матричных элементов. Использованный в программе оператор (вызов функции)

```
printf ("% 6.1f", A (i, j));
```

после макроподстановок будет иметь вид:

```
printf ("«% 6.1f", x[5*(i-1)+(j-1)]);
```

На рис. 3.1 приведена иллюстративная схема одномерного массива x[ ] и виртуальной (существующей только в воображении программиста, использующего макроопределения) матрицы для рассмотренной программы.

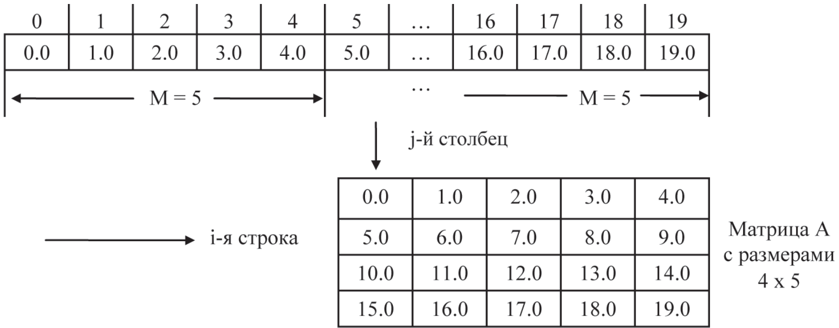


Рис. 3.1. *Имитация матрицы с помощью макроопределения и одномерного массива:*
A(1,1) соответствует x[5(1–1)+(1–1)] = = x[0];*
A(1,2) соответствует x[5(1–1)+(2–1)] = = x[1];*
A(2,1) соответствует x[5(2–1)+(1–1)] = = x[5];*
A(3,4) соответствует x[5(3–1)+(4–1)] = = x[13]*

Отличия макросов от функций. Сравнивая макросы с функциями, заметим, что, в отличие от функции, определение которой всегда присутствует в одном экземпляре, тексты, формируемые макросом, вставляются в программу столько раз, сколько раз используется макрос. Обратим внимание на еще одно отличие: функция определена для данных того типа, который указан в спецификации ее параметров и возвращает значение только одного конкретного типа. Макрос пригоден для обработки параметров любого типа, допустимых в выражениях, формируемых при обработке строки замещения. Тип получаемого значения зависит только от типов параметров и от самих выражений. Таким образом, макрос может заменять несколько функций. Например, приведенные макросы max( ) и ABS( ) верно работают для параметров любых целых и вещественных типов, а результат зависит только от типов параметров.

Отметим как рекомендацию, что для устранения неоднозначных или неверных использований макроподстановок параметры в строке замещения и ее саму полезно заключать в скобки.

Еще одно отличие: аргументы функций – это выражения, а аргументы вызова макроса – препроцессорные лексемы, разделенные запятыми. *Аргументы макрорасширениям не подвергаются.*

Препроцессорные операции в строке замещения. В последовательности лексем, образующей строку замещения, предусматривается использование двух операций – '**#**' и '**##**', первая из которых помещается перед параметром, а вторая – между любыми двумя лексемами. Операция '**#**' требует, чтобы текст, замещающий данный параметр в формируемой строке, заключался в двойные кавычки.

В качестве полезного примера с операцией '**#**' рассмотрим следующее макроопределение:

#define print (A) **printf** (#A"=%f", A)

К макросу print (A) можно обращаться, подставляя вместо параметра A произвольные выражения, формирующие результаты вещественного типа. Пример:

print (sin (a/2)); – обращение к макросу;
printf ("sin (a/2)""=%f", sin (a/2)); – макрорасширение.

Фрагмент программы:

```
double a=3.14159;
print (sin (a/2));
```

Результат выполнения (на экране дисплея):

```
sin (a/2)=1.0
```

Операция '**##**', допускаемая только между лексемами строки замещения, позволяет выполнять конкатенацию лексем, включаемых в строку замещения.

Чтобы пояснить роль и место операции '**##**', рассмотрим, как будут выполняться макроподстановки в следующих трех макроопределениях с одинаковым списком параметров и одинаковыми аргументами.

#define zero (a, b, c, d) a (bcd)
#define one (a, b, c, d) a (b c d)
#define two (a, b, c, d) a (b##c##d)

Макровызов	**Результат макроподстановки**
`zero(sin, x, +, y)`	`sin(bcd)`
`one(sin, x, +, y)`	`sin(x + y)`
`two(sin, x, +, y)`	`sin(x+y)`

В случае zero( ) последовательность «bcd» воспринимается как отдельный идентификатор. Замена параметров b, c, d не выполнена. В строке замещения макроса one( ) аргументы отделены пробелами, которые сохраняются в результате. В строке замещения для макроса two( ) использована операция '**##**', что позволило выполнить конкатенацию аргументов без пробелов между ними.

3.6. Вспомогательные директивы

В отличие от директив **#include**, **#define** и всего набора команд условной компиляции (**#if...**), рассматриваемые в данном параграфе директивы не так часто используются в практике программирования.

Препроцессорные обозначения строк. Для нумерации строк можно использовать директиву

#line *константа*

которая указывает компилятору, что следующая ниже строка текста имеет номер, определяемый целой десятичной константой. Директива может одновременно изменять не только номер строки, но и имя файла:

#line *константа «имя_файла»*

Как пишут в литературе по языку Си [5], директиву **#line** можно «встретить» сравнительно редко, за исключением случая, когда текст программы на языке Си генерирует какой-то другой препроцессор.

Смысл директивы **#line** становится очевидным, если рассмотреть текст, который препроцессор формирует и передает на компиляцию. После препроцессорной обработки каждая строка имеет следующий вид:

имя_файла номер_строки текст_на_языке_Си

Например, пусть препроцессор получает для обработки файл «www.c» с таким текстом:

```
#define N   3    /* Определение константы */
void main ( )
{
   #line  23 "file.c"
   double z[3*N];
}
```

После препроцессора в файле с именем «www.i» будет получен следующий набор строк:

```
www.c    1:
www.c    2:    void main( )
www.c    3:    {
www.c    4:
file.c  23:      double z[3*3]
file.c  24:    }
```

Обратите внимание на отсутствие в результирующем тексте препроцессорных директив и комментария. Соответствующие строки пусты, но включены в результирующий текст. Для них выделены порядковые номера (1 и 4). Следующая строка за директивой **#line** обозначена в соответствии со значением константы (23) и указанным именем файла «file.c».

Реакция на ошибки. Обработка директивы

#error *последовательность_лексем*

приводит к выдаче диагностического сообщения в виде, определенном последовательностью лексем. Естественно применение директивы **#error** совместно с условными препроцессорными командами. Например, определив некоторую препроцессорную переменную NAME

#define NAME 5

в дальнейшем можно проверить ее значение и выдать сообщение, если у NAME окажется другое значение:

```
#if (NAME != 5)
#error NAME должно быть равно 5 !
```

Сообщение будет выглядеть так:

```
Error <имя_файла> <номер_строки>:
Error directive: NAME должно быть равно 5 !
```

В случае выявления такой аварийной ситуации дальнейшая препроцессорная обработка исходного текста прекращается, и только та часть текста, которая предшествует условию **#if...**, попадает в выходной файл препроцессора.

Пустая директива. Существует директива, использование которой не вызывает никаких действий. Она имеет вид:

#

Прагмы. Директива

#pragma *последовательность_лексем*

определяет действия, зависящие от конкретной реализации компилятора. Например, в некоторые компиляторы входит вариант этой директивы для извещения компилятора о наличии в тексте программы команд на языке ассемблера.

Возможности команды **#pragma** могут быть весьма разнообразными и важными. Стандарта для них не существует. Если конкретный препроцессор встречает прагму, которая ему неизвестна, он ее просто игнорирует как пустую директиву. В некоторых реализациях включена прагма

#pragma pack(n)

где n может быть 1, 2 или 4.

Прагма «pack» позволяет влиять на упаковку смежных элементов в структурах и объединениях (см. главу 6).

Соглашение может быть таким:
pack(1) – выравнивание элементов по границам байтов;
pack(2) – выравнивание элементов по границам слов;
pack(4) – выравнивание элементов по границам двойных слов.

В некоторые компиляторы включены прагмы, позволяющие изменять способ передачи параметров функциям, порядок помещения параметров в стек и т. д.

3.7. Встроенные макроимена

Существуют встроенные (заранее определенные) макроимена, доступные препроцессору во время обработки. Они позволяют получить следующую информацию:

__LINE__ – десятичная константа – номер текущей обрабатываемой строки файла с программой на Си. Принято, что номер первой строки исходного файла равен 1;

__FILE__ – строка символов – имя компилируемого файла. Имя изменяется всякий раз, когда препроцессор встречает директиву **#include** с указанием имени другого файла. Когда включения файла по команде **#include** завершаются, восстанавливается предыдущее значение макроимени **__FILE__**;

__DATE__ – строка символов в формате «месяц число год», определяющая дату обработки исходного файла. Например, после препроцессорной обработки текста программы, выполненной 10 марта 2011 года, оператор

```
printf(__DATE__);
```

станет таким:

```
printf("Mar 10 2011");
```

__TIME__ – строка символов вида «часы:минуты:секунды», определяющая время начала обработки препроцессором исходного файла;

__STDC__ – константа, равная 1, если компилятор работает в соответствии с ANSI-стандартом. В противном случае значение макроимени **__STDC__** не определено. Стандарт языка Си предполагает, что наличие имени **__STDC__** определяется реализацией, так как макрос **__STDC__** относится к нововведениям стандарта.

В конкретных реализациях набор предопределенных имен зачастую шире.

Для получения более полных сведений о предопределенных препроцессорных именах следует обращаться к документации по конкретному компилятору.

Контрольные вопросы

1. Укажите формы препроцессорной директивы #**include**.
2. Объясните возможности использования директивы #**include** во включаемых текстах.
3. Для чего используется препроцессорная директива #**define**?
4. Объясните назначение директивы #**undef**.
5. Как препроцессорными средствами защитить текст от повторных включений?
6. С какого символа начинается препроцессорная директива?
7. Может ли препроцессор обрабатывать тексты, отличные от кода на языке Си?
8. Что является результатом работы препроцессора?
9. Как препроцессор обрабатывает строку, начинающуюся с символа # с последующим пробелом?
10. Объясните область действия директивы #**define**.
11. Объясните формат объявления препроцессорного идентификатора.
12. Можно ли препроцессорный идентификатор последовательно связывать с разными строками замещений?
13. Как с помощью директивы #**define** ввести константу?
14. Перечислите препроцессорные операции.
15. Как с помощью директив препроцессора разметить текст, чтобы он был «чувствителен» к количеству его включений в программу?
16. Как с помощью директивы #**define** можно вести макроопределение?
17. Где используется препроцессорная операция #?
18. Что нужно, чтобы включить в строку замещения макроса несколько операторов языка Си?
19. Как разместить строку замещения директивы #**define** в нескольких строках текста программы?
20. Перечислите различия и сходства макросов и функций.
21. Назовите препроцессорные директивы ветвлений обработки текста.
22. Что может служить операндом логического препроцессорного выражения?

23. Укажите назначения и возможности препроцессорных операций.
24. Чем отличаются препроцессорные макроопределения от макроподстановок?
25. Как организовать рекурсивную обработку текста с помощью средств препроцессора?
26. Как сделать идентификатор определенным для препроцессора?
27. Может ли содержать препроцессорные директивы текст, включаемый в программу директивой #**include**?
28. Для чего в строке замещения макроса может быть использована препроцессорная операция ##?
29. Почему имена параметров макроса в строке замещения рекомендуется заключать в круглые скобки?
30. Приведите примеры использования директивы #pragma.

Глава 4
УКАЗАТЕЛИ, МАССИВЫ, СТРОКИ

В предыдущих главах были введены все базовые (основные) типы языка Си. Для их определения и описания используются служебные слова: **char**, **short**, **int**, **long**, **signed**, **unsigned**, **float**, **double**, **enum**, **void**.

В языке Си, кроме базовых типов, разрешено вводить и использовать производные типы, каждый из которых получен на основе более простых типов. Стандарт языка определяет три способа получения производных типов:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

С массивами и функциями мы уже немного знакомы по материалам главы 2, а вот указатели требуют особого рассмотрения. В языке Си указатели введены как объекты, значениями которых служат адреса других объектов либо функций. Рассмотрим вначале указатели на объекты.

4.1. Указатели на объекты

Адреса и указатели. Начнем изучение указателей, обратившись к понятию переменной. Каждая переменная в программе – это объект, имеющий имя и значение. По имени можно обратиться к переменной и получить (а затем, например, напечатать) ее значение. В операторе присваивания выполняется обратное действие – имени переменной из левой части оператора присваивания ставится в соответствие значение выражения его правой части. С точки зрения машинной реализации, имя переменной соответствует адресу того участка памяти, который для нее выделен, а значение переменной – содержимому этого участка памяти. Соотношение между именем и адресом условно представлено на рис. 4.1.

Рис. 4.1. *Соотношение между именем и адресом*

На рис. 4.1 имя переменной явно не связано с адресом, однако, например, в операторе присваивания E=C+B; имя переменной E адресует некоторый участок памяти, а выражение C+B определяет значение, которое должно быть помещено в этот участок памяти. В операторе присваивания адрес переменной из левой части оператора обычно не интересует программиста и недоступен. Чтобы получить адрес в явном виде, в языке Си применяют унарную операцию **&**. Выражение &E позволяет получить адрес участка памяти, выделенного на машинном уровне для переменной E.

Операция **&** применима только к объектам, имеющим имя и размещенным в памяти. Ее нельзя применять к выражениям, константам-литералам, битовым полям структур (см. главу 6).

Рисунок 4.2, взятый с некоторыми изменениями из [6], хорошо иллюстрирует связь между именами, адресами и значениями переменных. На рис. 4.2 предполагается, что в программе использована, например, такая последовательность определений (с инициализацией):

```
char  ch='G';
int  date=1937;
float  summa=2.015E-6;
```

Примечание

В примере переменная ch занимает 1 байт, date – 2 байта и summa – 4 байта. В современных ПК переменная типа int может занимать 4 байта, а переменная типа float – 8 байтов.

В соответствии с приведенным рисунком переменные размещены в памяти, начиная с байта, имеющего шестнадцатеричный адрес

Машинный адрес:	1A2B	1A2C	1A2D	1A2E	1A2F	1A30	1A31	1A32
	байт	байт	байт	байт	байт	байт	байт	байт
Значение в памяти:	'G'	1937		$2.015*10^{-6}$				
Имя:	**ch**	**date**		**summa**				

Рис. 4.2. *Разные типы данных в памяти ЭВМ*

1A2B. При указанных выше размерах участков памяти в данном примере &ch = = 1A2B (адрес переменной ch); &date = = 1A2C; &summa = = 1A2E. Адреса имеют целочисленные беззнаковые значения, и их можно обрабатывать как целочисленные величины.

Имея возможность с помощью операции **&** определять адрес переменной или другого объекта программы, нужно уметь его сохранять, преобразовывать и передавать. Для этих целей в языке Си введены переменные типа «указатель», которые для краткости будем называть просто указателями, если это не приводит к неоднозначности или путанице. Указатель в языке Си можно определить как переменную, значением которой служит адрес объекта конкретного типа. Кроме того, значением указателя может быть заведомо не равное никакому адресу значение, принимаемое за нулевой адрес. Для его обозначения в ряде заголовочных файлов, например в файле **stdio.h**, определена специальная константа **NULL**.

Как и всякие переменные, указатели нужно определять и описывать, для чего используется, во-первых, разделитель '*'. В описании и определении переменных типа «указатель» необходимо сообщать, на объект какого типа ссылается описываемый указатель. Поэтому, кроме разделителя '*', в определения и описания указателей входят спецификации типов, задающие типы объектов, на которые ссылаются указатели. Примеры определения указателей:

```
char *z;
/* z - указатель на объект символьного типа */
int *k,*i;
/* k, i - указатели на объекты целого типа */
float *f;
/* f - указатель на объект вещественного типа */
```

После определения указателя к нему применима унарная операция '*', называемая операцией разыменования, или операцией обращения по адресу. Операндом операции разыменования всегда является указатель. Результат этой операции – тот объект, который адресует указатель-операнд. Таким образом, *z обозначает объект типа **char** (символьная переменная), на который указывает z; *k – объект типа **int** (целая переменная), на который указывает k, и т. д. Обозначения *z, *i, *f имеют права переменных соответствующих типов. Оператор *z=' '; засылает символ «пробел» в тот участок памяти, адрес которого определяет указатель z. Оператор *k=*i=0; заносит целые нулевые значения в те участки памяти, адреса которых заданы указателями k, i.

Обратите внимание на то, что указатель может ссылаться на объекты того типа, который присутствует в определении указателя. Исключением являются указатели, в определении которых использован тип **void** – отсутствие значения. Такие указатели могут ссылаться на объекты любого типа, однако к ним нельзя применять операцию разыменования, то есть операцию '*'.

Скрытую в операторе присваивания E=B+C; работу с адресом переменной левой части можно сделать явной, если заменить один оператор присваивания следующей последовательностью:

```
/* Определения переменных и указателя m: */
int E,C,B,*m;
/* Значению m присвоить адрес переменной E: */
m=&E;
/* Переслать значение выражения C+B в участок
   памяти с адресом, равным значению m: */
*m=B+C;
```

Данный пример не объясняет необходимости применения указателей, а только иллюстрирует их особенности. Возможности и преимущества указателей проявляются при работе с функциями, массивами, строками, структурами и т. д. Перед тем как привести более содержательные примеры использования указателей, остановимся подробнее на допустимых действиях с указателями.

Операции над указателями. В языке Си допустимы следующие (основные) операции над указателями: присваивание; получение значения того объекта, на который ссылается указатель (синонимы: косвенная адресация, разыменование, раскрытие ссылки); получение адреса самого указателя; унарные операции изменения значения

указателя; аддитивные операции и операции сравнений. Рассмотрим перечисленные операции подробнее.

Операция присваивания предполагает, что слева от знака операции присваивания помещено имя указателя, справа – указатель, уже имеющий значение, либо константа **NULL**, определяющая условное нулевое значение указателя, либо адрес любого объекта того же типа, что и указатель слева.

Если для имен действуют описания предыдущих примеров, то допустимы операторы:

```
i=&date;
k=i;
z=NULL;
```

Комментируя эти операторы, напомним, что выражение **имя_указателя* позволяет получить значение, находящееся по адресу, который определяет указатель. В предыдущих примерах было определено значение переменной date (1937), затем ее адрес присвоен указателю i и указателю k, поэтому значением *k является целое 1937. Обратите внимание, что имя переменной date и разыменования *i, *k указателей i, k обеспечивают в этом примере доступ к одному и тому же участку памяти, выделенному только для переменной date. Любая из операций *k=*выражение,* *i=*выражение,* date=*выражение* приведет к изменению содержимого одного и того же участка в памяти ЭВМ.

Иногда требуется присвоить указателю одного типа значение указателя (адрес объекта) другого типа. В этом случае используется «приведение типов», механизм которого понятен из следующего примера:

```
char *z;      /* z - указатель на символ */
int *k;       /* k - указатель на целое */
z=(char *)k; /* Преобразование указателей */
```

Подобно любым переменным, переменная типа указатель имеет имя, собственный адрес в памяти и значение. Значение можно использовать, например печатать или присваивать другому указателю, как это сделано в рассмотренных примерах. Адрес указателя может быть получен с помощью унарной операции &. Выражение *&имя_указателя* определяет, где в памяти размещен указатель. Содержимое этого участка памяти является значением указателя. Со-

отношение между именем, адресом и значением указателя иллюстрирует рис. 4.3.

Рис. 4.3. *Имя, адрес и значение указателя*

С помощью унарных операций '++' и '––' числовые (арифметические) значения переменных типа указатель меняются по-разному в зависимости от типа данных, с которыми связаны эти переменные. Если указатель связан с типом **char**, то при выполнении операций '++' и '––' его числовое значение изменяется на 1 (указатель z в рассмотренных примерах). Если указатель связан с типом **int** (указатели i, k), то операции ++i, i++, ––k, k–– изменяют числовые значения указателей на 2. Указатель, связанный с типом **float** или **long** унарными операциями '++', '––', изменяется на 4. Размеры участков памяти указаны в соответствии с примечанием на стр. 142. Таким образом, при изменении указателя на единицу указатель «переходит к началу» следующего (или предыдущего) поля той длины, которая определяется типом.

Аддитивные операции по-разному применимы к указателям, точнее имеются некоторые ограничения при их использовании. Две переменные типа указатель нельзя суммировать, однако к указателю можно прибавить целую величину. При этом вычисляемое значение зависит не только от значения прибавляемой целой величины, но и от типа объекта, с которым связан указатель. Например, если указатель, как в примере, относится к целочисленному объекту типа **int**, то прибавление к нему единицы увеличивает реальное значение на **2**, то есть выполняется «переход» к адресу следующего участка.

В отличие от операции сложения, операция вычитания применима не только к указателю и целой величине, но и к двум указателям на объекты одного типа. С ее помощью можно находить разность (со знаком) двух указателей (одного типа) и тем самым определять «расстояние» между размещением в памяти двух объектов. При

этом «расстояние» вычисляется в единицах, кратных «длине» отдельного элемента данных того типа, к которому отнесен указатель.

Например, после выполнения операторов

```
int x[5], *i, *k, j;
i=&x[0];  k=&x[4];   j=k-i;
```

j принимает значение 4, а не 8, как можно было бы предположить, исходя из того, что каждый элемент массива x[ ] занимает два байта.

В данном примере разность указателей присвоена переменной типа **int**. Однако тип разности указателей определяется по-разному в зависимости от особенностей компилятора и аппаратной платформы. Чтобы сделать язык Си независимым от реализаций, в заголовочном файле **stddef.h** определено имя (название) **ptrdiff_t**, с помощью которого обозначается тип разности указателей в конкретной реализации.

В следующей программе используется рассмотренная возможность однозначного задания типа разности указателей. Программа будет корректно выполняться со всеми компиляторами, соответствующими стандартам языка Си.

```
#include <stdio.h>
#include <stddef.h>
void main()
{
   int x[5];
   int *i,*k;
   ptrdiff_t j;
   i=&x[0];
   k=&x[4];
   j=k-i;
   printf("\nj=%d",(int)j);
}
```

Результат будет таким:

```
j=4
```

Арифметические операции и указатели. Унарные адресные операции '&' и '*' имеют более высокий приоритет, чем арифметические операции. Рассмотрим следующий пример, иллюстрирующий это правило:

```
float a=4.0, *u, z;
u=&z;
*u=5;
a=a + *u + 1;
/* a равно 10, u - не изменилось, z равно 5  */
```

При использовании адресной операции '*' в арифметических выражениях следует остерегаться случайного сочетания знаков операций деления '/' и разыменования '*', так как комбинацию '/*' компилятор воспринимает как начало комментария. Например, выражение

```
a/*u
```

следует заменить таким:

```
a/(*u)
```

Унарные операции '*****' и '**++**' или '**--**' имеют одинаковый приоритет и при размещении рядом выполняются ***справа налево***.

Добавление целочисленного значения n к указателю, адресующему некоторый элемент массива, приводит к тому, что указатель получает значение адреса того элемента, который отстоит от текущего на n позиций (элементов). Если длина элемента массива равна d байтов, то численное значение указателя изменяется на (d*n). Рассмотрим следующий фрагмент программы, иллюстрирующий перечисленные правила:

```
int x[4]={ 0, 2, 4, 6 }, *i, y;
i=&x[0];  /* i равно адресу элемента x[0]*/
y=*i;     /* y равно 0;  i равно &x[0]  */
y=*i++;   /* y равно 0;  i равно &x[1]  */
y=++*i;   /* y равно 3;  i равно &x[1]  */
y=*++i;   /* y равно 4;  i равно &x[2]  */
y=(*i)++; /* y равно 4;  i равно &x[2]  */
y=++(*i); /* y равно 6;  i равно &x[2]  */
```

Указатели и отношения. К указателям применяются операции сравнения '**>**', '**>=**', '**!=**', '**==**', '**<=**', '**<**'. Таким образом, указатели можно использовать в отношениях. Но сравнивать указатели допустимо только с другими указателями того же типа или с константой **NULL**, обозначающей значение условного нулевого адреса.

Приведем пример, в котором используются операции над указателями и выводятся (печатаются) получаемые значения. Обратите внимание, что для вывода значений указателей (адресов) в форматной строке функции **printf( )** используется спецификация преобразования **%p**.

```
#include <stdio.h>
float x[ ] = { 10.0, 20.0, 30.0, 40.0, 50.0 };
void main( )
{
 float *u1, *u2;
 int i;
 printf("\n Адреса указателей: &u1=%p &u2=%p", &u1, &u2 );
 printf("\n Адреса элементов массива: \n");
 for(i=0; i<5; i++)
      {
       if (i==3) printf("\n");
       printf(" &x[%d] = %p", i, &x[i]);
      }
 printf("\n Значения элементов массива: \n");
 for(i=0; i<5; i++)
      {
       if (i==3) printf("\n");
       printf(" x[%d] = %5.1f ", i, x[i]);
      }
 for(u1=&x[0], u2=&x[4]; u2>=&x[0]; u1++, u2--)
      {
       printf("\n u1=%p *u1=%5.1f  u2=%p *u2=%5.1f",u1,*u1,u2,*u2);
       printf("\n u2-u1=%d", u2-u1);
      }
}
```

При печати значений разностей указателей и адресов в функции **printf( )** использована спецификация преобразования **%d** – вывод знакового десятичного целого.

Возможный результат выполнения программы (конкретные значения адресов могут быть другими):

```
Адреса указателей: &u1=FFF4  &u2=FFF2
Адреса элементов массива:
&x[0]=00A8    &x[1]=00AC    &x[2]=00B0
&x[3]=00B4    &x[4]=00B8
Значения элементов массива:
```

```
x[0]=10.0      x[1]=20.0      x[2]=30.0
x[3]=40.0      x[4]=50.0
u1=00A8       *u1=10.0      u2=00B8       *u2=50.0
u2-u1=4
u1=00AC       *u1=20.0      u2=00B4       *u2=40.0
u2-u1=2
u1=00B0       *u1=30.0      u2=00B0       *u2=30.0
u2-u1=0
u1=00B4       *u1=40.0      u2=00AC       *u2=20.0
u2-u1=-2
u1=00B8       *u1=50.0      u2=00A8       *u2=10.0
u2-u1=-4
```

На рис. 4.4 приводится схема размещения в памяти массива **float** x[5] и указателей до начала выполнения цикла изменения указателей.

	Основная память	Адреса
x[0]	10.0	00A8=&x[0]
x[1]	20.0	00AC=&x[1]
x[2]	30.0	00B0=&x[2]
x[3]	40.0	00B4=&x[3]
x[4]	50.0	00B8=&x[4]
	. . .	
U2	00B8	FFF2=&U2
U1	00A8	FFF4=&U1

Рис. 4.4. *Схема размещения в памяти массива и указателей*

4.2. Указатели и массивы

Указатели и доступ к элементам массивов. По определению, указатель – это либо объект со значением «адрес объекта» или «адрес функции», либо выражение, позволяющее получить адрес объекта или функции. Рассмотрим фрагмент:

```
int x,y;
int *p =&x;
p=&y;
```

Здесь p – указатель-объект, а &x, &y – указатели-выражения, то есть адреса-константы. Мы уже знаем, что p – переменная того же типа, что и значения &x, &y. Различие между адресом (то есть указателем-выражением) и указателем-объектом заключается в возможности изменять значения указателей-объектов. Именно поэтому указатели-выражения называют указателями-константами или адресами, а для указателя-объекта используют название *указатель-переменная* или просто *указатель*.

В соответствии с синтаксисом языка Си имя массива без индексов является указателем-константой, то есть адресом его первого элемента (с нулевым индексом). Это нужно учитывать и помнить при работе с массивами и указателями.

Рассмотрим задачу «инвертирования» массива символов и различные способы ее решения с применением указателей (заметим, что задача может быть легко решена и без указателей – с использованием индексации). Предположим, что длина массива типа **char** равна 80.

Первое решение задачи инвертирования массива:

```
char z[80],s;
char *d,*h;
/* d и h - указатели на символьные объекты */
for (d=z, h=&z[79]; d<h; d++,h--)
   {
      s=*d;
      *d=*h;
      *h=s;
   }
```

В заголовке цикла указателю d присваивается адрес первого (с нулевым индексом) элемента массива z. Здесь можно было бы применить и другую операцию, а именно d=&z[0]. Указатель h получает значение адреса последнего элемента массива z. Далее работа с указателями в заголовке ведется как с обычными целочисленными переменными. Цикл выполняется до тех пор, пока d<h. После каждой итерации значение d увеличивается, значение h уменьшается на 1. При первой итерации в теле цикла выполняется обмен значе-

ний z[0] и z[79], так как d – адрес z[0], h – адрес z[79]. При второй итерации значением d является адрес z[1], для h – адрес z[78] и т. д.

Второе решение задачи инвертирования массива:

```
char  z[80] ,s,  *d,  *h;
for  (d=z,  h=&z[79];  d<h;)
   {
    s=*d;   *d++=*h;   *h-- = s;
   }
```

Приращение указателя d и уменьшение указателя h перенесены в тело цикла. Напоминаем, что в выражениях *d++ и *h–– операции увеличения (постфиксный инкремент) и уменьшения (постфиксный декремент) на 1 имеют тот же приоритет, что и унарная адресная операция '*'. Поэтому изменяются на 1 не значения элементов массива, на которые указывают d и h, а сами указатели. Последовательность действий такая: по значению указателя d (или h) обеспечивается доступ к элементу массива; в этот элемент заносится значение из правой части оператора присваивания; затем увеличивается (уменьшается) на 1 значение указателя d (или h).

Третье решение задачи инвертирования массива (используется цикл с предусловием):

```
char  z[80],  s,  *d,  *h;
d=z;
h=&z[79];
while  (d  <  h)
   {
     s=*d;    *d++ = *h;    *h-- = s;
   }
```

Четвертое решение задачи инвертирования массива (имитация индексированных переменных указателями со смещениями):

```
char  z[80],s;
int  i;
for  (i=0;  i<40;  i++)
   {
    s  =  *(z+i);
    *(z+i)  =  *(z+(79-i));
    *(z+(79-i))  =  s;
   }
```

Последний пример демонстрирует возможность использования вместо индексированного элемента z[i] выражения *(z+i). В языке Си, как мы упоминали, имя массива без индексов есть адрес его первого элемента (с нулевым значением индекса). Прибавив к имени массива целую величину, получаем адрес соответствующего элемента, таким образом, &z[i] и z+i – это две формы определения адреса одного и того же элемента массива, отстоящего на i позиций от его начала.

Итак, в соответствии с синтаксисом языка операция индексирования E1[E2] определена таким образом, что она эквивалентна *(E1+E2), где E1 – имя массива, E2 – целое. Для многомерного массива правила остаются теми же. Таким образом, E[n][m][k] эквивалентно *(E[n][m]+k) и, далее, *(*(*(E+n)+m)+k).

Отметим, что имя массива не является переменной типа *указатель*, а есть константа – адрес начала массива. Таким образом, к имени массива не применимы операции '++' (увеличения), '––' (уменьшения), имени массива нельзя присвоить значение, то есть имя массива не может использоваться в левой части оператора присваивания.

В рассмотренных примерах указатели относились к символьным переменным, и поэтому их приращения были единичными. Однако это обеспечивалось только особенностями представления в памяти символов – каждый символ занимает в памяти один байт, и поэтому адреса смежных элементов символьного массива отличаются на 1. В случае массивов с другими элементами (другого типа) единичному изменению указателя, как уже отмечалось, соответствует большее изменение адреса.

В следующей программе продемонстрируем еще раз особенности изменения указателей при переходе от элемента к элементу в массивах с элементами разных типов:

```
#include <stdio.h>
/* Изменение значений указателей на элементы массива */
void main( )
{
 char z[5]; int m[5];   float a[5];
 char *uz;   int *um;     float *ua;
 printf("\n Адреса элементов символьного "
         "массива:\n");
 for (uz=z; uz <= &z[4]; uz++)
      printf(" %10p",uz);
 printf("\n Адреса элементов целочисленного "
```

```
        "массива:\n");
 for (um=m; um <= &m[4]; um++)
      printf ("%10p",um);
 printf("\n Адреса элементов вещественного "
        "массива:\n");
 for (ua=a; ua <= &a[4]; ua++)
      printf(" %10p",ua);
}
```

Результат выполнения программы:

```
Адреса элементов символьного массива :
FFF0    FFF1    FFF2    FFF3    FFF4
Адреса элементов целочисленного массива:
FFF6    FFF8    FFFA    FFEC    FFFE
Адреса элементов вещественного массива:
FFD0    FFD4    FFD8    FFDC    FFE0
```

Как подтверждает рассмотренный пример, изменение указателя на 1 приводит к разным результатам в зависимости от типа объекта, с которым связан указатель. Значение указателя изменяется на длину участка памяти, выделенного для элемента, связанного с указателем. Символы занимают в памяти по одному байту, поэтому значение указателя uz изменяется на 1 при переходе к соседнему элементу символьного массива. Для целочисленного массива переход к соседнему элементу изменяет указатель um на 2. Для массива вещественных элементов с плавающей точкой переход к соседнему элементу изменяет указатель ua на 4.

Массивы динамической памяти. В соответствии со стандартом языка массив представляет собой совокупность элементов, каждый из которых имеет одни и те же атрибуты (характеристики). Все элементы размещаются в смежных участках памяти подряд, начиная с адреса, соответствующего началу массива, то есть значению & *имя_массива* [0].

При традиционном определении массива:

тип имя_массива [*количество_элементов*];

имя_массива становится указателем на область памяти, выделяемой для размещения элементов массива. *Количество_элементов* в соответствии с синтаксисом языка должно быть константным выражением. *Тип* явно определяет размеры памяти, выделяемой для каждого элемента массива.

Таким образом, общее количество элементов массива и размеры памяти, выделяемой для него, полностью и однозначно заданы определением. Это не всегда удобно. Иногда нужно, чтобы память для массива выделялась в таких размерах, какие нужны для решения конкретной задачи, причем потребности в памяти заранее неизвестны и не могут быть фиксированы.

Формирование массивов с переменными размерами можно организовать с помощью указателей и средств для динамического выделения памяти. Начнем рассмотрение указанных средств с библиотечных функций, описанных в заголовочных файлах **alloc.h** и **stdlib.h** стандартной библиотеки (файл **alloc.h** не является стандартным). В табл. 4.1 приведены сведения об этих библиотечных функциях. Функции **malloc( ), calloc( )** и **realloc( )** динамически выделяют память в соответствии со значениями параметров и возвращают адрес начала выделенного участка памяти. Для универсальности тип возвращаемого значения каждой из этих функций есть **void***. Этот указатель (указатель такого типа) можно преобразовать к указателю любого типа с помощью операции явного приведения типа (*тип* *).

Таблица 4.1. Функции для выделения и освобождения памяти

Функция	Прототип и краткое описание
malloc	**void** * **malloc** (**unsigned** s); Возвращает указатель на начало области (блока) динамической памяти длиной в s байт. При неудачном завершении возвращает значение **NULL**
calloc	**void** * **calloc** (**unsigned** n, **unsigned** m); Возвращает указатель на начало области (блока) обнуленной динамической памяти, выделенной для размещения n элементов по m байт каждый. При неудачном завершении возвращает значение **NULL**
realloc	**void** * **realloc** (**void** * bl, **unsigned** ns); Изменяет размер блока ранее выделенной динамической памяти до размера ns байт. bl – адрес начала изменяемого блока. Если bl равен **NULL** (память не выделялась), то функция выполняется как **malloc**
free	**void** * **free** (**void** * bl); Освобождает ранее выделенный участок (блок) динамической памяти, адрес первого байта которого равен значению bl

Функция **free**( ) решает обратную задачу – освобождает память, выделенную перед этим с помощью одной из трех функций: **calloc**( ), **malloc**( ) или **realloc**( ). Сведения об этом участке памяти

передаются в функцию **free( )** с помощью указателя – параметра типа **void** *. Преобразование указателя любого типа к типу **void *** выполняется автоматически, поэтому вместо параметра **void ***bl можно подставить в качестве аргумента указатель любого типа без операции явного приведения типов.

Следующая программа иллюстрирует на несложной задаче особенности применения функций выделения **(malloc)** и освобождения **(free)** динамической памяти. Решается следующая задача: ввести и напечатать в обратном порядке набор вещественных чисел, количество которых заранее не фиксировано, а вводится до начала ввода самих числовых значений. Текст программы может быть таким:

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{
 /* Указатель для выделяемого блока памяти */
 float *t;
 int i,n;
 printf("\nn="); /* n - число элементов */
 scanf("%d",&n);
 t=(float *)malloc(n*sizeof(float));
 for(i=0; i<n; i++)  /* Цикл ввода чисел */
    { printf("x[%d]=",i);
      scanf("%f",&t[i]);
    }
 /* Цикл печати результатов */
 for(i=n-1; i>=0; i--)
    {
      if (i%2==0) printf("\n");
      printf("\tx[%d]=%f",i,t[i]);
    }
 free (t); /* Освобождение памяти */
}
```

Результат выполнения программы (компилятор gcc из операционной системы BC++3.1 и компилятор из операционной системы UNIX FreeBSD):

```
n=4      <Enter>
x[0]=10  <Enter>
x[1]=20  <Enter>
```

```
x[2]=30   <Enter>
x[3]=40   <Enter>

                    x[3]=40.000000        x[2]=30.000000
                    x[1]=20.000000        x[0]=10.000000
```

В программе **int** n – количество вводимых чисел типа **float**, t – указатель на начало области, выделяемой для размещения n вводимых чисел. Указатель t принимает значение адреса области, выделяемой для n значений типа **float**. Обратите внимание на приведение к типу **(float*)** значения, возвращаемого функцией **malloc( )**. Доступ к участкам выделенной области памяти выполняется с помощью операции индексирования: t[i] и t[i–1]. Остальное очевидно из текста программы. Оператор **free**(t); содержит вызов функции, освобождающей выделенную ранее динамическую память, адресованную указателем t.

Массивы в каноническом смысле, вводимые с помощью определения массива, отличаются от массивов динамической памяти. Наиболее существенным отличием является отсутствие «имени собственного» у динамического массива. Чтобы продемонстрировать важность этого различия, рассмотрим стандартную процедуру определения количества элементов канонического массива с помощью операции **sizeof**.

Напомним, что операция **sizeof** имеет две формы вызова:

sizeof (*тип*)
sizeof *выражение*

В качестве типа могут использоваться типы любых объектов (нельзя использовать тип функции), за исключением типа **void**. В результате выполнения операции **sizeof** вычисляется размер в байтах ее операнда. Если операнд есть выражение, то вычисляется его значение, для этого значения определяется тип, для которого и вычисляется длина в байтах. Например, в компиляторе gcc из ОС FreeBSD:

sizeof (long) обычно равно 4;
sizeof (long double) обычно равно 10.

Чтобы в программе вычислить количество элементов канонически определенного конкретного массива, удобно использовать операцию **sizeof**, применяя ее дважды – к имени массива и к его нулевому элементу. Если операндом для **sizeof** служит имя массива (без

индексов), то возвращается значение размера памяти, отведенной для массива в целом. Таким образом, значением выражения

sizeof (*имя_массива*) /**sizeof** (*имя_массива*[0])

будет количество элементов в массиве.

В следующем фрагменте «перебираются» все элементы массива x:

```
float x[8];
for (i=0; sizeof(x)/sizeof(x[0])>i; i++)
....x[i]...
```

Если применить операцию **sizeof** к тому указателю, который получил значение адреса начала памяти, выделенной для динамического массива с помощью одной из функций табл. 4.1, то будет вычислен размер не динамического участка памяти, а только размер самого указателя. Для иллюстрации этого положения рассмотрим фрагмент программы:

```
double *pointer;
pointer=(double*) malloc(100);
... sizeof(pointer)
```

Независимо от значения аргумента функции **malloc( )** значение выражения **sizeof**(pointer) всегда будет одинаковым – равным величине участка памяти для переменной (для указателя) pointer.

Массивы указателей и моделирование многомерных массивов. Хотя синтаксисом языка Си массив определяется как одномерная совокупность его элементов, но каждый элемент может быть, в свою очередь, массивом. Именно так конструируются многомерные массивы. Язык Си не накладывает явных ограничений на размерность массива, однако каждая реализация обычно вводит такое ограничение. Максимальное значение размерности массивов определяет константа (препроцессорная), определенная в заголовочном файле стандартной библиотеки.

В главе 3 матрицы имитировались с применением одномерных массивов и макроопределений, обеспечивающих доступ к элементам с помощью двух индексов.

В главе 2 рассматривались двумерные массивы фиксированных размеров и приводились примеры их использования для представления матриц. Однако матрицы (двумерные массивы) имели фиксированные размеры. Очевидного способа прямого определения мно-

гомерных массивов с несколькими переменными размерами в языке Си не существует. Решение, как и для случая одномерных массивов динамической памяти, находится в области системных средств (библиотечных функций) для динамического управления памятью. Но прежде чем рассматривать их возможности для многомерных массивов, необходимо ввести массивы указателей.

Массив указателей фиксированных размеров вводится одним из следующих определений:

тип * *имя_массива* [*размер*];
тип * *имя_массива* [ ]=*инициализатор;*
тип * *имя_массива* [*размер*]=*инициализатор;*

где *тип* может быть как одним из базовых типов, так и производным типом; *имя_массива* – свободно выбираемый идентификатор; *размер* – константное выражение, вычисляемое в процессе трансляции; *инициализатор* – список в фигурных скобках значений типа *тип**.

Примеры:

```
int data[6];   /* обычный массив      */
int *pd[6];     /* массив указателей */
int *pi[ ]={ &data[0], &data[4], &data[2] };
```

Здесь каждый элемент массивов pd и pi является указателем на объекты типа **int**. Значением каждого элемента pd[j] и pi[k] может быть адрес объекта типа **int**. Все 6 элементов массива pd указателей типа **int *** не инициализированы. В массиве pi три элемента, которые инициализированы адресами конкретных элементов массива data.

Интересные возможности массива указателей появляются в тех случаях, когда его элементы адресуют либо элементы другого массива, либо указывают на начало одномерных массивов соответствующего типа. В таких случаях очень красиво решаются задачи сортировки сложных объектов с разными размерами. Например, можно ввести массив указателей, адресовать с помощью его элементов строки матрицы и затем решать задачи упорядочения строк матрицы, не переставляя строки двумерного массива, представляющего матрицу. Если с двумерным массивом связать несколько одномерных массивов указателей, то можно упорядочить строки матрицы одновременно по разным правилам, «добираясь» до строк с помощью разных массивов указателей.

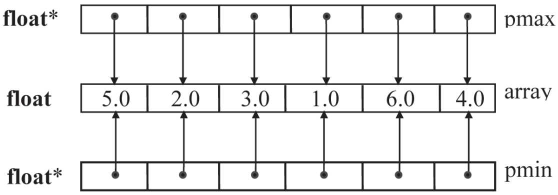
В качестве простого примера рассмотрим программу одновременного упорядочения и по возрастанию, и по убыванию одномерного массива без перестановки его элементов.

```
#include  <stdio.h>
#define B 6
void main( )
{
   float array[ ]={5.0, 2.0, 3.0, 1.0, 6.0, 4.0};
   float * pmin[B];
   float * pmax[B];
   float * e;
   int i,j;
   for (i=0; i<B; i++)
       pmin[i]=pmax[i]=&array[i];
   for (i=0; i<B-1; i++)
   for (j=i+1; j<B; j++)
      {
         if (*pmin[i]<*pmin[j])
            {
               e=pmin[i];
               pmin[i]=pmin[j];
               pmin[j]=e;
            }
         if (*pmax[i]>*pmax[j])
            {
               e=pmax[i];
               pmax[i]=pmax[j];
               pmax[j]=e;
            }
      }
   printf("\n По убыванию: \n");
   for (i=0; i<B; i++)
         printf("\t%5.3f",*pmin[i]);
   printf("\n По возрастанию: \n");
   for (i=0; i<B; i++)
         printf("\t%5.3f",*pmax[i]);
}
```

Результаты выполнения программы:

```
По убыванию:
    6.000     5.000     4.000     3.000     2.000     1.000
По возрастанию:
    1.000     2.000     3.000     4.000     5.000     6.000
```

В программе с помощью перестановки значений элементов массива указателей pmin[6] определяется последовательность «просмотра» элементов массива array[6] в порядке убывания их значений. Массив указателей pmax[6] решает такую же задачу, но для «просмотра» массива array[6] в порядке возрастания их значений. Рисунок 4.5 иллюстрирует исходную и результирующую адресации элементов массива array[6] элементами массивов указателей. Для «обмена» значений элементов массивов pmax[6] и pmin[6] введен вспомогательный указатель **float** * e.



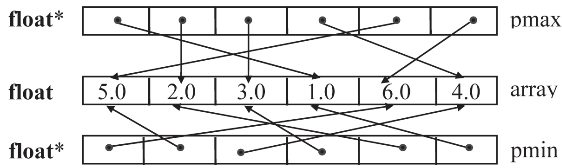


Рис. 4.5. *Массивы указателей в задаче упорядочения:*
а – массивы до упорядочения; б – массивы после упорядочения

«Матрица» со строками разной длины. Массив в языке Си должен иметь элементы одного типа и, естественно, одного размера. В ряде случаев возникает необходимость обрабатывать, подобно элементам массива, объекты разного размера. Например, попытка оформить в виде двумерного массива строки чисел с разным количеством числовых значений при обычном подходе потребует определить двумерный массив с максимально допустимыми размерами. Наличие более коротких строк не может уменьшить общих размеров массива – излишние требования к памяти налицо. Использование массивов указателей и средств динамического выделения памяти позволяет обойти указанные затруднения и более рационально распределить память. Для иллюстрации этих возможностей рассмотрим следующую задачу, очень схожую с задачей, рассмотренной ранее.

Необходимо ввести и распечатать в обратном порядке набор строк числовых значений. Количество строк в наборе вводится в начале работы программы, а длина каждой строки (то есть количество чисел – элементов в ней) вводится перед каждой последовательностью числовых значений элементов строки.

Программа для решения задачи может быть такой:

```
#include <stdio.h>
#include <alloc.h>
void main( )
{
   double ** line;
            /* line - указатель для блока памяти,
               выделяемого для массива указателей */
   int i,j,n;
   double dd;
   int * m; /* Указатель для блока памяти */
   printf("\n\nВведите количество строк n=");
   scanf("%d",&n);
   line=(double**)calloc(n,sizeof(double*));
   m=(int *)malloc(sizeof(int)*n);

/* Цикл по количеству строк*/
   for(i=0;i<n;i++)
   {
    printf("Введите длину строки m[%d]=",i);
    scanf("%d",&m[i]);
    line[i]=(double*)calloc(m[i],sizeof(double));
    for(j=0;j<m[i];j++)
    {
       printf("line[%d][%d]=",i,j);
       scanf("%le",&dd);
       line[i][j]=dd;
    }
   }
   printf("\nРезультаты обработки:");
  for(i=n-1;i>=0;i--)
  {
   printf("\nСтрока %d, элементов %d:\n",i,m[i]);
   for(j=0;j<m[i];j++)
      printf("\t%f", line[i][j]);
   free(line[i]); /* Освободить память строки */
  }
  /* Освободить память от массива указателей: */
```

```
 free(line);
 /* Освободить память от массива int: */
 free(m);
}
```

Результаты выполнения программы (компилятор gcc операционной системы UNIX FreeBSD):

```
Введите количество строк n=5    <Enter>
Введите длину строки m[0]=4     <Enter>

line[0][0]=0.0
line[0][1]=0.1
line[0][2]=0.2
line[0][3]=0.3

Введите длину строки m [1]=3   <Enter>

line[1][0]=1.0
line[1][1]=1.1
line[1][2]=1.2
Введите длину строки m [2]=1   <Enter>

line[2][0]=2.0

Введите длину строки m [3]=4   <Enter>

line[3][0]=3.0
line[3][1]=3.1
line[3][2]=3.2
line[3][3]=3.3

Введите длину строки m [4]=2   <Enter>

line [4][0]=4.0
line [4][1]=4.1

Результат обработки:

Строка 4, элементов 2:
4.000000      4.100000
Строка 3, элементов 4:
3.000000      3.100000      3.200000      3.300000
Строка 2, элементов 1:
2.000000
Строка 1, элементов 3:
1.000000      1.100000      1.200000
Строка 0, элементов 4:
0.000000      0.100000      0.200000      0.300000
```

Приведенным результатам выполнения программы соответствует рис. 4.6, где условно изображены все массивы, динамически формируемые в процессе выполнения программы.

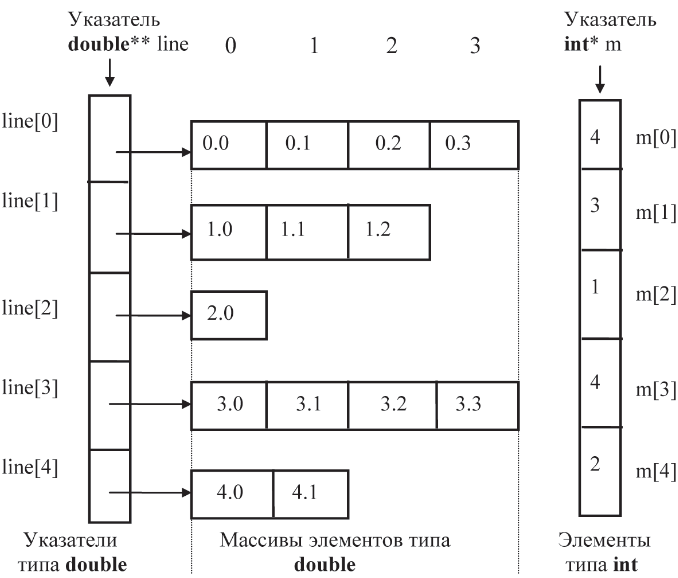


Рис. 4.6. *Схема «матрицы» со строками разной длины для случая: число строк n = 5, количество элементов в строках: 4, 3, 1, 4, 2*

В программе использованы:

- line – указатель на массив указателей типа **double** *, каждый из которых, то есть line[i], адресует динамически выделяемый участок памяти длиной в m[i] элементов типа **double**;
- m – указатель на массив целых **(int)**, значение каждого элемента равно длине массива, на который указывает line[i].

Для иллюстрации разных функций выделения памяти в программе использованы **calloc( )** и **malloc( )**. Различие между ними заключается в количестве и смысле параметров (см. табл. 4.1), а также в том, что функция **calloc( )** обнуляет содержимое выделенного блока памяти.

При выходе из программы все блоки динамически выделяемой памяти рекомендуется явным образом освобождать. Для этих целей используется несколько вызовов функции **free( )**. В цикле по i после печати очередной строки функция **free** (line[i]) освобождает участок

памяти, адресуемой указателем line[i]. После окончания цикла освобождаются блоки, выделенные для массива указателей **free**(line) и массива целых **free**(m).

4.3. Символьная информация и строки

Для представления текстовой информации в языке Си используются символы (константы), символьные переменные и строки (строковые константы), для которых в языке Си не введено отдельного типа, в отличие от некоторых других языков программирования.

Символьные константы были рассмотрены в главе 1. Для символьных данных введен базовый тип **char**. Описание символьных переменных имеет вид:

char *список_имен_переменных;*

Например:

```
char a,z;
```

Ввод-вывод символьных данных. Для ввода и вывода символьных значений в форматных строках библиотечных функций **printf( )** и **scanf( )** используется спецификация преобразования %c. Некоторые особенности работы с символьными данными уже рассматривались выше. Смотрите, например, разные способы «инвертирования» массива символов в §4.2. Продолжая эту тему, рассмотрим следующую задачу.

Ввести предложение, слова в котором разделены пробелами и в конце которого стоит точка. Удалить повторяющиеся пробелы между отдельными словами (оставить по одному пробелу), вывести отредактированное предложение на экран.

Текст программы:

```
/* Удаление повторяющихся пробелов */
#include <stdio.h>
void main( )
{
   char z, s;   /* z - текущий вводимый символ */
   printf("\n Напишите предложение с точкой "
          "в конце:\n");
   for (z=s=' '; z!='.'; s=z)
   { /* s - предыдущий символ */
      scanf("%c",&z);
```

```
    if (z==' ' && s==' ') continue;
    printf("%c",z);
  }  /* Конец цикла обработки */
}    /* Конец программы */
```

В программе две символьные переменные: z – для чтения очередного символа и s – для хранения предыдущего. В заголовке цикла переменные z и s получают значения «пробел». Очередной символ вводится как значение переменной z, и пара z, s анализируется. Если хотя бы один из символов значений пары отличен от пробела, то значение z печатается. В заголовке цикла z сравнивается с символом «точка» и при несовпадении запоминается как значение s. Далее цикл повторяется до появления на входе точки, причем появление двух пробелов (z и s) приводит к пропуску оператора печати.

Пример работы программы:

```
Напишите предложение с точкой в конце:
YYYYY    yyyyy   hhhhh   ttttt. <Enter>
YYYYY yyyyy hhhhh ttttt.
```

Помимо **scanf( )** и **printf( )**, для ввода и вывода символов в библиотеке предусмотрены специальные функции обмена:

- **getchar( )** – функция без параметров. Позволяет читать из входного потока (обычно с клавиатуры) по одному символу за обращение. Как и при использовании функции **scanf( )**, чтение вводимых данных начинается после нажатия клавиши <Enter>. Это дает возможность исправлять вводимую информацию (стирая последние введенные символы с помощью клавиши <Backspace>) до начала ее чтения программой;
- **putchar(**X**)** – выводит символьное значение X в стандартный выходной поток (обычно на экран дисплея).

Проиллюстрируем применение этих функций на следующей задаче: «Ввести предложение, в конце которого стоит точка, и подсчитать общее количество символов, отличных от пробела (не считая точки)».

```
/* Подсчет числа отличных от пробелов символов*/
#include <stdio.h>
void main( )
{
  char z; /* z - вводимый символ  */
  int k;  /* k - количество значащих символов */
```

```
   printf("Напишите предложение с точкой в "
          "конце:\n");
   for (k=0; (z=getchar( ))!='.'; )
      if (z!=' ') k++;
   printf("\n Количество символов=%d",k);
}   /* Конец программы       */
```

В заголовке цикла **for** выражение z=**getchar( )** заключено в скобки, так как операция присваивания (см. табл. 1.4) имеет более низкий ранг, чем операции сравнения. Если скобки опустить, то последовательность операций будет такой: функция **getchar( )** введет значение символа и выполнит его сравнение с символом '.'. Результат сравнения присваивается переменной z. По смыслу же необходимо введенный символ присвоить переменной z и сравнить его с точкой.

Результат выполнения программы:

```
Напишите предложение с точкой в конце:
1   2   3   4   5   6   7   8   9   0. <Enter>
Количество символов=10
```

Внутренние коды и упорядоченность символов. В языке принято соглашение, что везде, где синтаксис позволяет использовать целые числа, можно использовать и символы, то есть данные типа **char**, которые при этом представляются числовыми значениями своих внутренних кодов. Такое соглашение дает возможность сравнительно просто упорядочивать символы, обращаясь с ними как с целочисленными величинами. Например, внутренние коды десятичных цифр в таблицах кодов ASCII упорядочены по числовым значениям, поэтому несложно перебирать символы десятичных цифр в нужном порядке.

Следующая программа печатает цифры от 0 до 9 и шестнадцатеричные представления их внутренних кодов.

```
/* Печать десятичных цифр: */
#include <stdio.h>
void main( )
{
   char z;
   for (z='0'; z<='9'; z++)
     {
      if (z=='0' || z=='5') printf("\n");
      printf(" %c-%x   ", z, z);
     }
} /* Конец программы */
```

Результат выполнения программы:

```
0-30    1-31    2-32    3-33    4-34
5-35    6-36    7-37    8-38    9-39
```

Обратите внимание на то, что символьная переменная z является операндом арифметической операции '++', выполняемой над числовым представлением ее внутреннего кода. Для изображения значения символов в форматной строке функции **printf( )** используется спецификация **%c**. Шестнадцатеричные коды выводятся с помощью спецификации **%x**. В приложении 1 можно посмотреть, что именно такими являются шестнадцатеричные коды символов. Для '0' – код 30, для '1' – код 31 и т. д.

Воспользовавшись упорядоченностью внутренних кодов букв латинского алфавита, очень просто его напечатать:

```
/* Печать латинского алфавита:   */
#include <stdio.h>
void main( )
{
   char z;
   for (z='A'; z<='Z'; z++)
      printf("%c",z);
} /* Конец программы   */
```

Результат выполнения программы:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Строки, или строковые константы. В программе строки, или строковые константы, представляются последовательностью изображений символов, заключенной в кавычки (не в апострофы), например «любые_символы». Среди символов строки могут быть эскейп-последовательности, соответствующие кодам неизображаемых (специальных) символьных констант.

Примеры строк:

```
"1234567890"
"\t Состав президиума"
" Начало строки \n и конец строки"
```

Строки уже многократно встречались в функциях **printf( )** и **scanf( )**. Однако у строк есть некоторые особенности. Транслятор отводит каждой строке отдельное место в памяти ЭВМ даже в тех случаях, когда несколько строк полностью совпадают (стандарт языка Си предполагает, что в конкретных реализациях это правило может не выполняться). Размещая строку в памяти, транслятор автоматически добавляет в ее конце символ '\0', то есть нулевой байт. В записи строки может быть и один символ: «А», однако в отличие от символьной константы 'А' (использованы апострофы) длина строки «А» равна двум байтам. Объясняется это тем, что, в отличие от других языков (например, от Паскаля), в языке Си нет отдельного типа для строк. Принято, что строка – это массив символов, то есть она всегда имеет тип **char**[ ]. Таким образом, строка считается значением типа «массив символов». Количество элементов в таком массиве на 1 больше, чем в изображении соответствующей строковой константы, так как в конец строки добавлен нулевой байт '\0', называемый терминальным символом.

Присвоить значение массиву символов (то есть строке) с помощью обычного оператора присваивания нельзя. Поместить строку в массив можно либо с помощью инициализации (при определении символьного массива), либо с помощью функций ввода. В функции **scanf( )** или **printf( )** для символьных строк используется спецификация преобразования **%s**.

При определении массива типа **char** с одновременной инициализацией можно не указывать пределы изменения индекса. Сказанное иллюстрирует следующая программа:

```
/* Печать символьной строки   */
#include <stdio.h>
void main( )
{
   char B[  ]="Сезам, откройся!";
   printf("%s",B);
} /* Конец программы */
```

Результат выполнения программы:

```
Сезам, откройся!
```

В программе длина массива В – 17 элементов, то есть длина строки, помещаемой в массив (16 символов), плюс терминальный

символ. Именно 17 байтов выделяется при инициализации массива в приведенном примере. Инициализация массива символов с помощью строковой константы представляет собой сокращенный вариант инициализации массива и введена в язык для упрощения. Можно воспользоваться обычной инициализацией, поместив начальные значения элементов массива в фигурные скобки и не забыв при этом поместить в конце списка начальных значений терминальный символ. Таким образом, в программе была бы допустима такая инициализация массива B:

```
char B[ ]={'С','е','з','а','м',',',' ','о','т',
           'к','р','о','й','с','я','!','\0' };
```

Соглашение о признаке окончания строки нужно соблюдать, формируя в программах строки из отдельных символов. В качестве примера рассмотрим следующую задачу: «Ввести предложение, заканчивающееся точкой, слова в котором отделены пробелами. Напечатать последнее слово предложения».

Анализ условия задачи позволяет выявить следующие ошибочные ситуации и особые случаи: отсутствие точки в конце предложения; пробел или пробелы перед первым словом предложения; несколько пробелов между словами; пробелы перед завершающей предложение точкой; отсутствие слов в предложении (только точка). Чтобы не усложнять решение задачи, примем соглашение о том, что вводимое предложение всегда размещается на одной строке дисплея, то есть длина его не превышает 80 символов. Это позволит легко выявлять отсутствие точки в конце предложения и ограничивает длину любого слова предложения. Чтобы учесть особые ситуации с пробелами, необходимо при анализе очередного введенного символа (переменная s) рассматривать и предыдущий символ (переменная ss). Для выявления отсутствия слов в предложении будем вычислять длину k каждого очередного вводимого слова. Если k==0, а вводится символ '.', то это признак пустого предложения.

Текст программы:

```
/* Напечатать последнее слово в предложении */
#include <stdio.h>
void main( )
{
   char s,ss;   /* s - вводимый символ   */
   /* ss - предыдущий введенный символ */
```

```
   char A[80]; /* Массив для слова */
   int i,k;    /* k - длина слова */
   printf("Напишите предложение с точкой в "
          "конце:\n");
   for (i=0,s=' ',k=0; i<=79; i++)
   {
      ss=s; s=getchar( );
      if (s==' ') continue;
      if (s=='.') break;
      if (ss==' ') k=0;
      A[k]=s; k++;
   }
/*Выход по точке или по окончании ввода строки*/
   if (i==80 || k==0)
        printf(" Неверное предложение \n");
   else
   {
      A[k]='\0'; /* Конец строки */
      printf("Последнее слово: %s",A);
   }
} /* Конец программы    */
```

Чтение вводимых данных выполняется посимвольно в цикле с параметром i. Если вводится пробел, то оператор **continue** вызывает переход к следующей итерации. Если введена точка, то цикл прерывается. При этом в первых k элементах массива A[] запоминается последнее слово предложения. Если введенный символ отличен от точки и пробела, то анализируется предыдущий символ. Если это пробел, то начинается ввод следующего слова и устанавливается нулевое значение k. В следующих операторах введенный символ записывается в k-й элемент массива A и k увеличивается на 1. Выход из цикла возможен при появлении точки или после ввода 80 символов. Последнее выполняется при отсутствии в предложении точки. В этом ошибочном случае i==80. Когда в предложении нет слов, k остается равным нулю. Если i<80 и k не равно 0, то в k-й элемент массива A записывается признак конца строки, и она как единое целое выводится на печать.

Результат выполнения программы:

```
Напишите предложение с точкой в конце:
Орфографический словарь. <Enter>
Последнее слово: словарь
```

Как еще раз иллюстрирует приведенная программа, работа с символьными строками – это работа с массивами типа **char**. При этом следует обращать внимание на обязательное присутствие в конце строки (в последнем занятом элементе массива) символа '\0' и не допускать его случайного уничтожения при обработке.

В качестве следующей задачи рассмотрим проверку вводимой цифровой информации: «Необходимо, введя строку, напечатать порядковый номер (позицию) каждого символа, который отличен от пробела или цифры». Задачу решает следующая программа:

```
/* Проверка вводимых числовых данных: */
#include <stdio.h>
void main( )
{
   char z[ ]="0123456789 ";
   char s;
   int i,j;
   printf("Введите строку символов:\n ");
   for (i=1; (s=getchar( ))!='\n'; i++)
   {
    for (j=0; j<11; j++)
       if (s == z[j]) break;
    if (j == 11)
         printf("Ошибка в символе %c с номером "
                "%d\n", s,i);
   }  /* Конец цикла ввода */
}  /* Конец программы */
```

В программе (в цикле по i) выполняется ввод отдельных символов до появления неизображаемого символа '\n' – перевод строки. Каждый введенный символ s сравнивается в цикле по j с элементами массива z, содержащего все допустимые символы. Если выявлено совпадение, то при выходе из цикла j не равно 11. При естественном завершении цикла (не найдено совпадения) j оказывается равным 11, и печатается номер ошибочного символа.

Пример результата выполнения программы:

```
Введите строку символов:    124E-22   16<Enter>
Ошибка в символе E с номером 4
Ошибка в символе - с номером 5
```

Строки и указатели. Мы уже рассмотрели взаимосвязь указателей с массивами, так что возможность связать указатель с каждой строкой нас уже не удивит. Рассмотрим следующие два определения:

```
char A[20];
/* Массив, в который можно записать строку */
char *B;
/* Указатель, с которым можно связать строку */
```

Массив A получает память автоматически при обработке определения. Строке, с которой мы хотим связать указатель B, память при обработке определения указателя не выделяется. Поэтому если далее следуют операторы

```
scanf(«%s»,A);   /* Оператор верен */
scanf(«%s»,B);   /* Оператор не корректен */
```

то первый из них допустим, а второй приводит к ошибке во время выполнения программы – попытка ввода в неопределенный участок памяти. Для исправления ошибки с указателем B нужно связать некоторый участок памяти. Для этого существуют несколько возможностей. Во-первых, переменной B можно присвоить адрес уже определенного символьного массива. Во-вторых, указатель B можно «настроить» на участок памяти, выделяемый с помощью средств динамического распределения памяти (см. табл. 4.1). Например, оператор

```
B=(char *) malloc(80);
```

выделяет 80 байт и связывает этот блок памяти с указателем B. Только теперь применение приведенного выше оператора ввода допустимо. Прототипы функции **malloc( )** и других функций распределения памяти находятся в файле **stdlib.h**.

С помощью указателей типа **char*** удобно получать доступ к строкам в виде массивов символов. Типичная задача – обработка слов или предложений, каждое из которых представлено в массиве типа **char** в виде строки (то есть в конце представления предложения или слова находится терминальный символ '\0'). Использование указателей, а не массивов с фиксированными размерами, особенно целесообразно, когда предложения или слова должны быть разной длины. В следующей программе определен и при инициализации

связан с набором строк одномерный массив point[ ] указателей типа **char***. Функция **printf( )** и спецификатор преобразования **%s** допускают использование в качестве параметра указателя на строку. При этом на дисплей (в выходной поток) выводится не значение указателя point[i], а содержимое адресуемой им строки.

Текст программы:

```
#include   <stdio.h>
void  main(  )
{
   char  *  point[  ]={"thirteen","fourteen",
        "fifteen","sixteen","seventeen","eighteen","nineteen"};
   int  i,n;
   n=sizeof(point)/sizeof(point[0]);
   for(i=0;i<n;i++)
        printf("\n%s",point[i]);
}
```

Результат выполнения программы:

```
thirteen
fourteen
fifteen
sixteen
seventeen
eighteen
nineteen
```

Контрольные вопросы

1. Дайте определение указателя.
2. Объясните соотношение между именем переменной и адресом.
3. Каким образом можно получить адрес переменной в явном виде?
4. Как иначе называется операция обращения к переменной по адресу?
5. Перечислите операции над указателями.
6. Нужно ли указывать тип объекта при определении указателя на него?
7. Опишите механизм «приведения типов».
8. Что происходит при добавлении целочисленного значения n к указателю, адресующему некоторый элемент массива?

9. Различаются ли свойства указателей на объекты разных классов памяти?
10. Для чего служит препроцессорная константа NULL?
11. Применимы ли к указателям операции сравнения?
12. В памяти какого типа может быть определен указатель?
13. В каком случае указатели получают начальные значения?
14. Можно ли указателю типа **void*** присвоить адрес любого объекта?
15. Необходимо ли приведение типов, для того чтобы с помощью разыменования получить доступ к объекту, адресованному указателем типа **void***?
16. Является ли некорректным непосредственное разыменование указателя типа **void***?
17. Одинаковы или различны размеры участков памяти, занимаемых указателями разных типов, в конкретной реализации языка Си?
18. Зависит ли размер участка памяти, доступного с помощью разыменования адреса (указателя), от его типа?
19. Возможна ли замена адреса объекта указателем, адресующим этот объект, в выражениях?
20. Дайте определение массива.
21. Какой результат будет получен при разыменовании имени массива?
22. Каков результат вычитания из адреса любого элемента массива имени массива?
23. С помощью указателей какого типа удобно получать доступ к строкам в виде массива символов?
24. Что вычислит операция **sizeof**, применяемая к указателю, адресующему строку?
25. Укажите результат операции **sizeof**, применяемой к строковой константе.
26. Можно ли утверждать, что любая часть символьного массива, не содержащая символов '\0' и ограниченная справа этим символом, воспринимается как символьная строка в стиле Си?
27. Какой объем памяти будет получен при определении указателя типа **char** **?
28. Для чего можно применить массив нетипизированных указателей **(void** *)?

Глава 5
ФУНКЦИИ

5.1. Общие сведения о функциях

Определение функции. В соответствии с синтаксисом в языке Си определены три производных типа: массив, указатель, функция. В этой главе рассмотрим функции.

О функциях в языке Си нужно говорить, рассматривая это понятие с двух сторон. Во-первых, функция, как мы только что сказали, – это один из производных типов (наряду с массивом и указателем). С другой стороны, функция – это минимальный исполняемый модуль программы на языке Си. Синонимами этого второго понятия в других языках программирования являются процедуры, подпрограммы, подпрограммы-функции, процедуры-функции. Все функции в языке Си имеют рекомендуемый стандартами языка единый формат определения:

тип имя_функции (спецификация_параметров)
тело_функции

Первая строка – это заголовок функции.

Здесь *тип* – либо **void** (для функций, не возвращающих значения), либо обозначение типа возвращаемого функцией значения. В предыдущих главах рассмотрены функции, возвращающие значения базовых типов (**char, int, double** и т. д.).

Имя_функции – либо **main** для основной (главной) функции программы, либо произвольно выбираемое программистом имя (идентификатор), не совпадающее со служебными словами и с именами других объектов (и функций) программы.

Спецификация_параметров – это либо пусто, либо список параметров, каждый элемент которого имеет вид:

обозначение_типа имя_параметра

Примеры спецификаций параметров уже приводились для случаев, когда параметры были базовых типов или массивами. Список

параметров функции может заканчиваться запятой с последующим многоточием «...». Многоточие обозначает возможность обращаться к функции с большим количеством параметров, чем явно указано в спецификации параметров. Такая возможность должна быть «подкреплена» специальными средствами в теле функции. В следующих параграфах этой главы особенности подготовки функций с переменным количеством аргументов будут подробно рассмотрены. Сейчас отметим, что мы уже хорошо знакомы с двумя библиотечными функциями, допускающими изменяемое количество аргументов. Вот их заголовки:

int printf (**char** * format, ...)
int scanf (**char** * format, ...)

Указанные функции форматированного вывода и форматированного ввода позволяют применять теоретически неограниченное количество аргументов. Обязательным является только параметр **char** * format – «форматная строка», внутри которой с помощью спецификаций преобразования определяется реальное количество аргументов, участвующих в обменах.

Тело_функции – это часть определения функции, ограниченная фигурными скобками и непосредственно размещенная вслед за заголовком функции. Тело функции может быть либо составным оператором, либо блоком. Напоминаем, что, в отличие от составного оператора, блок включает определения объектов (переменных, массивов и т. д.). Особенность языка Си состоит в невозможности определить внутри тела функции иную функцию. Другими словами, определения функций не могут быть вложенными.

Обязательным, но не всегда явно используемым оператором тела функции является оператор возврата из функции в точку вызова, имеющий две формы:

return;
return выражение;

Первая форма соответствует завершению функции, не возвращающей никакого значения, то есть функции, перед именем которой в ее определении указан тип **void**. *Выражение* во второй форме оператора **return** должно иметь тип, указанный перед именем функции в ее определении, либо иметь тип, допускающий автоматическое преобразование к типу возвращаемого функцией значения.

Мы отметили обязательность оператора возврата из тела функции, однако оператор **return** программист может явно не использо-

вать в теле функции, возвращающей значение типа **void** (ничего не возвращающей). В этом случае компилятор автоматически добавляет оператор **return** в конец тела функции перед закрывающейся фигурной скобкой «**}**».

Итак, в языке Си допустимы функции и с параметрами, и без параметров, функции, возвращающие значения указанного типа и ничего не возвращающие.

Описание функции и ее тип. Для корректного обращения к функции сведения о ней должны быть известны компилятору, то есть до вызова функции в том же файле стандартом рекомендуется помещать ее определение или описание. Для функции описанием служит ее прототип:

тип имя_функции (спецификация_параметров);

В отличие от заголовка функции, в ее прототипе могут не указываться имена параметров. Например, допустимы и эквивалентны следующие прототипы одной и той же функции:

```
double f (int n, float x);
double f (int, float);
```

Вызов функции. Для обращения к функции используется выражение с операцией «круглые скобки»:

обозначение_функции (список_аргументов)

Операндами операции '**( )**' служат *обозначение_функции* и *список_аргументов*. Наиболее естественное и понятное *обозначение_функции* – это ее имя. Кроме того, функцию можно обозначить, разыменовав указатель на нее. Так как указатели на функции мы еще не ввели (см. следующие параграфы этой главы), то ограничимся пока для обозначения функций их именами.

Список аргументов – это список выражений, количество которых равно числу параметров функции (исключение составляют функции с переменным количеством аргументов). Соответствие между параметрами и аргументами устанавливается по их взаимному расположению в списках. Порядок вычисления значений аргументов (слева направо или справа налево) стандарт языка Си не определяет.

Между параметрами и аргументами должно быть соответствие по типам. Лучше всего, когда тип аргумента совпадает с типом параметра. В противном случае компилятор автоматически добавляет команды преобразования типов, что возможно только в том случае,

если такое приведение типов допустимо. Например, пусть определена функция с прототипом:

```
int g(int, long);
```

Далее в программе использован вызов:

```
g(3.0+m, 6.4e+2)
```

Оба аргумента в этом вызове имеют тип **double**. Компилятор, ориентируясь на прототип функции, автоматически предусмотрит такие преобразования:

```
g((int)(3.0+m), (long) 6.4e+2)
```

Так как вызов функции является выражением, то после выполнения операторов тела функции в точку вызова возвращается некоторое значение, тип которого строго соответствует типу, указанному перед именем функции в ее определении (и прототипе). Например, функция

```
float ft(double  x, int n)
{
   if (x<n) return x;
   return n;
}
```

всегда возвращает значение типа **float**. В выражения, помещенные в операторы **return**, компилятор автоматически добавит средства для приведения типов, то есть получим (невидимые программисту) операторы:

```
return (float) x;
return (float) n;
```

Особое внимание нужно уделить правилам передачи параметров при обращении к функциям. Синтаксис языка Си предусматривает только один способ передачи параметров – передачу по значениям. Это означает, что параметры функции локализованы в ней, то есть недоступны вне определения функции, и никакие операции над параметрами в теле функции не изменяют значений аргументов.

Передача параметров по значению предусматривает следующие шаги:

1. При компиляции функции (точнее, при подготовке к ее выполнению) выделяются участки памяти для параметров, то есть параметры оказываются внутренними объектами функции. При этом для параметров типа **float** формируются объекты типа **double**, а для параметров типов **char** и **short int** создаются объекты типа **int**. Если параметром является массив, то формируется указатель на начало этого массива и он служит представлением массива-параметра в теле функции.
2. Вычисляются значения выражений, использованных в качестве аргументов при вызове функции.
3. Значения выражений-аргументов заносятся в участки памяти, выделенные для параметров функции. При этом **float** преобразуется в **double**, а **char** и **short int** – в тип **int** (см. п. 1).
4. В теле функции выполняется обработка с использованием значений внутренних объектов-параметров, и результат передается в точку вызова функции как возвращаемое ею значение.
5. Никакого влияния на аргументы (на их значения) функция не оказывает.
6. После выхода из функции освобождается память, выделенная для ее параметров.

Вызов функции всегда является выражением, однако размещение такого выражения в тексте программы зависит от типа возвращаемого функцией значения. Если в качестве типа возвращаемого значения указан тип **void**, то функция является функцией без возвращаемого результата. Такая функция не может входить ни в какие выражения, требующие значения, а должна вызываться в виде отдельного выражения-оператора:

имя_функции (список_аргументов);

Например, следующая функция возвращает значение типа **void**:

```
void print(int gg, int mm, int dd)
{
   printf("\n год: %d",gg);
   printf(",\t месяц: %d,",mm);
   printf(",\t день: %d.", dd);
}
```

Обращение к ней

```
print(1966, 11, 22);
```

приведет к такому выводу на экран:

```
год: 1966,    месяц: 11,    день: 22.
```

Может оказаться полезной и функция, которая не только не возвращает никакого значения (имеет возвращаемое значение типа **void**), но и не имеет параметров. Например, такая:

```
#include <stdio.h>
void Real_Time (void)
{
   printf("\n Текущее время: %s", _ _TIME_ _
         " (час: мин: сек.)");
}
```

При обращении

```
Real_Time ( );
```

в результате выполнения функции будет выведено на экран дисплея сообщение:

```
Текущее время: 14:16:25  (час:  мин:  сек.)
```

5.2. Указатели в параметрах функций

Указатель-параметр. В предыдущем параграфе мы достаточно подробно рассмотрели механизм передачи параметров при вызове функций. Схема передачи параметров по значениям не оставляет никаких надежд на возможность непосредственно изменить аргумент за счет выполнения операторов тела функции. И это действительно так. Объект вызывающей программы, использованный в качестве аргумента, не может быть изменен из тела функции. Однако существует косвенная возможность изменять значения объектов вызывающей программы действиями в вызванной функции. Эту возможность обеспечивает аппарат указателей. С помощью указателя в вызываемую функцию можно передать адрес любого объекта из

вызывающей программы. С помощью выполняемого в тексте функции разыменования указателя мы получаем доступ к адресуемому указателем объекту из вызывающей программы.

Тем самым, не изменяя самого параметра (указатель-параметр постоянно содержит только адрес одного и того же объекта), можно изменять объект вызывающей программы.

Продемонстрируем изложенную схему на простом примере:

```
#include <stdio.h>
void positive(int * m) /* Определение функции */
{
   *m = *m > 0 ? *m : -*m;
}
void main()
{
   int k=-3;
   positive(&k);
   printf("\nk=%d", k);
}
```

Результат выполнения программы:

```
k=3
```

Параметр функции positive( ) – указатель типа **int** *. При обращении к ней из основной программы **main( )** в качестве аргумента используется адрес &k переменной типа **int**. Внутри функции значение аргумента (то есть адрес &k) «записывается» в участок памяти, выделенный для указателя **int** *m. Разыменование *m обеспечивает доступ к тому участку памяти, на который в этот момент «смотрит» указатель m. Тем самым в выражении

```
*m = *m>0 ? *m :- *m
```

все действия выполняются над значениями той переменной основной программы (**int** k), адрес которой (&k) использован в качестве аргумента.

Рисунок 5.1 графически иллюстрирует взаимосвязь функции positive( ) и основной программы. При обращении к функции positive( ) она присваивает абсолютное значение той переменной, адрес которой использован в качестве ее аргумента.

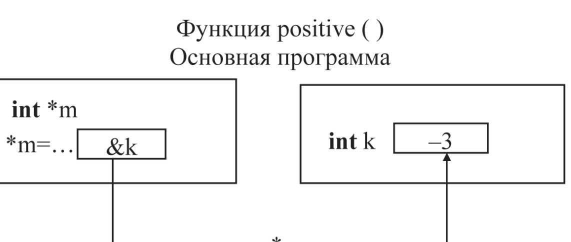


Рис. 5.1. *Схема «настройки» параметра-указателя*

Пояснив основные принципы воздействия с помощью указателей из тела функции на значения объектов вызывающей программы, напомним, что этой возможностью мы уже неоднократно пользовались, обращаясь к функции **scanf( )**. Функция **scanf( )** имеет один обязательный параметр – форматную строку – и некоторое количество необязательных параметров, каждый из которых должен быть адресом того объекта, значение которого вводится с помощью функции **scanf( )**. Например:

```
long l;
double d;
scanf("%ld%le", &l, &d);
```

Здесь в форматной строке спецификаторы преобразования **%ld** и **%le** обеспечивают ввод (чтение) от клавиатуры соответственно значений типов **long int** и **double**. Передача этих значений переменным **long** l и **double** d обеспечивается с помощью их адресов &l и &d, которые используются в качестве аргументов функции **scanf( )**.

Имитация подпрограмм. Подпрограммы отсутствуют в языке Си, однако если использовать обращение к функции в виде оператора-выражения, то получим аналог оператора вызова подпрограммы. (Напомним, что выражение приобретает статус оператора, если после него стоит точка с запятой.) Выше были приведены в качестве примеров функции

void print(**int** gg, **int** mm, **int**) и **void** Real_Time(**void**),

каждая из которых очень похожа на подпрограммы других языков программирования.

К сожалению, в языке Си имеется еще одно препятствие для непосредственного использования функции в роли подпрограммы – это рассмотренная выше передача параметров только значениями, то есть передаются значения переменных, а не их адреса. Другими словами, в результате выполнения функции нельзя изменить значения ее аргументов. Таким образом, если Z( ) – функция для вычисления периметра и площади треугольника по длинам сторон E, F, G, то невозможно, записав оператор-выражение Z(E, F, G, PP, SS);, получить результаты (периметр и площадь) в виде значений переменных PP и SS. Так как параметры передаются только значениями, то после выполнения функции Z( ) значения переменных PP и SS останутся прежними.

Возможно следующее решение задачи. В определении функции параметры, с помощью которых результаты должны передаваться из функции в точку вызова, специфицируются как указатели. Тогда с помощью этих указателей может быть реализован доступ из тела функции к тем объектам вызывающей программы, которые адресуются параметрами-указателями. Для пояснения сказанного рассмотрим следующую программу:

```
#include <stdio.h>
void main()
{
   float x,y;
   /* Прототип функции */
   void aa(float *, float *);
   printf("\n Введите: x=");
   scanf("%f",&x);
   printf(" Введите: y=");
   scanf("%f",&y);
   /* Вызов функции   */
   aa(&x,&y);
printf(" \n Результат: x=%f y=%f", x, y);
}
/* Функция, меняющая местами значения переменных,
   на которые указывают фактические параметры: */
void aa(float * b, float * c)
/* b и c - указатели */
{
   float e; /* Вспомогательная переменная */
   e=*b;
   *b=*c;
   *c=e;
}
```

В основной программе описаны переменные x, y, значения которых пользователь вводит с клавиатуры. Параметрами функции aa( ) служат указатели типа **float** *. Задача функции – поменять местами значения тех переменных, на которые указывают ее параметры-указатели. При обращении к функции aa( ) адреса переменных x, y используются в качестве аргументов. Поэтому после выполнения программы возможен, например, такой результат:

```
Введите:  x=33.3      <Enter>
Введите:  y=66.6      <Enter>
Результат:  x=66.600000   y=33.300000
```

Имитация подпрограммы (функция) для вычисления периметра и площади треугольника:

```
#include <math.h> /* для функции sqrt( ) */
#include <stdio.h>
void main( )
{
   float x,y,z,pp,ss;
   /* Прототип: */
   int triangle(float, float, float, float *, float *);
   printf("\n Введите: x=");
   scanf("%f",&x);
   printf("\t   y=");
   scanf("%f",&y);
   printf("\t   z=");
   scanf("%f",&z);
   if (triangle(x,y,z,&pp,&ss) == 1)
      {
         printf(" Периметр = %f",pp);
         printf(", площадь = %f",ss);
      }
   else
   printf("\n Ошибка в данных ");
}
/* Определение функции: */
int  triangle(float a,float b, float c,float * perimeter,
              float * area)
{
    float e;
    *perimeter=*area=0.0;
    if (a+b<=c || a+c<=b || b+c<=a)
```

```
        return 0;
    *perimeter=a+b+c;
    e=*perimeter/2;
    *area=sqrt(e*(e-a)*(e-b)*(e-c));
    return 1 ;
}
```

Пример результатов выполнения программы:

```
Введите   x=3   <Enter>
          y=4   <Enter>
          z=5   <Enter>
```

Периметр = 12.000000, площадь = 6.000000.

5.3. Массивы и строки как параметры функций

Массивы в параметрах. Если в качестве параметра функции используется обозначение массива, то на самом деле внутрь функции передается только адрес начала массива. Применяя массивы в качестве параметров для функций из главы 2, мы не отмечали этой особенности. Например, заголовок функции для вычисления скалярного произведения векторов выглядел так:

float Scalar_Product(**int** n, **float** a[ ], **float** b[ ]). . .

Но заголовок той же функции можно записать и следующим образом:

float Scalar_Product(**int** n, **float** *a, **float** *b). . .

Конструкции **float** b[ ]; и **float** *b; совершенно равноправны в спецификациях параметров. Однако в первом случае роль имени b как указателя не так явственна. Во втором варианте все более очевидно – b явно специфицируется как указатель типа **float** *.

В теле функции Scalar_Product( ) из главы 2 обращение к элементам массивов-параметров выполнялось с помощью индексированных элементов a[i] и b[i]. Однако можно обращаться к элементам массивов, разыменовывая соответствующие значения адресов, то есть используя выражения *(a+i) и *(i+b).

Так как массив всегда передается в функцию как указатель, то внутри функции можно изменять значения элементов массива-ар-

гумента, определенного в вызывающей программе. Это возможно и при использовании индексирования, и при разыменовании указателей на элементы массива.

Для иллюстрации указанных возможностей рассмотрим функцию, возводящую в квадрат значения элементов одномерного массива и вызывающую ее программу:

```
#include <stdio.h>
/* Определение функции: */
void quart(int n, float * x)
{
   int i;
   for(i=0;i<n;i++)
   /* Присваивание после умножения: */
   *(x+i)*=*(x+i);
}
void main()
{
   /* Определение массива: */
   float z[ ]={1.0, 2.0, 3.0, 4.0};
   int j;
   quart(4,z); /* Обращение к функции */
   /* Печать измененного массива */
   for(j=0;j<4;j++)
      printf("\nz[%d]=%f",j,z[j]);
}
```

Результат выполнения программы:

```
z[0]=1.000000
z[1[=4.000000
z[2]=9.000000
z[3]=16.000000
```

Чтобы еще раз обратить внимание на равноправие параметров в виде массива и указателя того же типа, отметим, что заголовок функции в нашей программе может быть и таким:

void quart **(int** n, **float** x [ ])

В теле функции разыменовано выражение, содержащее имя массива-параметра, то есть вместо индексированной переменной x[i] используется *(x+i). Эту возможность мы уже неоднократно отмечали. Более интересная возможность состоит в том, что можно изменять

внутри тела функции значение указателя на массив, то есть в теле цикла записать, например, такой оператор:

```
*x*=*x++;
```

Обратите внимание, что неверным для нашей задачи будет следующий вариант этого оператора:

```
*x++*=*x++;     /*ошибка!*/
```

В этом ошибочном случае «смещение» указателя x вдоль массива будет при каждой итерации цикла не на один элемент массива, а на 2, так как x изменяется и в левом, и в правом операндах операции присваивания.

Следует отметить, что имя массива внутри тела функции не воспринимается как константный (не допускающий изменений) указатель, однако такая возможность отсутствует в основной программе, где определен соответствующий массив-параметр. Если в цикле основной программы вместо значения z[j] попытаться использовать в функции **printf( )** выражение *z++, то получим сообщение об ошибке, то есть следующие операторы ***не верны***:

```
for (j=0; j<4; j++)
printf("\nz[%d]=%f", j, *z++);
```

Сообщение об ошибке при компиляции выглядит так:

```
Error. . . Lvalue required
```

Подводя итоги, отметим, что, с одной стороны, имя массива является константным указателем со значением, равным адресу нулевого элемента массива. С другой стороны, имя массива, использованное в качестве параметра, играет в теле функции роль обычного (неконстантного) указателя, то есть может использоваться в качестве леводопустимого выражения. Именно поэтому в спецификациях параметров эквивалентны, как указано выше, например, такие формы:

double x[ ]   и   **double** *x

Строки как параметры функций. Строки в качестве параметров могут быть специфицированы либо как одномерные массивы типа **char** [ ], либо как указатели типа **char** *. В обоих случаях с помощью

параметра в функцию передается адрес начала символьного массива, содержащего строку. В отличие от обычных массивов, для параметров-строк нет необходимости явно указывать их длину. Терминальный символ '\0', размещаемый в конце каждой строки, позволяет всегда определить ее длину, точнее, позволяет перебирать символы строки и не выйти за ее пределы. Как примеры использования строк в качестве параметров функций рассмотрим несколько функций, решающих типовые задачи обработки строк. Аналоги большинства из приводимых ниже функций имеются в библиотеке стандартных функций (см. приложение 3). Их прототипы и другие средства связи с этими функциями находятся в заголовочных файлах **string.h** и **stdlib.h**. Однако для целей темы, посвященной использованию строк в качестве параметров, удобно заглянуть «внутрь» таких функций.

Итак, в иллюстративных целях приведем определения функций для решения некоторых типовых задач обработки строк. Будем для полноты картины использовать различные способы задания строк-параметров.

Функция вычисления длины строки (в стандартной библиотеке ей соответствует функция **strlen( )**, см. приложение 3):

```
int len (char e[ ])
{
   int m;
   for (m=0; e[m]!='\0'; m++);
       return m;
}
```

В этом примере и в заголовке, и в теле функции нет даже упоминаний о родстве массивов и указателей. Однако компилятор всегда воспринимает массив как указатель на его начало, а индекс – как смещение относительно начала массива. Следующий вариант той же функции явно реализует механизм работы с указателями:

```
int len (char *s)
{
   int m;
   for (m=0; *s++!='\0'; m++)
       return m;
}
```

Для параметра-указателя s внутри функции выделяется участок памяти, куда записывается значение аргумента. Так как s не константа, то значение этого указателя может изменяться. Именно поэтому допустимо выражение s++.

Функция инвертирования строки-аргумента с параметром-массивом:

```
void invert(char e[ ])
{
   char s;
   int i, j, m;
   /*m - номер позиции символа '\0' в строке e */
   for (m=0; e[m]!='\0'; m++);
      for (i=0, j=m-1; i<j; i++, j--)
      {
       s=e[i];
       e[i]=e[j];
       e[j]=s;
      }
}
```

В определении функции invert( ) с помощью ключевого слова **void** указано, что функция не возвращает значения.

В качестве упражнения можно переписать функцию invert( ), заменив параметр-массив параметром-указателем типа **char***.

При выполнении функции invert( ) строка – параметр, например «сироп» превратится в строку «порис». При этом терминальный символ '\0' остается на своем месте в конце строки. Пример использования функции invert( ):

```
#include <stdio.h>
void main( )
{
   char ct[ ]="0123456789";
   /* Прототип функции: */
   void invert(char [ ]);
   /* Вызов функции: */
   invert(ct);
   printf("\n%s",ct);
}
```

Результат выполнения программы:

```
9876543210
```

Функция поиска в строке ближайшего слева вхождения другой строки (в стандартной библиотеке имеется подобная функция **strstr( )**, см. приложение 3):

```
/*Поиск строки СТ2 в строке СТ1 */
int index (char * СТ1, char * СТ2)
{
   int i, j, m1, m2;
   /* Вычисляются m1 и m2 - длины строк */
   for (m1=0; СТ1[m1] !='\0'; m1++);
   for (m2=0; СТ2[m2] !='\0'; m2++);
   if (m2>m1)
       return -1;
   for (i=0; i<=m1-m2; i++)
   {
    for (j=0; j<m2; j++) /*Цикл сравнивания */
       if (СТ2[j] !=СТ1[i+j])
           break;
    if (j==m2)
        return i;
   } /* Конец цикла по i */
   return -1;
}
```

Функция index( ) возвращает номер позиции, начиная с которой СТ2 полностью совпадает с частью строки СТ1. Если строка СТ2 не входит в СТ1, то возвращается значение –1.

Пример обращения к функции index( ):

```
#include <stdio.h>
void main( )
{
   char C1[ ]="сумма масс";
   /* Прототип функции: */
   int index(char [ ], char [ ]);
   char C2[ ]="ма";
   char C3[ ]="ам";
   printf("\nДля %s индекс=%d",C2,index(C1,C2));
   printf("\nДля %s индекс=%d",C3,index(C1,C3));
}
```

Результат выполнения программы:

```
Для ма индекс=3
Для ам индекс=-1
```

В функции index( ) параметры специфицированы как указатели на тип **char**, а в теле функции обращение к символам строк выполняется с помощью индексированных переменных. Вместо параметров-указателей подставляют в качестве аргументов имена символьных массивов C1[ ], C2[ ], C3[ ]. Никакой неточности здесь нет: к массивам допустимы обе формы обращения – и с помощью индексированных переменных, и с использованием разыменования указателей.

Функция сравнения строк. Для сравнения двух строк можно написать следующую функцию (в стандартной библиотеке имеется близкая к этой функция **strcmp( )**, см. приложение 3):

```
int row(char C1[ ], char C2[ ])
{
   int i,m1,m2; /* m1,m2 - длины строк C1,C2 */
   for (m1=0;*(C1+m1)= '\0'; m1++);
   for (m2=0;*(C2+m2)= '\0'; m2++);
   if (m1!=m2) return -1;
   for (i=0; i<m1; i++)
      if (*C1++ != *C2++) return (i+1);
   return 0;
}
```

В теле функции обращение к элементам массивов-строк реализовано через разыменование указателей. Функция row( ) возвращает: значение –1, если длины строк-аргументов C1, C2 различны; 0 – если все символы строк совпадают. Если длины строк одинаковы, но символы не совпадают, то возвращается порядковый номер (слева) первых несовпадающих символов.

Особенность функции row( ) – спецификация параметров как массивов и обращение к элементам массивов внутри тела функции с помощью разыменования. При этом за счет операций C1++ и C2++ изменяются начальные «настройки» указателей на массивы. Одновременно в той же функции к тем же массивам-параметрам выполняется обращение и с помощью выражений *(C1+m1) и *(C2+m2), при вычислении которых значения C1 и C2 не меняются.

Функция соединения строк. Следующая функция позволяет «присоединить» к первой строке-аргументу вторую строку-аргумент (в стандартной библиотеке есть подобная функция **strncat( )**, см. приложение 3):

```
/* Соединение (конкатенация) двух строк: */
void conc(char *C1, char *C2)
```

```
{
   int i,m;   /* m - длина 1-й строки    */
   for (m=0; *(C1+m)!='\0'; m++);
   for (i=0; *(C2+i)!='\0'; i++)
         *(C1+m+i)=*(C2+i);
   *(C1+m+i)='\0';
}
```

Результат возвращается как значение той строки, которая адресована аргументом, замещающим первый параметр C1. Обратите внимание на то, что при использовании функции conc( ) длина строки, заменяющей параметр C1, должна быть достаточной для приема результирующей строки.

Функция выделения подстроки. Для выделения из строки, представленной параметром C1, фрагмента заданной длины (подстроки) можно предложить такую функцию:

```
void substr(char *C1, char *C2, int n, int k)
/* C1 - исходная строка                    */
/* C2 - выделяемая подстрока */
/* n   - начало выделяемой подстроки */
/* k   - длина выделяемой подстроки   */
{
   int i,m;   /* m - длина исходной строки */
   for (m=0; C1[m]!='\0'; m++);
         if (n<0 || n>m || k<0 || k>m-n)
              {
                 C2[0]='\0';
                 return;
              }
   for (i=n; i<k+n; i++)
       C2[i-n]=C1[i-1];
   C2[i-n]='\0';
   return;
}
```

Результат выполнения функции – адресованная параметром C2 строка из k символов, выделенных из строки, адресованной параметром C1, начиная с символа, имеющего номер n. При неверном сочетании значений аргументов возвращается пустая строка.

Функция копирования содержимого строки. Так как в языке Си отсутствует оператор присваивания для строк, то полезно иметь

специальную функцию, позволяющую «переносить» содержимое строки в другую строку (такая функция **strcpy( )** есть в стандартной библиотеке, но она имеет другое возвращаемое значение):

```
/* Копирование содержимого строки C2 в C1 */
void copy(char *C1, char *C2)
/* C2 - оригинал, C1 - копия */
{
   int i;
   for (i=0; C2[i]!='\0'; i++)
      C1[i]=C2[i];
   C1[i]='\0';
}
```

Пример использования функции copy( ):

```
#include <stdio.h>
void main()
{
   char X[ ]="SIC TRANSIT GLORIA MUNDI!";
   /* Прототип функции: */
   void copy(char[ ], char[ ]);
   char B[100];
   /* Обращение к функции: */
   copy (B,X);
   printf("%s\n", B);
}
```

Результат выполнения тестовой программы:

```
SIC TRANSIT GLORIA MUNDI!
```

Другой вариант функции копирования строки:

```
void copy(char C1[ ], char C2[ ])
{
   int i=0;
   do
      {
       C1[i] = C2[i];
      }
   while (C1[i++]!='\0');
}
```

В третьем варианте той же функции операция присваивания перенесена в выражение-условие продолжения цикла. Ранги операций требуют заключения выражения-присваивания в скобки:

```
void copy(char *C1, char *C2)
{
   int i=0;
   while ((C1[i] = C2[i]) != '\0')
       i++;
}
```

Так как ненулевое значение в выражении после **while** считается истинным, то явное сравнение с '\0' необязательно и возможно следующее упрощение функции:

```
void copy(char * C1, char * C2)
{
   int i=0;
   while (C1[i] = C2[i])
       i++;
}
```

И наконец, наиболее короткий вариант:

```
void copy (char * C1, char * C2)
{
   while (*C1++ = *C2++);
}
```

В заключение параграфа продемонстрируем возможности библиотеки стандартных функций для задач обработки строк, точнее, перепишем функцию конкатенации строк, используя библиотечную функцию **strlen( )**, позволяющую вычислить длину строки-параметра.

Функция для сцепления (конкатенации) строк:

```
#include <string.h>
void concat(char * C1, char * C2)
{
   int m, i;
   m=strlen (C1);    i=0;
   while ((*(C1+m+i) = *(C2+i))!='\0')
       i++;
}
```

Вместо функции **strlen( )** можно было бы использовать определенную выше функцию len( ). При обращении к функции concat( ), а также при использовании похожей на нее библиотечной функции strcat( ) нужно, чтобы длина строки, использованной в качестве первого параметра, была достаточной для размещения результирующей (объединенной) строки. В противном случае результат выполнения непредсказуем.

Резюме по строкам-параметрам. Часть из приведенных функций для работы со строками, а именно функции: конкатенации строк conc( ), инвертирования строки invert( ), копирования одной строки в другую copy( ), выделения подстроки substr( ); – предусматривают изменение объектов вызывающей программы. Это возможно и допустимо только потому, что параметрами являются указатели. Это либо имена массивов, либо указатели на строки. Как уже говорилось, параметры-указатели позволяют функции получить непосредственный доступ к объектам той программы, из которой функция вызвана. В приведенных определениях функций этот доступ осуществляется как с помощью индексированных переменных, так и с помощью разыменования указателей. На самом деле компилятор языка Си, встречая, например, такое обращение к элементу массива s[i], всегда заменяет его выражением *(s+i), то есть указателем s со смещением i.

Такую замену, то есть использование указателей со смещениями вместо индексированных переменных, можно явно использовать в определениях функций. Иногда это делает нагляднее связь между функцией и вызывающей ее программой.

5.4. Указатели на функции

Указатели при вызове функций. До сих пор мы рассматривали функцию как минимальный исполняемый модуль программы, обмен данными с которым происходит через набор параметров и с помощью значений, возвращаемых функцией в точку вызова. Теперь перейдем к вопросу о том, почему в языке Си функция введена как один из производных типов. Необходимость в таком типе связана, например, с задачами, в которых функция (или ее адрес) должна выступать в качестве параметра другой функции или в качестве значения, возвращаемого другой функцией. Обратимся к уже рассмотренному ранее выражению «вызов функции» (точнее, к более общему варианту вызова):

обозначение_функции (список_аргументов)

где *обозначение_функции* (только в частном случае это идентификатор) должно иметь тип «указатель на функцию, возвращающую значение конкретного типа».

В соответствии с синтаксисом языка указатель на функцию – это выражение или переменная, используемые для представления адреса функции. По определению, указатель на функцию содержит адрес первого байта или первого слова выполняемого кода функции (арифметические операции над указателями на функции запрещены).

Самый употребительный указатель на функцию – это ее имя (идентификатор). Именно так указатель на функцию вводится в ее определении:

тип имя_функции (спецификация_параметров)
тело_функции

Прототип

тип имя_функции (спецификация_параметров);

также описывает имя функции именно как указатель на функцию, возвращающую значение конкретного типа.

Имя_функции в ее определении и в ее прототипе – указатель-константа. Он навсегда связан с определяемой функцией и не может быть «настроен» на что-либо иное, чем ее адрес. Для идентификатора *имя_функции* термин «указатель» обычно не используют, а говорят об имени функции.

Указатель на функцию как переменная вводится отдельно от определения и прототипа какой-либо функции. Для этих целей используется конструкция:

*тип (*имя_указателя) (спецификация_параметров);*

где *тип* – определяет тип возвращаемого функцией значения; *имя_указателя* – идентификатор, произвольно выбранный программистом; *спецификация_параметров* – определяет состав и типы параметров функции.

Например, запись

```
int (*point) (void);
```

определяет указатель-переменную с именем point на функции без параметров, возвращающие значения типа **int**.

Важнейшим элементом в определении указателя на функции являются круглые скобки. Если записать

```
int * funct (void);
```

то это будет не определением указателя, а прототипом функции без параметров с именем funct, возвращающей значения типа **int***.

В отличие от имени функции (например, funct), указатель point является переменной, то есть ему можно присваивать значения других указателей, определяющих адреса функций программы. Принципиальное требование – тип указателя-переменной должен полностью соответствовать типу функции, адрес которой ему присваивается.

Мы уже говорили, что тип функции определяется типом возвращаемого значения и спецификацией ее параметров. Таким образом, попытка выполнить следующее присваивание

```
point=funct; /* Ошибка - несоответствие типов */
```

будет ошибочной, так как типы возвращаемых значений для point и funct различны.

Неконстантный указатель на функцию, настроенный на адрес конкретной функции, может быть использован для вызова этой функции. Таким образом, при обращении к функции перед круглыми скобками со списком аргументов можно помещать: *имя_функции* (то есть константный указатель); *указатель-переменную* того же типа, значение которого равно адресу функции; *выражение разыменования* такого же указателя с таким же значением. Следующая программа иллюстрирует три способа вызова функций.

```
#include <stdio.h>
void f1 (void)
{
   printf ("\n Выполняется f1( )");
}
void f2 (void)
{
   printf ("\n Выполняется f2( )");
}
void main ()
{
```

```
    void (*point) (void);
    /*point - указатель-переменная на функцию */
    f2 (); /* Явный вызов функции f2()*/
    point=f2; /* Настройка указателя на f2()*/
    (*point)(); /* Вызов f2() по ее адресу
                   с разыменованием указателя */
    point=f1; /* Настройка указателя на f1()*/
    (*point)(); /* Вызов f1() по ее адресу
                   с разыменованием указателя */
    point (); /* Вызов f1() без явного разыменования указателя */
}
```

Результат выполнения программы:

```
Выполняется f2( )
Выполняется f2( )
Выполняется f1( )
Выполняется f1( )
```

Все варианты обращения к функциям с использованием указателей-констант (имен функций) и указателей-переменных (неконстантных указателей на функции) в программе продемонстрированы и снабжены комментариями. Приведем общий вид двух операций вызова функций с помощью неконстантных указателей:

*(*имя_указателя) (список_аргументов)*
имя_указателя (список_аргументов)

В обоих случаях тип указателя должен соответствовать типу вызываемой функции, и между аргументами и параметрами должно быть соответствие. При использовании явного разыменования обязательны круглые скобки, то есть в нашей программе будет ошибочной запись

```
*point ();   /* Ошибочный вызов */
```

Это полностью соответствует синтаксису языка. Операция '( )' – «круглые скобки» имеет более высокий приоритет, чем операция разыменования '*'. В этом ошибочном вызове вначале выполнится вызов point( ), а уж к результату будет применена операция разыменования.

При определении указателя на функции он может быть инициализирован. В качестве инициализирующего выражения должен ис-

пользоваться адрес функции того же типа, что и тип определяемого указателя, например:

```
int fic (char); /* Прототип функции */
int (*pfic) (char)=fic;
/* pfic - указатель на функцию */
```

Массивы указателей на функции. Такие массивы по смыслу ничем не отличаются от массивов других объектов, однако форма их определения несколько необычна:

тип **(****имя_массива* [*размер*] **)** **(***спецификация_параметров***)***;*

где *тип* определяет тип возвращаемых функциями значений; *имя_массива* – произвольный идентификатор; *размер* – количество элементов в массиве; *спецификация_параметров* – определяет состав и типы параметров функций.

Пример:

```
int (*parray [4]) (char);
```

здесь parray – массив указателей на функции, каждому из которых можно присвоить адрес определенной выше функции **int** fic **(char)** и адрес любой функции с прототипом вида

int *имя-функции* **(char)**;

Массив в соответствии с синтаксисом языка является производным типом наряду с указателями и функциями. Массив функций создать нельзя, однако, как мы показали, можно определить массив указателей на функции. Тем самым появляется возможность создавать «таблицы переходов» (jump tables), или «таблицы передачи управления». С помощью таблицы переходов удобно организовывать ветвления с возвратом по результатам анализа некоторых условий. Для этого все ветви обработки (например, N+1 штук) оформляются в виде однотипных функций (с одинаковым типом возвращаемого значения и одинаковой спецификацией параметров). Определяется массив указателей из N+1 элементов, каждому элементу которого присваивается адрес конкретной функции обработки. Затем формируются условия, на основе которых должна выбираться та или иная функция (ветвь) обработки. Вводится индекс, значение которого должно находиться в пределах от 0 до N включительно, где (N+1) – количество ветвей обработки. Каждо-

му условию ставится в соответствие конкретное значение индекса. По конкретному значению индекса выполняются обращение к элементу массива указателей на функции и вызов соответствующей функции обработки:

имя_массива [*индекс*] (*список_аргументов*);
(**имя массива* [*индекс*]) (*список_аргументов*);

Описанную схему использования массивов указателей на функции удобно применять для организации меню, точнее программ, которыми управляет пользователь с помощью меню.

```
#include <stdlib.h> /* Для функции exit ( ) */
#include <stdio.h>
#define N 2
void act0(char * name)
{
   printf("%s: Работа завершена!\n",name);
   exit(0);
}
void act1(char * name)
{
   printf("%s: работа 1\n",name);
}
void act2(char * name)
{
   printf("%s: работа 2\n",name);
}
void main()
{ /* Массив указателей на функции: */
void (* pact[ ])(char *)={act0,act1,act2};
char string[12];
int number;
printf("\n\nВводите имя: ");
scanf("%s",string);
printf("Вводите номера работ от 0 до %d:\n",N);
while(1)
   {
      scanf("%d",&number);
      /* Ветвление по условию   */
      pact[number](string);
   }
}
```

Пример выполнения программы:

```
Вводите имя: Peter  <Enter>
Вводите номера работы от 0 до 2:
1  <Enter>
Peter: работа 1
1  <Enter>
Peter: работа 1
2  <Enter>
Peter: работа 2
0  <Enter>
Peter: Работа завершена!
```

В программе для упрощения нет защиты от неверно введенных данных, то есть возможен выход индекса за пределы, определенные для массива pact[ ] указателей на функции. При такой ситуации результат непредсказуем.

Указатели на функции как параметры позволяют создавать функции, реализующие тот или иной метод обработки другой функции, которая заранее не определена. Например, можно определить функцию для вычисления определенного интеграла. Подынтегральная функция может быть передана в функцию вычисления интеграла с помощью параметра-указателя. Пример функции для вычисления определенного интеграла с помощью формулы прямоугольников:

```
double rectangle
      (double (* pf)(double), double a, double b)
{
   int N=20;
   int i;
   double h,s=0.0;
   h=(b-a)/N;
   for (i=0; i<N; i++)
      s+=pf(a+h/2+i*h);
   return h*s;
}
```

Параметры функции rectangle( ): pf – указатель на функцию с параметром типа **double**, возвращающую значение типа **double**. Это указатель на функцию, вычисляющую значение подынтегральной функции при заданном значении аргумента. Параметры a, b – пределы интегрирования. Число интервалов разбиения отрезка интегри-

рования фиксировано: N=20. Пусть текст функции под именем rect.c сохранен в каталоге пользователя.

Предположим, что функция rectangle( ) должна быть использована для вычисления приближенных значений интегралов (Абрамов С. А., Зима Е. В. Начала информатики. – М.: Наука, 1989. – С. 83):

$$\int_{-1}^{2} \frac{x}{(x^2+1)^2}\,dx \text{ и } \int_{0}^{1/2} 4\cos^2 x\,dx.$$

Программа для решения этой задачи может иметь следующий вид:

```
#include <math.h>
#include <stdio.h>
#include "rect.c" /* Включение определения функции  rectangle( ) */
double ratio(double x) /* Подынтегральная функция */
{
   double z; /* Вспомогательная переменная */
   z=x*x+1;
   return x/(z*z);
}
double cos4_2(double v)  /* Подынтегральная функция */
{
   double w;  /* Вспомогательная переменная */
   w=cos(v);
   return 4*w*w;
}
void main()
{
   double a,b,c;
   a=-1.0;
   b=2.0;
   c=rectangle(ratio,a,b);
   printf("\n Первый интеграл: %f",c);
   printf("\n Второй интеграл: %f",
          rectangle(cos4_2,0.0,0.5));
}
```

Результат выполнения программы:

```
Первый интеграл: 0,149847
Второй интеграл: 1.841559
```

Комментарии к тексту программы могут быть следующими. Директива **#include** "rect.c" включает на этапе препроцессорной обработки в программу определение функции rectangle( ). Предполагается, как упомянуто выше, что текст этого определения находится в файле rect.c. В языке Си, как говорилось в главе 3, существует соглашение об обозначении имен включаемых файлов. Если имя файла заключено в угловые скобки '< >', то считается, что это один из файлов стандартной библиотеки компилятора. Например, файл <**math.h**> содержит средства связи с библиотечными математическими функциями. Файл, название которого помещено в кавычках " ", воспринимается как файл из текущего каталога. Именно там в этом примере препроцессор начинает разыскивать файл rect.c.

Определения функций ratio( ) и cos4_2( ), позволяющих вычислять значения подынтегральных выражений по заданному значению аргумента, ничем не примечательны. Их прототипы соответствуют требованиям спецификации первого параметра функции rectangle( ):

double *имя* (**double**)

В основной программе **main**( ) функция rectangle( ) вызывается дважды с разными значениями аргументов. Для разнообразия вызовы выполнены по-разному, но каждый раз первый аргумент – это имя конкретной функции, то есть константный указатель на функцию.

Указатель на функцию как возвращаемое функцией значение. При организации меню в тех случаях, когда количество вариантов и соответствующее количество действий определяются не в точке исполнения, а в «промежуточной» функции, удобно возвращать из этой промежуточной функции адрес той конкретной функции, которая должна быть выполнена. Этот адрес можно возвращать в виде значения указателя на функцию.

Рассмотрим программу, демонстрирующую особенности такой организации меню.

В программе определены три функции: первые две функции f1( ) и f2( ) с прототипом вида

int f (**void**);

(пустой список параметров, возвращается значение типа **int**) и третья функция menu( ) с прототипом

int (*menu(**void**)) (**void**);

которая возвращает значение указателя на функции с пустыми списками параметров, возвращающие значения типа **int**.

При выполнении функции menu( ) пользователю дается возможность выбора из двух пунктов меню. Пунктам меню соответствуют определенные выше функции f1( ) и f2( ), указатель на одну из которых является возвращаемым значением. При неверном выборе номера пункта возвращаемое значение становится равным **NULL**.

В основной программе определен указатель r, который может принимать значения адресов функций f1( ) и f2( ). В бесконечном цикле выполняются обращения к функции menu( ), и если результат равен **NULL**, то программа печатает «The End» и завершает выполнение. В противном случае вызов

```
t=(*r) ( );
```

обеспечивает исполнение той из функций f1( ) или f2( ), адрес которой является значением указателя r.

Текст программы[1]:

```
#include <stdio.h>
int f1(void)
{
   printf(" The first actions: ");
   return 1;
}
int f2(void)
{
   printf(" The second actions: ");
   return 2;
}
int (* menu(void))(void)
{
   int choice; /* Номер пункта меню */
   /* Массив указателей на функции: */
   int (* menu_items[])() = {f1, f2};
   printf("\n Pick the menu item  (1 or 2): ");
   scanf("%d",&choice);
   if (choice < 3 && choice > 0)
       return menu_items[choice - 1];
   else
       return NULL;
   }
```

[1] Исходный вариант программы предложен С. М. Лавреновым.

```
void main()
{
   int (*r)(void);   /* Указатель на функции */
   int t;
   while (1)
      {  /* Обращение к меню:   */
       r=menu();
       if (r == NULL)
        {
         printf("\nThe End!");
         return;
       }
       /* Вызов выбранной функции */
       t=(*r)();
       printf("\tt= %d",t);
      }
}
```

Результаты выполнения программы:

```
Pick the menu item (1 or 2): 2  <Enter>
The second actions: t=2
Pick the menu item (1 or 2): 1  <Enter>
The first actions: t=1
Pick the menu item (1 or 2): 24 <Enter>
The End!
```

В функции menu( ) определен массив menu_items[ ] указателей на функции. В качестве инициализирующих значений в списке использованы имена функций f1( ) и f2( ):

int (* menu_items[ ]) ( ) = {f1, f2};

Такое определение массива указателей на функции по меньшей мере не очень наглядно. Упростить подобные определения можно с помощью вспомогательных обозначений (имен), вводимых спецификатором **typedef**. Например, то же самое определение массива указателей можно ввести так:

typedef int (*Menu_action) (**void**);
Menu_action menu_items [ ] = {f1, f2};

Здесь **typedef** вводит обозначение Menu_action для типа «указатель на функции с пустым списком параметров, возвращающие значения типа **int**».

Библиотечные функции с указателями на функции в параметрах. Эти функции играют важную роль при решении задач на языке Си. В стандартную библиотеку компилятора с языка Си включены, по крайней мере, следующие функции:

- **qsort ( )** – функция быстрой сортировки массива;
- **search ( )** – функция поиска в упорядоченном массиве элемента с заданными свойствами.

У каждой из этих функций один из параметров – указатель на функцию. Для функции быстрой сортировки нужен указатель на функцию, позволяющую задать правила упорядочения (сравнения) элементов. В функции поиска параметр – указатель на функцию позволяет задать функцию, с помощью которой программист должен сформулировать требования к искомому элементу массива.

Опыт работы на языке Си показал, что даже не новичок в области программирования испытывает серьезные неудобства, разбирая синтаксис определения конструкций, включающих указатели на функции. Например, не каждому сразу становится понятным такое определение прототипа библиотечной функции:

```
void qsort(void * base, size_t nelem, size_t width,
        int (*fcmp)(const void * p1, const void * p2));
```

Это прототип функции быстрой сортировки, входящей в стандартную библиотеку функций. Прототип находится в заголовочном файле **stdlib.h**. Функция **qsort( )** сортирует содержимое таблицы (массива) однотипных элементов, неоднократно вызывая функцию сравнения, подготовленную пользователем. Для вызова функции сравнения ее адрес должен заместить указатель fcmp, специфицированный как параметр. При использовании **qsort( )** программист должен подготовить таблицу сортируемых элементов в виде одномерного массива фиксированной длины и написать функцию, позволяющую сравнивать два любых элемента сортируемой таблицы. Остановимся на параметрах функции **qsort( )**:

- **base** – указатель на начало таблицы (массива) сортируемых элементов (адрес нулевого элемента массива);
- **nelem** – количество сортируемых элементов в таблице (целая величина, не большая размера массива) – сортируются первые **nelem** элементов от начала массива;
- **width** – размер элемента таблицы (целая величина, определяющая в байтах размер одного элемента массива);

- **fcmp** – указатель на функцию сравнения, получающую в качестве параметров два указателя **p1**, **p2** на элементы таблицы и возвращающую в зависимости от результата сравнения целое число:
  - если ***p1 < *p2**, функция **fcmp ( )** возвращает отрицательное целое < 0;
  - если ***p1 == *p2**, **fcmp ( )** возвращает 0;
  - если ***p1 > *p2**, **fcmp ( )** возвращает положительное целое > 0.

При сравнении символ «меньше, чем» **(<)** означает, что после сортировки левый элемент отношения ***p1** должен оказаться в таблице перед правым элементом ***p2**, то есть значение ***p1** должно иметь меньшее значение индекса в массиве, чем ***p2**. Аналогично (но обратно) определяется расположение элементов при выполнении соотношения «больше, чем» **(>)**.

В следующей программе функция **qsort( )** используется для упорядочения массива указателей на строки разной длины. Упорядочение должно быть выполнено таким образом, чтобы последовательный перебор массива указателей позволял получать строки в алфавитном порядке. Сами строки в процессе сортировки не меняют своих положений. Изменяются только значения указателей в массиве.

```
#include <stdio.h>
#include <stdlib.h>   /* Для функции qsort() */
#include <string.h>   /* Для сравнения строк: strcmp()*/
/* Определение функции для сравнения: */
int compare(const void *a, const void *b)
{
   unsigned long *pa = (unsigned long *)a,
                 *pb = (unsigned long *)b;
   return strcmp((char *)*pa, (char *)*pb);
}
void main()
{
   char *pc[] = {
         "One - 1",
         "Two - 2",
         "Three - 3",
         "Four - 4",
         "Five - 5",
```

```
      "Six - 6",
      "Seven - 7",
      "Eight - 8" };
  /* Размер таблицы:  */
  int n = sizeof(pc)/sizeof(pc[0]);
  int i;
  printf("\n   До сортировки:");
  for (i = 0; i < n; i++)
     printf("\npc [%d] = %p -> %s",
           i,pc[i],pc[i]);
  /* Вызов функции упорядочения: */
  qsort((void *)
     pc,/* Адрес начала сортируемой таблицы */
     n,/* Число элементов сортируемой таблицы */
     sizeof(pc[0]), /* Размер одного элемента */
     compare /* Имя функции сравнения (указатель) */
        );
  printf("\n\n   После сортировки:");
  for (i = 0; i < n; i++)
     printf("\npc [%d] = %p -> %s",
           i,pc[i],pc[i]);
}
```

Результаты выполнения программы:

- до сортировки:

```
pc [0] = 00B8 -> One - 1
pc [1] = 00C0 -> Two - 2
pc [2] = 00C8 -> Three - 3
pc [3] = 00D2 -> Four - 4
pc [4] = 00DC -> Five - 5
pc [5] = 00E5 -> Six - 6
pc [6] = 00ED -> Seven - 7
pc [7] = 00F7 -> Eight - 8
```

- после сортировки:

```
pc [0] = 00F7 -> Eight - 8
pc [1] = 00DC -> Five - 5
pc [2] = 00D2 -> Four - 4
pc [3] = 00B8 -> One - 1
pc [4] = 00ED -> Seven - 7
pc [5] = 00E5 -> Six - 6
pc [6] = 00C8 -> Three - 3
pc [7] = 00C0 -> Two - 2
```

Вывод адресов, то есть значений указателей pc[i], выполняется в шестнадцатеричном виде с помощью спецификации преобразования **%p**.

Обратите внимание на значения указателей pc[i]. До сортировки разность между pc[1] и pc[0] равна длине строки «**One – 1**» и т. д. После упорядочения pc[0] получит значение, равное исходному значению pc[7], и т. д.

Для выполнения сравнения строк (а не элементов массива **pc[ ]**) в функции **compare( )** использована библиотечная функция **strcmp( )**, прототип которой в заголовочном файле **string.h** имеет вид:

```
int strcmp(const char *s1, const char *s2);
```

Функция **strcmp( )** сравнивает строки, связанные с указателями s1 и s2. Сравнение выполняется посимвольно, начиная с начальных символов строк и до тех пор, пока не встретятся несовпадающие символы либо не закончится одна из строк.

Прототип функции **strcmp( )** требует, чтобы параметры имели тип **(const char *)**. Входные параметры функции **compare( )** имеют тип **(const void *)**, как предусматривает определение функции **qsort( )**. Необходимые преобразования для наглядности выполнены в два этапа. В теле функции **compare( )** определены два вспомогательных указателя типа **(unsigned long *)**, которым присваиваются значения адресов элементов сортируемой таблицы (элементов массива pc**[ ]**) указателей. В свою очередь, функция **strcmp( )** получает разыменования этих указателей, то есть адреса символьных строк. Таким образом, выполняется сравнение не элементов массива **char *** pc**[ ]**, а тех строк, адреса которых являются значениями pc[i]. Однако функция **qsort( )** работает с массивом pc**[ ]** и меняет местами только значения его элементов. Последовательный перебор массива pc**[ ]** позволяет в дальнейшем получить строки в алфавитном порядке, что иллюстрирует результат выполнения программы. Так как pc[i] – указатель на некоторую строку, то по спецификации преобразования **%s** выводится сама строка.

Если не использовать вспомогательных указателей pa, pb, то функцию сравнения строк можно вызвать из тела функции **compare( )** таким оператором:

```
return strcmp((char *)(*(unsigned long *)a),
          (char *)(*(unsigned long *)b));
```

где каждый указатель **(void *)** вначале преобразуется к типу **(unsigned long *)**. Последующее разыменование «достает» из не-

скольких смежных байтов значение соответствующего указателя pc[i], затем преобразование **(char *)** формирует такой указатель на строку, который нужен функции **strcmp( )**.

5.5. Функции с переменным количеством аргументов

В языке Си допустимы функции, количество аргументов у которых при компиляции функции не фиксировано. Кроме того, могут быть неизвестными и типы аргументов. Количество и типы аргументов становятся известными только в момент вызова функции, когда явно задан их список. При определении и описании таких функций спецификация параметров заканчивается запятой и многоточием. Формат прототипа функции с переменным количеством аргументов:

тип имя (спецификация_явных_параметров, ...);

где *тип* – тип возвращаемого функцией значения; *имя* – имя функции; *спецификация_явных_параметров* – список спецификаций параметров, количество и типы которых фиксированы и известны в момент компиляции. Эти параметры можно назвать обязательными.

Каждая функция с переменным количеством аргументов должна иметь хотя бы один обязательный параметр. После списка явных (обязательных) параметров ставится запятая, а затем многоточие, извещающее компилятор, что дальнейший контроль соответствия количества и типов аргументов при обработке вызова функции проводить не нужно. Сложность в том, что у переменного списка аргументов нет даже имени, поэтому не понятно, как найти его начало и конец.

Каждая функция с переменным списком аргументов должна иметь механизм определения их количества и их типов. Принципиально различных подходов к созданию этого механизма два. Первый подход предполагает добавление в конец списка реально использованных (необязательных) аргументов специального аргумента-индикатора с уникальным значением, которое будет сигнализировать об окончании списка. При таком подходе в теле функции параметры последовательно перебираются, и их значения сравниваются с заранее известным концевым признаком. Второй подход предусматривает передачу в функцию сведений о реальном количестве аргументов. Эти сведения о реальном количестве используемых аргументов можно передавать в функцию с помощью одного из явно

задаваемых (обязательных) параметров. В обоих подходах – и при задании концевого признака, и при указании числа реально используемых аргументов – переход от одного аргумента к другому выполняется с помощью указателей, то есть с использованием адресной арифметики. Проиллюстрируем сказанное примерами.

Доступ к адресам параметров из списка. Следующая программа включает функцию с изменяемым списком параметров, первый из которых (единственный обязательный) определяет число действительно используемых при вызове необязательных аргументов.

```
#include <stdio.h>
/*Функция суммирует значения своих аргументов типа int */
long summa(int k,...)
    /* k - число суммируемых аргументов   */
{
   int *pick = &k; /* Настроили указатель на параметр k*/
   long total = 0;
   for(; k; k--)
       total += *(++pick);
   return total;
}
void main()
{
   printf("\n summa(2, 6, 4) = %d",summa(2,6,4));
   printf("\n summa(6, 1, 2, 3, 4, 5, 6) = %d",
          summa(6,1,2,3,4,5,6));
}
```

Результат выполнения программы:

```
summa(2, 6, 4) = 10
summa(6, 1, 2, 3, 4, 5, 6) = 21
```

Для доступа к списку аргументов используется указатель pick типа **int ***. Вначале ему присваивается адрес явно заданного параметра k, то есть он устанавливается на начало списка аргументов в памяти. Затем в цикле указатель pick перемещается по адресам следующих аргументов, соответствующих неявным параметрам. С помощью разыменования *(++pick) выполняется выборка их значений. Параметром цикла суммирования служит значение k, которое уменьшается на 1 после каждой итерации и, наконец, становится нулевым. Особенность функции – возможность работы только с целочисленными

аргументами, так как указатель pick после обработки значения очередного параметра «перемещается вперед» на величину **sizeof(int)** и должен быть всегда установлен на начало следующего аргумента.

Недостаток функции – возможность ошибочного задания неверного количества реально используемых аргументов.

Следующий пример содержит функцию для вычисления произведения переменного количества аргументов. Признаком окончания списка аргументов служит аргумент с нулевым значением.

```
#include <stdio.h>
/* Функция вычисляет произведение аргументов:*/
double prod(double arg, ...)
{
   double aa = 1.0; /* Формируемое произведение*/
   double *prt = &arg; /* Настроили указатель на параметр arg */
   if (*prt == 0.0)
       return 0.0;
   for (; *prt; prt++)
       aa *= *prt;
   return aa;
}
void main()
{
   double prod(double,...);/* Прототип функции */
   printf("\n prod(2e0, 4e0, 3e0, 0e0) = %e",
                            prod(2e0,4e0,3e0,0e0));
   printf("\n prod(1.5, 2.0, 3.0, 0.0) = %f",
                            prod(1.5,2.0,3.0,0.0));
   printf("\n prod(1.4, 3.0, 0.0, 16.0, 84.3, 0.0)=%f",
                            prod(1.4,3.0,0.0,16.0,84.3,0.0));
   printf( "\n prod(0e0) = %e",prod(0e0));
}
```

Результат выполнения программы:

```
prod(2e0, 4e0, 3e0, 0e0) = 2.400000e+01
prod(1.5, 2.0, 3.0, 0.0) = 9.000000
prod(1.4, 3.0, 0.0, 16.0, 84.3, 0.0) = 4.200000
prod(0e0) = 0.000000e+00
```

В функции prod( ) перемещение указателя prt по списку аргументов выполняется всегда за счет изменения prt на величину **sizeof(double)**. Поэтому все параметры при обращении к функции

prod( ) должны иметь тип **double**. В вызовах функции проиллюстрированы некоторые варианты задания аргументов. Обратите внимание на вариант с нулевым значением аргумента в середине списка. Аргументы вслед за этим значением игнорируются. Недостаток функции – неопределенность действий при отсутствии в списке аргумента с нулевым значением.

Чтобы функция с переменным количеством аргументов могла воспринимать аргументы различных типов, необходимо в качестве исходных данных каким-то образом передавать ей информацию о типах параметров. Для однотипных параметров возможно, например, такое решение – передавать с помощью дополнительного обязательного параметра признак типа аргумента. Определим функцию, выбирающую минимальное из значений аргументов, которые могут быть двух типов: или только **long**, или только **int**. Признак типа аргумента будем передавать как значение первого обязательного параметра. Второй обязательный аргумент указывает количество параметров, из значений которых выбирается минимальное. В следующей программе предложен один из вариантов решения сформулированной задачи:

```
#include <stdio.h>
void main()
{         /* Прототип функции: */
   long minimum(char, int , ...);
   printf("\n\tminimum('l',3,10L,20L,30L) = %ld",
      minimum('l',3,10L,20L,30L));
   printf("\n\tminimum('i',4,11, 2, 3, 4) = %ld",
      minimum('i',4,11,2,3,4));
   printf( "\n\tminimum('k', 2, 0, 64) = %ld",
      minimum('k',2,0,64));
}
/* Определение функции с переменным списком параметров */
long minimum(char z, int k,...)
{
   if (z == 'i')
      {
       int *pi = &k + 1; /* Настроились на первый
                            необязательный параметр */
       int min = *pi; /* Значение первого
                         необязательного параметра */
       for(; k; k--, pi++)
           min = min > *pi ? *pi : min;
```

```
      return (long)min;
     }
  if (z == 'l')
     {
       long *pl = (long*)(&k+1);
       long min = *pl; /* Значение первого
                          необязательного параметра */
        for(; k; k--, pl++)
          min = min > *pl ? *pl : min;
        return (long)min;
     }
  printf("\nОшибка! Неверно задан 1-й параметр:");
  return 2222L;
}
```

Результат выполнения программы:

```
      minimum('l', 3, 10L, 20L, 30L) = 10
      minimum('i', 4, 11, 2, 3, 4) = 2
Ошибка! Неверно задан 1-й параметр:
      minimum('k',2,0,64)=2222
```

В приведенных примерах функций с изменяемыми списками аргументов перебор параметров выполнялся с использованием адресной арифметики и явным применением указателей нужных типов. К проиллюстрированному способу перехода от одного параметра к другому нужно относиться с осторожностью. Дело в том, что порядок размещения параметров в памяти ЭВМ зависит от реализации компилятора. В компиляторах имеются опции, позволяющие изменять последовательность размещения параметров. Стандартная для языка Си на современном ПК с процессором фирмы Intel последовательность: меньшее значение адреса у первого параметра функции, а остальные размещены подряд в соответствии с увеличением адресов. Противоположный порядок обработки и размещения будет у функций, определенных и описанных с модификатором **pascal**. Этот модификатор и его антипод – модификатор **cdecl** являются дополнительными ключевыми словами, определенными для ряда компиляторов. Не останавливаясь подробно на возможностях, предоставляемых модификатором **pascal**, отметим два факта. Во-первых, применение модификатора **pascal** необходимо в тех случаях, когда функция, написанная на языке Си, будет вызываться из программы, подготовленной на Паскале. Во-вторых, функция с модификатором

pascal не может иметь переменного списка аргументов, то есть в ее определении и в ее прототипе нельзя использовать многоточие.

Макросредства для переменного числа аргументов. Вернемся к особенностям конструирования функций со списками параметров переменной длины и различных типов. Предложенный выше способ передвижения по списку параметров имеет один существенный недостаток – он ориентирован на конкретный тип машин и привязан к реализации компилятора. Поэтому функции могут оказаться немобильными.

Для обеспечения мобильности программ с функциями, имеющими изменяемые списки аргументов, в каждый компилятор стандарт языка предлагает включать специальный набор макроопределений, которые становятся доступными при включении в текст программы заголовочного файла **stdarg.h**. Макрокоманды, обеспечивающие простой и стандартный (не зависящий от реализации) способ доступа к конкретным спискам аргументов переменной длины, имеют следующий формат:

void va_start(va_list *param*, *последний_явный_параметр*);
type **va_arg(va_list** *param*, *type*);
void va_end(va_list *param*);

Кроме перечисленных макросов, в файле **stdarg.h** определен специальный тип данных **va_list**, соответствующий потребностям обработки переменных списков параметров. Именно такого типа должны быть первые аргументы, используемые при обращении к макрокомандам **va_start( )**, **va_arg( )**, **va_end( )**. Кроме того, для обращения к макросу **va_arg( )** необходимо в качестве второго аргумента использовать обозначение типа (*type*) очередного аргумента, к которому выполняется доступ. Объясним порядок использования перечисленных макроопределений в теле функции с переменным количеством аргументов (рис. 5.2). Напомним, что каждая такая функция с переменным количеством аргументов должна иметь хотя бы один явно специфицированный параметр, за которым после запятой стоит многоточие. В теле функции обязательно определяется объект типа **va_list**. Например, так:

va_list factor;

Определенный таким образом объект factor обладает свойствами указателя. С помощью макроса **va_start( )** объект factor связывается с первым необязательным параметром, то есть с началом списка неизвестной длины. Для этого в качестве второго аргумента при

va_arg (factor, *type*)

Реальные аргументы						
обязательные			необязательные			
int	**int**	**float**	**int**	**char**	**int[ ]**	**long double**
N	K	sum	Z	cc	ro	smell

va_list factor; /* Определили указатель */
va_start (factor, sum); /* Настройка указателя (см. ниже) */

N	K	sum	Z	cc	ro	smell

int ix;
ix=**va_arg** (factor, **int**); /* Считали в ix значение Z типа **int**,
одновременно сместили указатель на длину переменной типа **int** */

N	K	sum	Z	cc	ro	smell

char cx;
cx=**va_arg** (factor, **char**); /* Считали в cx значение cc типа **char** и
сместили указатель на длину объекта типа **char** */

N	K	sum	Z	cc	ro	smell

int * pr;
pr=**va_arg**(factor, **int**); /* Считали в pr адрес ro типа **int*** массива типа
int [ ] и сместили указатель factor на размер указателя **int*** */

long double pp;
pp=**va_arg** (factor, **long double**); /* Считали значение smell */

Рис. 5.2. *Схема обработки переменного списка параметров с помощью макросов стандартной библиотеки*

обращении к макросу **va_start( )** используется последний из явно специфицированных параметров функции (предшествующий многоточию):

va_start (factor, *последний_явный_параметр*);

Рассмотрев выше способы перемещения по списку параметров с помощью адресной арифметики, мы понимаем, что указатель factor сначала «нацеливается» на адрес последнего явно специфицированного параметра, а затем перемещается на его длину и тем самым устанавливается на начало переменного списка аргументов. Именно поэтому функция с переменным списком аргументов должна иметь хотя бы один явно специфицированный параметр.

Теперь с помощью разыменования указателя factor мы можем получить значение первого аргумента из переменного списка. Однако нам неизвестен тип этого аргумента. Как и без использования макросов, тип аргументов нужно каким-то образом передать в функцию. Если это сделано, то есть определен тип *type* очередного аргумента, то обращение к макросу позволяет, во-первых, получить значение очередного (сначала первого) аргумента типа *type*. Вторая задача макрокоманды **va_arg( )** – заменить значение указателя factor на адрес следующего аргумента в списке. Теперь, узнав каким-то образом тип, например *type1*, этого следующего аргумента, можно вновь обратиться к макросу:

va_arg (factor, *type1*)

Это обращение позволяет получить значение следующего аргумента и переадресовать указатель factor на аргумент, стоящий за ним в списке, и т. д.

Макрокоманда **va_end( )** предназначена для организации корректного возврата из функции с переменным списком аргументов. Ее единственным аргументом должен быть указатель типа **va_list**, который использовался в функции для перебора параметров. Таким образом, для наших иллюстраций вызов макрокоманды должен иметь вид:

va_end (factor);

Макрокоманда **va_end( )** должна быть вызвана после того, как функция обработает весь список аргументов.

Макрокоманда **va_end( )** обычно модифицирует свой аргумент (указатель типа **va_list**), и поэтому его нельзя будет повторно использовать без предварительного вызова макроса **va_start( )**.

Примеры функций с переменным количеством аргументов. Для иллюстрации особенностей использования описанных макросов рассмотрим функцию, формирующую в динамической памяти массив из элементов типа **double**. Количество элементов определяется значением первого обязательного параметра функции, имеющего тип **int**. Значения элементов массива передаются в функцию с помощью переменного числа необязательных аргументов типа **double**. Текст функции вместе с основной программой, из которой выполняется ее вызов:

```
#include <stdarg.h> /* Для макросредств */
#include <stdlib.h> /* Для функции calloc( ) */
```

```
double * set_array (int k, ...)
{
   int i;
   va_list par; /* Вспомогательный указатель */
   double * rez; /* Указатель на результат */
   rez=calloc(k,sizeof(double));
   if (rez == NULL)
       return NULL;
   va_start (par, k); /* "Настройка" указателя */
   /* Цикл "чтения" параметров: */
   for (i=0; i<k; i++)
       rez[i]=va_arg (par, double);
   va_end (par);
   return rez;
}
#include <stdio.h>
void main( )
{
   double * array; int j;
   int n=5;
   printf("\n");
   array=set_array (n, 1.0, 2.0, 3.0, 4.0, 5.0);
   if (array == NULL) return;
   for (j=0; j<n; j++)
       printf ("\t%f", array [j]);
   free(array);
}
```

Результат выполнения программы:

```
1.000000    2.000000    3.000000    4.000000    5.000000
```

В теле функции обратите внимание на применение функции **calloc( )**, позволяющей выделить память для массива. Первый параметр функции **calloc( )** – количество элементов массива, второй – размер в байтах одного элемента. При неудачном выделении памяти функция **calloc( )** возвращает нулевое значение адреса, что проверяет следующий ниже условный оператор.

В функциях с переменным количеством аргументов есть один уязвимый пункт – если при «чтении» аргументов с помощью макроса **va_arg( )** указатель выйдет за пределы явно использованного списка аргументов, то результат, возвращаемый макросом **va_arg( )**, не определен. Таким образом, в нашей функции set_array( ) коли-

чество явно заданных аргументов переменного списка ни в коем случае не должно быть меньше значения первого аргумента (заменяющего **int k**).

В основной программе **main( )** определен указатель **double** * array, которому присваивается результат (адрес динамического массива), возвращаемый функцией set_array( ). Затем с помощью указателя array в цикле выполняется доступ к элементам массива, и их значения выводятся на экран дисплея.

В следующей программе в качестве еще одной иллюстрации механизма обработки переменного списка аргументов используется функция для конкатенации любого количества символьных строк. Строки, предназначенные для соединения в одну строку, передаются в функцию с помощью списка указателей-аргументов. В конце списка неопределенной длины всегда помещается нулевой указатель **NULL**.

```
#include <stdio.h>
#include <string.h> /* Для функций обработки строк */
#include <stdarg.h> /* Для макросредств          */
#include <stdlib.h> /* Для функции malloc( ) */
char *concat(char *s1, ...)
{
   va_list par;/* Указатель на аргументы списка */
   char *cp = s1;
   char *string;
   int len = strlen(s1);/* Длина 1-го аргумента */
   va_start(par, s1);    /* Начало переменного списка */
   /* Цикл вычисления общей длины строк: */
   while (cp = va_arg(par, char *))
        len += strlen(cp);
   /* Выделение памяти для результата: */
   string = (char *)malloc(len + 1);
   strcpy(string, s1); /* Копируем 1-й параметр */
   va_start(par, s1);   /* Начало переменного списка */
   /* Цикл конкатенации строк: */
   while (cp = va_arg(par, char *))
        strcat(string, cp); /* Конкатенация двух строк */
   va_end(par);
   return string;
}
void main()
{
   char* concat(char* s1, ...); /* Прототип функции */
```

```
   char* s;           /* Указатель для результата */
   s=concat("\nNulla ","Dies ","Sine ", "Linea!", NULL);
   s = concat(s,
           " - Ни одного дня без черточки!",
           "\n\t",
           "(Плиний Старший о художнике Апеллесе)", NULL);
   printf("\n%s",s);
}
```

Результат выполнения программы:

```
Nulla Dies Sine Linea! - Ни одного дня без черточки!
     (Плиний Старший о художнике Апеллесе)
```

В приведенной функции concat( ) количество аргументов переменно, но их тип заранее известен и фиксирован. В ряде случаев полезно иметь функцию, аргументы которой изменяются как по числу, так и по типам. В этом случае, как уже говорилось, нужно сообщать функции о типе очередного аргумента. Поучительным примером таких функций служат библиотечные функции форматного ввода-вывода языка Си:

printf(char* format, ...);
scanf(char* format, ...);

В обеих функциях форматная строка, связанная с указателем format, содержит спецификации преобразования (**%d** – для десятичных чисел, **%e** – для вещественных данных в форме с плавающей точкой, **%f** – для вещественных значений в форме с фиксированной точкой и т. д.). Кроме того, эта форматная строка в функции **printf( )** может содержать произвольные символы, которые выводятся на дисплей без какого-либо преобразования. Чтобы продемонстрировать особенности построения функций с переменным числом аргументов, классики языка Си [1] рекомендуют самостоятельно написать функцию, подобную функции **printf( )**. Последуем их совету, но для простоты разрешим использовать только спецификации преобразования «**%d**» и «**%f**».

```
/* Упрощенный аналог printf( ).
   По мотивам K&R, [2], стр. 152 */
#include <stdio.h>
#include <stdarg.h> /* Для макросредств
                    переменного списка параметров */
```

```
void miniprint(char *format, ...)
{
   va_list ap;/* Указатель на необязательный параметр*/
   char *p;  /* Для просмотра строки format */
   int ii;   /* Целые параметры                  */
   double dd;     /* Параметры типа double */
   va_start(ap,format);/*Настроились на первый параметр */
   for (p = format; *p; p++)
       {
          if (*p != '%')
             {
                printf("%c",*p);
                continue;
             }
          switch (*++p)
             { case 'd': ii = va_arg(ap,int);
                   printf("%d",ii);
                   break;
                case 'f': dd = va_arg(ap,double);
                   printf("%f",dd);
                   break;
                default: printf("%c",*p);
             }      /* Конец переключателя   */
       }/* Конец цикла просмотра строки-формата */
   va_end(ap);/*Подготовка к завершению функции */
}
void main()
{
   void miniprint(char *, ...); /* Прототип */
   int k = 154;
   double e = 2.718282;
   miniprint("\nЦелое k= %d,\tчисло e= %f",k,e);
}
```

Результат выполнения программы:

```
Целое k= 154,        число e= 2.718282
```

Интересной особенностью предложенной функции miniprint**( )** и ее серьезных прародителей – библиотечных функций языка Си **printf( )** и **scanf( )** – является использование одного явного параметра и для задания типов последующих аргументов, и для определения их количества. Для этого в строке, определяющей формат

вывода, записывается последовательность спецификаций, каждая из которых начинается символом '**%**'. Предполагается, что количество спецификаций равно количеству аргументов в следующем за форматом списке. Конец обмена и перебора аргументов определяется по достижении конца форматной строки, когда *p = = '\0'.

5.6. Рекурсивные функции

Рекурсивной называют функцию, которая прямо или косвенно сама вызывает себя.

При каждом обращении к рекурсивной функции создается новый набор объектов автоматической памяти, локализованных в теле функции.

Рекурсивные алгоритмы эффективны, например, в тех задачах, где рекурсия использована в определении обрабатываемых данных. Поэтому серьезное изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой. Рассмотрим вначале только принципиальные возможности, которые предоставляет язык Си для организации рекурсивных алгоритмов.

Различают прямую и косвенную рекурсии. Функция называется *косвенно рекурсивной* в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов определяемой (первой) функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в теле функции явно используется вызов этой же функции, то имеет место *прямая рекурсия*, то есть функция, по определению, рекурсивная (иначе – самовызываемая или самовызывающая: *self-calling*). Классический пример – функция для вычисления факториала неотрицательного целого числа.

```
long fact(int k)
{ if (k < 0) return 0;
  if (k == 0) return 1;
  return k * fact(k-1);
}
```

Для отрицательного аргумента результата (по определению факториала) не существует. В этом случае функция возвратит нулевое значение. Для нулевого параметра функция возвращает значение 1, так как, по определению, 0! равен 1. В противном случае вызывается та же функция с уменьшенным на 1 значением параметра и

результат умножается на текущее значение параметра. Тем самым для положительного значения параметра **k** организуется вычисление произведения

```
k * (k-1) * (k-2) *...* 3 * 2 * 1 * 1
```

Обратите внимание, что последовательность рекурсивных обращений к функции fact( ) прерывается при вызове fact(0). Именно этот вызов приводит к последнему значению 1 в произведении, так как последнее выражение, из которого вызывается функция, имеет вид: `1*fact(1-1)`.

В языке Си отсутствует операция возведения в степень, и следующая рекурсивная функция вычисления целой степени вещественного ненулевого числа может оказаться полезной (следует отметить, что в стандартной библиотеке есть функция **pow( )** для возведения в степень данных типа **double**. См. приложение 3):

```
double expo(double a, int n)
{ if (n == 0) return 1;
   if (a == 0.0) return 0;
   if (n > 0) return a * expo(a, n-1);
   if (n < 0) return expo(a, n+1) / a;
}
```

При обращении вида expo(2.0, 3) рекурсивно выполняются вызовы функции expo( ) с изменяющимся вторым аргументом: expo(2.0,3), expo(2.0,2), expo(2.0,1), expo(2.0,0). При этих вызовах последовательно вычисляется произведение

```
2.0 * 2.0 * 2.0 * 1
```

и формируется нужный результат.

Вызов функции для отрицательного значения степени, например:

```
expo(5.0,-2)
```

эквивалентен вычислению выражения

```
expo(5.0,0) / 5.0 / 5.0
```

Отметим математическую неточность. В функции expo( ) для любого показателя при нулевом основании результат равен нулю, хотя

возведение в нулевую степень нулевого основания должно приводить к ошибочной ситуации.

В качестве еще одного примера рекурсии рассмотрим функцию определения с заданной точностью eps корня уравнения $f(x) = 0$ на отрезке $[a, b]$. Предположим, что исходные данные задаются без ошибок, то есть eps > 0, $b > a$, $f(a) * f(b) < 0$, и вопрос о возможности существования нескольких корней на отрезке $[a, b]$ нас не интересует. Не очень эффективная рекурсивная функция для решения поставленной задачи содержится в следующей программе:

```
#include <stdio.h>
#include <math.h> /*Для математических функций*/
#include <stdlib.h>    /* Для функции exit() */
/* Рекурсивная функция для определения корня математической функции
методом деления пополам: */
double recRoot(double f(double), double a, double b, double eps)
{
   double fa = f(a), fb = f(b), c, fc;
   if (fa * fb > 0)
    {
     printf("\nНеверен интервал локализации "
            "корня!");
      exit(1);
    }
   c = (a + b)/2.0;
   fc = f(c);
   if (fc == 0.0 || b - a < eps)
      return c;
   return (fa * fc < 0.0) ? recRoot(f, a, c,eps):
                            recRoot(f, c, b,eps);
}
int counter=0; /*Счетчик обращений к тестовой функции */
void main()
{
   double root,
      A=0.1, /* Левая граница интервала */
      B = 3.5, /* Правая граница интервала */
      EPS = 5e-5; /* Точность локализации корня */
   double giper(double); /* Прототип тестовой функции  */
   root = recRoot(giper, A, B, EPS);
   printf("\nЧисло обращений к тестовой функции "
         "= %d",counter);
```

```
   printf("\nКорень = %f",root);
}
/* Определение тестовой функции: */
double giper(double x)
{
   counter++; /*Счетчик обращений - глобальная переменная */
   return (2.0/x * cos(x/2.0));
}
```

Результат выполнения программы:

```
Число обращений к тестовой функции = 54
Корень = 3.141601
```

В рассматриваемой программе пришлось использовать библиотечную функцию **exit( )**, прототип которой размещен в заголовочном файле **stdlib.h**. Функция **exit( )** позволяет завершить выполнение программы и возвращает операционной системе значение своего аргумента.

Неэффективность предложенной программы связана, например, с излишним количеством обращений к программной реализации функции, для которой определяется корень. При каждом рекурсивном вызове recRoot**( )** повторно вычисляются значения f(a), f(b), хотя они уже известны после предыдущего вызова. Предложите свой вариант исключения лишних обращений к f( ) при сохранении рекурсивности.

В литературе по программированию рекурсиям уделено достаточно внимания как в теоретическом плане, так и в плане рассмотрения механизмов реализации рекурсивных алгоритмов. Сравнивая рекурсию с итерационными методами, отмечают, что рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно (см., например, Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.). В тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений, гораздо эффективнее применять итеративные методы. Таким образом, определение корня математической функции, возведение в степень и вычисление факториала только иллюстрируют схемы организации рекурсивных функций, но не являются примерами эффективного применения рекурсивного подхода к вычислениям.

5.7. Классы памяти и организация программ

Локализация объектов. До сих пор в примерах программ мы использовали в основном только два класса памяти – автоматическую и динамическую память. (В последнем примере из §5.6 использована глобальная переменная **int** counter, которая является внешней по отношению к функциям программ **main( )**, giper( ).) Для обозначения *автоматической памяти* могут применяться спецификаторы классов памяти **auto** или **register**. Однако и без этих ключевых слов-спецификаторов класса памяти любой объект (например, массив или переменная), определенный внутри блока (например, внутри тела функции), воспринимается как объект автоматической памяти. Объекты автоматической памяти существуют только внутри того блока, где они определены. При выходе из блока память, выделенная объектам типа **auto** или **register**, освобождается, то есть объекты исчезают. При повторном входе в блок для тех же объектов выделяются новые участки памяти, содержимое которых никак не зависит от «предыстории». Говорят, что объекты автоматической памяти локализованы внутри того блока, где они определены, а время их существования (время «жизни») определяется присутствием управления внутри этого блока. Другими словами, автоматическая память всегда внутренняя, то есть к ней можно обратиться только в том блоке, где она определена.

Пример:

```
#include   <stdio.h>
/*  Переменные  автоматической  памяти  */
void  autofunc(void)
{
   int  K=1;
   printf("\tK=%d",K);
   K++;
   return;
}
void  main()
{
   int  i;
   for  (i=0;i<5;i++)
      autofunc();
}
```

Результат выполнения программы:

```
K=1         K=1         K=1         K=1         K=1
```

Результат выполнения этой программы очевиден, и его можно было бы не приводить, если бы не существовало еще одного класса внутренней памяти – *статической внутренней памяти*. Рассмотрим программу, очень похожую на приведенную:

```
#include   <stdio.h>
/* Локальные переменные статической памяти */
void stat(void)
{
   static int K=1;
   printf("\tK=%d",K);
   K++;
   return;
}
void main()
{
   int i;
   for (i=0;i<5;i++)
      stat();
}
```

Результат выполнения программы:

```
K=1         K=2         K=3         K=4         K=5
```

Отличие функций autofunc( ) и stat( ) состоит в наличии спецификатора **static** при определении переменной **int** K, локализованной в теле функции stat( ). Переменная K в функции autofunc( ) – это переменная автоматической памяти, она определяется и инициализируется при каждом входе в функцию. Переменная **static int** K получает память и инициализируется только один раз. При выходе из функции stat( ) последнее значение внутренней статической переменной K сохраняется до последующего вызова этой же функции. Сказанное иллюстрирует результат выполнения программы.

Глобальные объекты. Следует обратить внимание на возможность использовать внутри блоков объекты, которые по месторасположению своего определения оказываются глобальными по отношению к операторам и определениям блока. Напомним, что блок – это не

только тело функции, но и любая последовательность определений и операторов, заключенная в фигурные скобки.

Выше в рекурсивной программе для вычисления приближенного значения корня математической функции переменная counter для подсчета количества обращений к тестовой функции giper( ) определена как глобальная для всех функций программы. Вот еще один пример программы с глобальной переменной:

```
#include  <stdio.h>
int N=5; /* Глобальная переменная */
void func(void)
{
   printf("\tN=%d",N);
   N--;
   return;
}
void main()
{
   int i;
   for (i=0;i<5;i++)
      {
        func();
        N+=2;
      }
}
```

Результат выполнения программы:

```
N=5       N=6       N=7       N=8       N=9
```

Переменная **int** N определена вне функций **main( )** и func( ) и является глобальным объектом по отношению к каждой из них. При каждом вызове func( ) значение N уменьшается на 1, в основной программе – увеличивается на 2, что и определяет результат.

Глобальный объект может быть «затенен» или «скрыт» внутри блока определением, в котором для других целей использовано имя глобального объекта. Модифицируем предыдущую программу:

```
#include  <stdio.h>
int N=5; /* Глобальная переменная */
void func(void)
{
   printf("\tN=%d",N);
```

```
   N--;
   return;
}
void main()
{
   int N; /* Локальная переменная */
   for (N=0;N<5;N++)
       func();
}
```

Результат выполнения программы:

```
N=5        N=4        N=3        N=2        N=1
```

Переменная **int** N автоматической памяти из функции **main( )** никак не влияет на значение глобальной переменной **int** N. Это разные объекты, которым выделены разные участки памяти. Внешняя переменная N «не видна» из функции **main( )**, и это результат определения **int** N внутри нее.

Динамическая память – это память, выделяемая в процессе выполнения программы. А вот на вопрос «Глобальный или локальный объект размещен в динамической памяти?» попытаемся найти правильный ответ.

После выделения динамической памяти она сохраняется до ее явного освобождения, что может быть выполнено только с помощью специальной библиотечной функции **free( )**.

Если динамическая память не была освобождена до окончания выполнения программы, то она освобождается автоматически при завершении программы. Тем не менее явное освобождение ставшей ненужной памяти является признаком хорошего стиля программирования.

В процессе выполнения программы участок динамической памяти доступен везде, где доступен указатель, адресующий этот участок. Таким образом, возможны следующие три варианта работы с динамической памятью, выделяемой в некотором блоке (например, в теле неглавной функции):

- указатель (на участок динамической памяти) определен как локальный объект автоматической памяти. В этом случае выделенная память будет недоступна при выходе за пределы блока локализации указателя, и ее нужно освободить перед выходом из блока;

- указатель определен как локальный объект статической памяти. Динамическая память, выделенная однократно в блоке, доступна через указатель при каждом повторном входе в блок. Память нужно освободить только по окончании ее использования;
- указатель является глобальным объектом по отношению к блоку. Динамическая память доступна во всех блоках, где «виден» указатель. Память нужно освободить только по окончании ее использования.

Проиллюстрируем второй вариант, когда объект динамической памяти связан со статическим внутренним (локализованным) указателем:

```
#include <stdio.h>
#include <alloc.h>   /* Для функций malloc( ), free( ) */
void dynam(void)
{
   static char *uc=NULL; /* Внутренний указатель */
  /* Защита от многократного выделения памяти: */
   if(uc == NULL)
      {
         uc=(char*)malloc(1);
         *uc='A';
      }
   printf("\t%c",*uc);
   (*uc)++;
   return;
}
void main( )
{
   int i;
   for (i=0; i<5; i++)
      dynam( );
}
```

Результат выполнения программы:

```
A       B       C       D       E
```

В следующей программе указатель на динамический участок памяти является глобальным объектом:

```
#include <stdio.h>
#include <alloc.h>   /* Для функций malloc( ), free( ) */
char * uk=NULL;    /* Глобальный указатель */
void dynam1 (void)
{
   printf ("\t%c", * uk);
   (* uk)++;
}
void main (void)
{
   int i;
   uk=(char*) malloc (1);
   *uk='A';
   for (i=0; i<5; i++)
      {
       dynam1( );
       (*uk)++;
      }
   free(uk);
}
```

Результат выполнения программы:

```
A       C       E       G       I
```

Динамический объект создается в основной функции и связывается с указателем uk. Там же он явным присваиванием получает начальное значение 'A'. За счет глобальности указателя динамический объект доступен в обеих функциях **main( )** и **dynam1( )**. При выполнении цикла в функции **main( )** и внутри функции **dynam1( )** изменяется значение динамического объекта.

Приводить пример для случая, когда указатель, адресующий динамически выделенный участок памяти, является объектом автоматической памяти, нет необходимости. Заканчивая обсуждение данной темы, отметим, что динамическая память после выделения доступна везде (в любой функции и в любом файле), где указатель, связанный с этой памятью, определен или описан как внешний объект. Следует только четко определить понятие внешнего объекта, к чему мы сейчас и перейдем.

Внешние объекты. Здесь в конце главы, посвященной функциям, самое подходящее время, чтобы взглянуть в целом на программу, состоящую из набора функций, размещенных в нескольких текстовых

файлах. Именно такой вид обычно имеет более или менее серьезная программа на языке Си. Отдельный файл с текстами программы иногда называют программным модулем, но нет строгих терминологических соглашений относительно модулей или файлов с текстами программы. На рис. 5.3 приведена схема программы, текст которой находится в двух файлах. Программа, как мы уже неоднократно говорили, представляет собой совокупность функций. Все функции внешние, так как внутри функции по правилам языка Си нельзя определить другую функцию. У всех функций, даже размещенных в разных файлах, должны быть различные имена.

Кроме функций, в программе могут использоваться *внешние_объекты* – переменные, указатели, массивы и т. д. Внешние объекты должны быть определены вне текста функций.

Внешние объекты могут быть доступны из многих функций программы, однако эта доступность не всегда реализуется автоматически – в ряде случаев нужно дополнительное вмешательство программиста. Если объект определен в начале файла с программой, то он является глобальным для всех функций, размещенных в файле, и доступен в них без всяких дополнительных предписаний. (Ограничение – если внутри функции имя глобального объекта использовано в качестве имени внутреннего объекта, то внешний объект становится недостижимым, то есть «невидимым» в теле именно этой функции.) На рис. 5.3:

- объект X: доступен в f11( ), f12( ) как глобальный; доступен как внешний в файле 2 только в тех функциях, где будет помещено описание **extern** X;
- объект Y: доступен как глобальный в f21( ) и f22( ); доступен как внешний в тех функциях файла 1, где будет помещено описание **extern** Y;
- объект Z: доступен как глобальный в f22( ) и во всех функциях файла 1 и файла 2, где помещено описание **extern** Z.

Если необходимо, чтобы внешний объект был доступен для функций из другого файла или функций, размещенных выше определения объекта, то он должен быть перед обращением дополнительно описан с использованием дополнительного ключевого слова **extern**. (Наличие этого слова по умолчанию предполагается и для всех функций, то есть не требуется в их прототипах.) Такое описание, со спецификатором слова **extern**, может помещаться в начале файла, и тогда объект доступен во всех функциях файла. Если это описание размещено в теле одной функции, тогда объект доступен именно в ней.

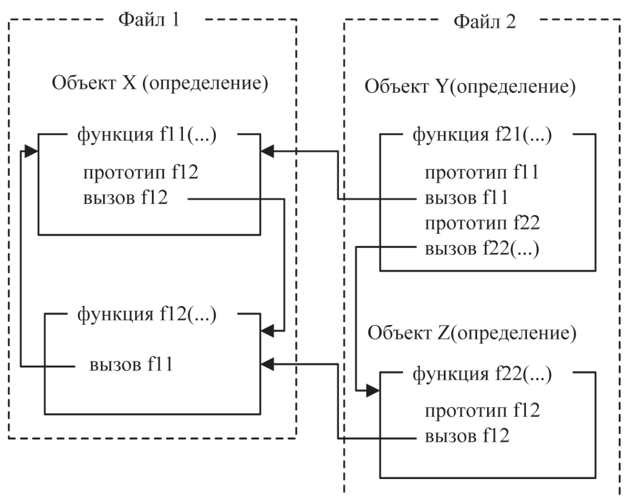


Рис. 5.3. *Схема программы, размещенной в двух файлах*

Описание внешнего объекта не есть его определение. Помните: в определении объекту всегда выделяется память, и он может быть инициализирован. Примеры определений:

```
double summa [5];
char D_Phil [ ]="Doctor of Philosophy";
long M=1000;
```

В описаниях инициализация невозможна, нельзя указать и количество элементов массивов:

```
extern double summa [ ];
extern char D_Phil [ ];
extern long M;
```

5.8. Параметры функции main( )

В соответствии с синтаксисом языка Си основная функция каждой программы может иметь такой заголовок:

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

Параметр argv – массив указателей на строки; argc – параметр типа **int**, значение которого определяет размер массива argv, то есть количество его элементов, envp – параметр-массив указателей на символьные строки, каждая из которых содержит описание одной из переменных среды (окружения). Под *средой* понимается та программа (обычно это операционная система), которая «запустила» на выполнение функцию **main( )**.

Назначение параметров функции **main( )** – обеспечить связь выполняемой программы с операционной системой, точнее с командной строкой, из которой запускается программа и в которую можно вносить данные и тем самым передавать исполняемой программе любую информацию.

Если внутри функции **main( )** нет необходимости обращаться к информации из командной строки, то параметры обычно опускаются.

При запуске программы в командной строке записывается имя выполняемой программы. Вслед за именем можно разместить нужное количество «слов», разделяя их друг от друга пробелами. Каждое «слово» из командной строки становится строкой-значением, на которую указывает очередной элемент параметра argv[i], где $0<i<argc$. Здесь нужно сделать одно уточнение.

Как и в каждом массиве, в массиве argv[ ] индексация элементов начинается с нуля, то есть всегда имеется элемент argv[0]. Этот элемент является указателем на полное название запускаемой программы. Например, если из командной строки в ОС Windows выполняется обращение к программе EXAMPLE из каталога CATALOG, размещенного на диске С, то вызов выглядит так:

```
C:\CATALOG\EXAMPLE.EXE
```

Значение argc в этом случае будет равно 1, а строка, которую адресует указатель argv[0], будет такой:

```
"C:\CATALOG\EXAMPLE.EXE"
```

В качестве иллюстрации сказанного приведем программу, которая выводит на экран дисплея всю информацию из командной строки, размещая каждое «слово» на новой строке экрана.

```
#include <stdio.h>
void main (int argc, char * argv [ ])
```

```
{
   int i;
   for (i=0; i<argc; i++)
      printf("\n argv [%d] -> %s", i, argv [i];
}
```

Пусть программа запускается из такой командной строки (в ОС Windows):

```
C:\VVP\test66 11  22   33
```

Результат выполнения программы:

```
argv [0] -> C:\VVP\test66.exe
argv [1] -> 11
argv [2] -> 22
argv [3] -> 33
```

В главной программе **main( )** разрешено использовать и третий параметр **char** * envp[ ]. Его назначение – передать в программу всю информацию об окружении, в котором выполняется программа. Следующий пример иллюстрирует возможности этого третьего параметра функции **main( )**.

```
#include <stdio.h>
void main (int argc, char *argv[], char *envp[])
{
   int n;
   printf("\nПрограмма '%s' "
         "\nЧисло параметров при запуске - %d",
         argv[0], argc-1;
   for (n=1; n<argc; n++)
      printf("\n%d-й параметр: %s", n, argv[n]);
   printf("\n\nСписок переменных окружения:");
   for (n=0; envp[n]; n++)
      printf("\n%s", envp[n]);
}
```

Пусть программа «запущена» на выполнение из такой командной строки (в ОС Windows):

```
C:\WWP\TESTPROG\MAINENVP.EXE qqq www
```

Результаты выполнения программы:

```
Программа 'C:\WWP\TESTPROG\MAINENVP.EXE'
Число параметров при запуске - 2
1-й параметр: qqq
2-й параметр: www

Список переменных окружения:
TMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\WINDOWS\COMMAND.COM
TEMP=c:\windows\temp
CLASSPATH=.;c:\cafe\java\lib\classes.zip
HOMEDRIVE=c:
HOMEPATH=\cafe\java
JAVA_HOME=c:\cafe\java
windir=C:\WINDOWS
NWLANGUAGE=English
PATH=C:\CAFE\BIN;C:\CAFE\JAVA\BIN;C:\BC5\BIN;
     C:\WINDOWS;C:\WINDOWS\COMMAND;C:\NC;
     C:\ARC;C:\DOS;C:\VLM;C:\TC\BIN
CMDLINE=tc
```

Нет необходимости в нашем пособии разбирать составные части окружающей среды, в которой выполняется программа. Поэтому мы не станем комментировать результаты, выводимые рассмотренной программой.

Контрольные вопросы

1. Приведите формат определения функции.
2. Укажите роль прототипа функции и правила его размещения.
3. Что такое «спецификация параметров»?
4. Объясните соотношение между участками памяти, выделяемыми для параметров и аргументов функции.
5. Могут ли определения функции быть вложенными?
6. Какой способ передачи параметров предусматривает синтаксис языка Си при обращении к функциям?
7. Припомните последовательность шагов при передаче параметров функции.

8. Объясните особенность применения указателей в параметрах функции.
9. Могут ли изменяться аргументы-указатели за счет исполнения операторов тела функции?
10. Каким образом в теле функции можно получить доступ к внешнему, по отношению к функции, объекту, использованному в качестве аргумента?
11. Объясните использование массивов в параметрах функции.
12. Укажите различия между массивом в месте его определения и массивом-параметром в теле функции.
13. Допускается ли использование указателей со смещениями вместо индексированных переменных в определениях функций?
14. Соответствует ли действительности следующее положение: «при каждом обращении к функции ее параметрам выделяются участки памяти, полностью независимые и отличные от участков памяти, выделенных для аргументов, использованных в вызове функции»?
15. В каких случаях необходимо использовать указатели на функции?
16. Приведите формат прототипа функции с переменным количеством аргументов.
17. Какие требования предъявляются к функциям с переменным списком параметров?
18. Приведите определение рекурсивной функции.
19. Какими способами реализуются итерационные и рекурсивные алгоритмы?
20. Можно ли итерационный алгоритм заменить на адекватный ему рекурсивный?
21. Объясните различия между прямой и косвенной рекурсией.
22. Какие спецификаторы классов памяти применяются для обозначения автоматической памяти?
23. Укажите свойства глобального объекта.
24. Перечислите варианты работы с динамической памятью.
25. Укажите свойства внешнего объекта.
26. Перечислите параметры функции **main()**.
27. Объясните, для чего служат указатель типа **va_list** и макрос **va_arg()**.

Глава 6
СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

6.1. Структурные типы и структуры

Производные типы. Из базовых типов (и типов, определенных программистом) можно формировать производные типы, к которым относятся: указатели, массивы, функции, структуры и объединения. Указатели, массивы, структуры и объединения являются производными типами данных. Иногда в качестве производного типа в руководствах по языку Си указывают строки, однако напоминаем, что в соответствии с синтаксисом языка строковый тип в нем отсутствует. Строка определяется как частный случай одномерного массива, то есть как массив с элементами типа **char**, последний элемент которого равен '\0'.

Указатели и массивы подробно рассмотрены в предыдущих главах. Эта глава посвящена структурам и объединениям. Прежде чем рассматривать их особенности и возможности, введем некоторые терминологические соглашения стандарта, которые не всегда понятны сами собой.

Данные базовых типов (**int**, **float**, ...) считаются скалярными данными. Массивы и структуры являются агрегирующими типами данных, в отличие от объединений и скалярных данных, которые относятся к неагрегирующим типам. С массивами и скалярными данными мы уже хорошо знакомы, поэтому различие между агрегирующими и неагрегирующими типами поясним на примере массивов и скалярных данных базовых типов. Обычно агрегирующий тип включает несколько компонентов. Например, массив **long** B[12] состоит из 12 элементов, причем каждый элемент имеет тип **long**. Язык Си позволяет определять и одноэлементные массивы. Так, массив **float** A[1] состоит из одного элемента A[0]. Однако одноэлементные массивы есть слишком частный случай обыкновенных массивов со многими элементами. Итак, в общем случае массив

есть агрегирующий тип данных, состоящий из набора однотипных элементов, размещенных в памяти подряд в соответствии с ростом индекса. Для агрегирующего типа данных (например, для массива) в основной памяти ЭВМ выделяется такое количество памяти, чтобы сохранять (разместить) одновременно значения всех элементов. Этим свойством массивов мы уже неоднократно пользовались. Например, следующее выражение позволяет вычислить количество элементов массива B[ ]:

sizeof (B) / **sizeof** (B [0])

Для данных неагрегирующих типов в основной памяти ЭВМ выделяется область памяти, достаточная для размещения только одного значения. С тем, как это происходит для скалярных типов, мы уже хорошо знакомы. С особенностями выделения памяти для объединений познакомимся в этой главе.

Структурный тип. Этот тип (производный агрегирующий) задает внутреннее строение определяемых с его помощью структур. Сказанное требует пояснений. Начнем с понятия структуры. *Структура* – это объединенное в единое целое множество поименованных элементов (компонентов) данных. В отличие от массива, всегда состоящего из однотипных элементов, компоненты структуры могут быть разных типов и все должны иметь различные имена. Например, может быть введена структура, описывающая товары на складе, с компонентами:

- название товара (**char***);
- оптовая (закупочная) цена (**long**);
- торговая наценка в процентах (**float**);
- объем партии товара (**int**);
- дата поступления партии товара (**char** [9]).

В перечислении элементов структуры «товары на складе» добавлены выбранные программистом типы этих элементов. В соответствии со смыслом компоненты могут иметь любой из типов данных, допустимых в языке. Так как товаров на складе может быть много, для определения отдельных структур, содержащих сведения о конкретных товарах, программист вводит производный тип, называемый *структурным*. Для нашего примера его можно ввести, например, так:

```
struct goods {
char* name;    /* Наименование */
```

```
long price;    /* Цена оптовая */
float percent; /* Наценка в % */
int vol;       /* Объем партии */
char date [9]; /* Дата поставки партии */
};
```

Здесь **struct** – спецификатор структурного типа (служебное слово); goods – предложенное программистом название (имя) структурного типа. (Используя транслитерацию английского термина «tag», название структурного типа в русскоязычной литературе по языку Си довольно часто называют тегом.) В фигурных скобках размещаются описания элементов, которые будут входить в каждый объект типа goods. Итак, формат определения структурного типа таков:

struct *имя_структурного_типа*
{ *определения_элементов* };

где **struct** – спецификатор структурного типа; *имя_структурного_типа* – идентификатор, произвольно выбираемый программистом; *определения_элементов* – совокупность одного или более описаний объектов, каждый из которых служит прототипом для элементов структур вводимого структурного типа.

Следует обратить внимание, что определение структурного типа (в отличие от определения функции) заканчивается точкой с запятой.

Конструкция **struct** *имя_структурного_типа* играет ту же роль, что и спецификаторы типов, например **double** или **int**. С помощью **struct** goods можно либо определить конкретную структуру (как, например, объект структурного типа **struct** goods), либо указатель на структуры такого типа. Пример:

```
/* Определение структуры с именем food */
struct goods food;
/* Определение указателя point_to на структуры*/
struct goods *point_to;
```

Кроме такого прямого определения поименованного структурного типа, может быть введен безымянный структурный тип и не полностью определенный (незавершенный) структурный тип. О безымянном структурном типе речь пойдет в этом параграфе чуть позже при определении структур как объектов. Не полностью определенный, то есть незавершенный структурный, тип потребуется в следующем параграфе при рассмотрении указателей на структуры, вводимые

в качестве элементов структур. Кстати, отметим, что незавершенным может быть не только структурный тип. В общем смысле незавершенным считается любое определение, в котором не содержится достаточно информации о размере будущего объекта.

Еще одну возможность ввести структурный тип дает служебное слово **typedef**, позволяющее ввести собственное обозначение для любого определения типа. В случае структурного типа он может быть введен и поименован следующим образом:

typedef struct {*определения_элементов*}
обозначение_структурного_типа;

Пример:

```
typedef   struct
{   double  real;
    double  imag;
}
complex;
```

Приведенное определение вводит структурный тип **struct** {**double** real; **double** imag;} и присваивает ему обозначение (название, имя) complex. С помощью этого обозначения можно вводить структуры (объекты) так же, как с обычным именем структурного типа (например, **struct** goods в предыдущем примере). Пример:

```
/*  Определены  две  структуры:  */
complex  sigma,  alfa;
```

Структурный тип, которому программист назначает имя с помощью **typedef**, может в то же время иметь второе имя, вводимое стандартным образом после служебного слова **struct**. В качестве примера введем определение структурного типа «рациональная дробь». Будем считать, что рациональная дробь – это пара целых чисел (m;n), где число n отлично от нуля. Определение такой дроби:

```
typedef  struct  rational_fraction
{  int  numerator;   /*  Числитель  */
   int  denominator; /*  Знаменатель  */
}  fraction;
```

Здесь fraction – обозначение структурного типа, вводимое с помощью **typedef**. Имя rational_fraction введено для того же струк-

турного типа стандартным способом. После такого определения структуры типа «рациональная дробь» могут вводиться как с помощью названия fraction, так и с помощью обозначения **struct** rational_fraction.

С помощью директивы **#define** можно вводить имена типов подобно тому, как это делается с помощью **typedef**. Например, сведения о книге могут быть введены таким образом:

```
#define BOOK struct \
{ char author[20];   /* Автор */   \
   char title [80];  /* Название */   \
   int year;         /* Год издания */   \
}
```

Здесь BOOK – препроцессорный идентификатор, вслед за которым в нескольких строках размещена «строка замещения». Обратите внимание на использование символа продолжения '\' для переноса строковой константы. Кроме того, отметим отсутствие точки с запятой после закрывающей скобки '}'. Далее в программе можно определять конкретные объекты-структуры или указатели с помощью имени BOOK, введенного на препроцессорном уровне:

```
BOOK ss, *pi;
```

После препроцессорных замен и исключения комментариев это предложение примет вид:

```
struct
{
   char author[20];
   char title [80];
   int year;
}
ss, *pi;
```

Определение структур. Определения элементов (компонентов) структурного типа внешне подобны определениям данных соответствующих типов. Однако имеется существенное отличие. При определении структурного типа его компонентам не выделяется память, и их нельзя инициализировать. Другими словами, структурный тип не является объектом.

Из нашего примера определения структурного типа с названием goods следует, что наименование товара будет связано с указателем типа **char***, имеющим имя name. Оптовая цена единицы товара будет значением элемента типа **long** с названием price. Торговая наценка будет значением элемента типа **float** с именем percent и т. д. Все это следует из приведенного определения структурного типа с названием goods. Но прежде чем элементы, введенные в определении структурного типа, смогут получить значения, должна быть определена хотя бы одна структура (то есть структурный объект) этого типа. Например, следующее определение вводит две структуры, то есть два объекта, типа goods:

```
struct   goods   coat,  tea;
```

Итак, если структурный тип определен и известно его имя, то формат определения конкретных структур (объектов структурного типа) имеет вид:

struct *имя_структурного_типа* *список_структур;*

где *список_структур* – список выбранных пользователем имен (идентификаторов).

Выше показано, что для структурного типа, имя которого введено с помощью служебного слова **typedef**, определение структур не должно содержать спецификатора типа **struct**. Например, указатель на структуры для представления комплексных чисел с помощью определенного выше обозначения структурного типа complex можно определить так:

```
complex   *p_comp;
```

Выше был определен структурный тип «рациональная дробь», для которого введены два имени. С их помощью так вводятся конкретные структуры:

```
     /*  Две  структуры:  */
struct   rational_fraction   ratio_1,  ratio_2;
     /*  Две  структуры:  */
fraction   zin,  rat_X_Y;
```

Кроме изложенной последовательности определения структур (определяется структурный тип, затем с использованием его имени

определяются структуры), в языке Си имеются еще две схемы их определения. Во-первых, структуры могут быть определены одновременно с определением структурного типа:

struct *имя_структурного_типа*
{ *определения_элементов* }
список_структур;

Пример одновременного определения структурного типа и структур (объектов):

```
struct student
{
   char name [15]; /* Имя */
   char surname [20]; /* Фамилия */
   int year; /* Курс */
} student_1, student_2, student_3;
```

Здесь определен структурный тип с именем student и три конкретные структуры student_1, student_2, student_3, которые являются полноправными объектами. В каждую из этих трех структур входят элементы, позволяющие представить имя (name), фамилию (surname), курс (year), на котором обучается студент.

После приведенного определения в той же программе можно определять любое количество структур, используя структурный тип student:

```
struct  student  leader, freshman;
```

Следующий вариант определения структур является некоторым упрощением приведенного варианта. Дело в том, что можно определять структуры, приведя «внутреннее строение» структурного типа, но не вводя его названия. Такой безымянный структурный тип обычно используется в программе для однократного определения структур:

struct
{ *определения_элементов* }
список_структур;

В качестве примера определим структуры, описывающие конфигурацию персонального компьютера с такими характеристиками:

- тип процессора (**char** [10]);
- рабочая частота в ГГц (**int**);

- объем основной памяти в Гбайтах (**int**);
- емкость жесткого диска в Гбайтах (**int**).

Пример определения структур безымянного типа:

```
struct  {
          char processor [10];
          int frequency;
          int memory;
          int disk;
} ACER, HP, Compaq;
```

В данном случае введены три структуры (три объекта) с именами ACER, HP, Compaq. В каждую из определенных структур входят элементы, в которые можно будет занести сведения о характеристиках конкретных ПК. Структурный тип «компьютер» не именован. Поэтому если в программе потребуется определять другие структуры с таким же составом элементов, то придется полностью повторить приведенное выше определение структурного типа.

Выделение памяти для структур. Мы уже договорились, что определение структурного типа не связано с выделением памяти, а при каждом определении структуры (объекта) ей выделяется память в таком количестве, чтобы могли разместиться данные всех элементов. На рис. 6.1 приведены условные схемы распределения памяти для одного из объектов (структур) типа goods. На первой схеме элементы структуры размещены подряд без пропусков между ними. Однако никаких гарантий о таком непрерывном размещении элементов структур стандарт языка Си не дает. Причиной появления неиспользованных участков памяти («дыр») могут явиться требования выравнивания данных по границам участков адресного пространства. Эти требования зависят от реализации, от аппаратных возможностей системы и иногда от режимов (опций) работы компилятора. На рис. 6.1 условно изображен и второй вариант с пропуском участка памяти между элементами **float** percent и **int** vol. Пропуск («дыра») может быть и после последнего элемента структуры. В этом случае память для следующего объекта, определенного в программе, будет выделена не сразу после структуры, а с промежутком, оставляемым для выравнивания по принятой границе участка адресного пространства.

Необходимость в выравнивании данных зависит от конкретной задачи. Например, доступ к целым значениям выполняется быстрее, если они имеют четные адреса, то есть выровнены по границам

Определение структурного типа:
struct goods { **char*** name; **long** price; **float** percent;
int vol; **char** date [9]
};

Элементы размещены подряд

Названия элементов:	name	price	percent	vol	date
их типы:	**char***	**long**	**float**	**int**	**char[9]**
байты:	←4→	←4→	←4→	←2→	←9→

Размещение элементов с выравниванием данных

Названия элементов:	name	price	percent	"дыра"	vol	date
их типы:	**char***	**long**	**float**		**int**	**char[9]**
байты:	←4→	←4→	←4→		←2→	←9→

Рис. 6.1. *Размещение структуры типа goods в памяти (размеры в байтах для разных реализаций могут быть другими)*

машинных слов. Противоположное требование состоит в плотной «упаковке» информации, когда идет борьба за уменьшение объема, занимаемого в памяти структурой или массивом структур. Влиять на размещение структур можно с помощью препроцессорной директивы #**pragma** (см. главу 3).

В зависимости от наличия «пропусков» между элементами изменяется общий объем памяти, выделяемый для структуры. Реальный размер памяти в байтах, выделяемый для структуры, можно определить с помощью операции

sizeof (*имя_структуры*)
sizeof (*имя_структурного_типа*)

Для нашего примера одинаковые результаты дадут операции:

```
sizeof (struct goods)
sizeof (tea)
sizeof coat
```

В последнем выражении имя структуры coat только для разнообразия помещено без скобок. Напомним, что операция определения размера имеет две формы:

sizeof (*имя_типа_данных*)
sizeof *выражение*

В случае операнда-выражения его не обязательно заключать в скобки, и его значение не вычисляется. Для него определяется тип и оценивается его размер (в байтах).

Примечание

На рис. 6.1 элементы несут следующую смысловую нагрузку:
- name – наименование товара;
- prict – оптовая цена;
- percent – наценка;
- vol – объем партии;
- date – дата поставки.

Инициализация и присваивание структур. Инициализация структур похожа на инициализацию массивов. Непосредственно в определении конкретной структуры после ее имени и знака '=' в фигурных скобках размещается список начальных значений элементов. Например:

```
struct goods coat={
   "пиджак черный", 3600, 7.5, 220, "12.01.2011" };
complex sigma={1.3, 12.6};
```

Сравнивая структуры с массивами, нужно обратить внимание на одну особенность, связанную с операцией присваивания. Если определены два массива одного и того же типа и одинаковых размеров, то присвоить элементам одного массива значения элементов другого массива можно только с помощью явного перебора элементов, например в цикле:

```
float z[5], x[5]={1.0, 2.0, 3.0, 4.0, 5.0 };
int j;
for (j=0; j<sizeof(x)/sizeof(x[0]); j++)
z[j]=x[j];
```

Попытка использовать имена массивов без индексов в операции присваивания будет обречена на провал (ведь имя массива есть неизменяемый указатель):

```
z=x; /* Ошибка в операции */
```

В то же время стандарт языка Си разрешает присваивание структур. Если не обращать внимания на смысл имен введенных выше

структур типа **struct** goods (tea – чай; coat – пиджак), то допустимо следующее присваивание:

```
tea=coat;
```

Определив структуру типа complex, можно выполнить, например, такое присваивание (структура sigma того же типа определена и инициализирована выше):

```
complex  sport;
sport=sigma;
```

Отметим, что для структур не определены операции сравнения даже на равенство. И сравнивать структуры нужно только поэлементно.

Доступ к элементам структур. Наиболее наглядно и естественно доступ к элементам структур обеспечивается с помощью уточненных имен. Конструкция

имя_структуры. имя_элемента

играет роль названия (имени) объекта того типа, к которому отнесен элемент в соответствии с определением структурного типа. В нашем примере с инициализацией структуры типа **struct** goods:

- coat.name – указатель типа **char*** на строку «пиджак черный»;
- coat.price – переменная типа **long** со значением 3600;
- coat.percent – переменная типа **float** со значением 7.5;
- coat.vol – переменная типа **int** со значением 220;
- coat.date – массив типа **char** [9], содержащий «12.01.2011».

Обратите внимание, что перед точкой стоит не название структурного типа, а имя конкретной структуры, для которой ее определением выделена память.

Уточненное имя – это выражение с двумя операндами и операцией «точка» между ними. Операция «точка» называется *операцией доступа к элементу структуры* (или объединения). У нее самый высокий ранг наряду со скобками (и операцией «стрелка» для доступа к элементам структуры через адресующий ее указатель, см. табл. 1.4).

Уточненное имя используется для выбора правого операнда операции «точка» из структуры (или объединения), задаваемой левым операндом. Левый операнд должен иметь структурный тип, а правый

операнд должен быть именем компонента (элемента) этой структуры. Тип результата операции «точка» – это тип именуемого уточненным именем компонента (элемента) структуры. Именно такие типы указаны в приведенных выше примерах, то есть coat.vol – объект типа **int** и т. д.

Если при определении структуры она инициализирована, то ее элементы получают соответствующие начальные значения. С помощью уточненных имен эти значения могут быть, например, выведены на экран дисплея.

Пример программы с инициализацией структуры и выводом значений ее элементов:

```
#include <stdio.h>
void main( )
{ struct goods {
                 char* name;
                 long price;
                 float percent;
                 int vol;
                 char date[9];
                };
   struct goods coat={
     "пиджак черный", 3600, 7.5, 220, "12.01.2011"};
   printf("\n Товар на складе:")
   printf("\n Наименование: %s.", coat.name);
   printf("\n Оптовая цена: %ld руб.", coat.price);
   printf("\n Наценка: %4.1f %%.", coat.percent);
   printf("\n Объем партии: %d штук.", coat.vol);
   printf("\n Дата поставки: %s.", coat.date);
}
```

Результат выполнения программы:

```
Товар на складе:
Наименование: пиджак черный.
Оптовая цена: 3600 руб.
Наценка: 7.5 %
Объем партии: 220 штук.
Дата поставки: 12.01.2011.
```

Уточненные имена элементов структур обладают всеми правами имен объектов соответствующих типов. Их можно использовать

в выражениях, их значения можно вводить с клавиатуры и т. д. Например, с помощью следующих операторов можно изменить торговую наценку (элемент coat.price) и вычислить розничную цену на определенный в программе товар (пиджак черный):

```
printf("\n Введите торговую наценку:");
scanf("%f", &coat.percent);
printf("Цена товара: %ld руб.",
      (long)(coat.price*(1.0+coat.percent/100));
```

Обратите внимание, что в качестве аргумента функции **scanf( )** используется адрес элемента percent структуры coat. Для этого операция получения адреса **&** применяется к уточненному имени coat.percent. При вычислении розничной цены товара приходится вводить явное приведение типов **(long)**, так как результат умножения элемента coat.price на вещественное выражение 1.0+coat.percent/100 имеет по умолчанию тип **double**.

Следующая программа выполняет сложение комплексных чисел, для представления которых использован структурный тип, имя которого вводит спецификатор **typedef**:

```
#include <stdio.h>
typedef struct {
                  double real;
                  double imag;
                } complex;
void main( )
{
   complex x, y, z;
   printf("\n Введите два комплексных числа:");
   printf("\n Вещественная часть: ");
   scanf(" %lf", &x.real);
   printf(" Мнимая часть:");
   scanf(" %lf", &x.imag);
   printf(" Вещественная часть:");
   scanf(" %lf", &y.real);
   printf(" Мнимая часть:");
   scanf(" %lf", &y.imag);
   z.real=x.real+y.real;
   z.imag=x.imag+y.imag;
   prinf("\n Результат: z.real=%f z.imag=%f", z.real, z.imag);
}
```

Возможный результат выполнения программы:

```
Введите два комплексных числа:
Вещественная часть: 1.3
Мнимая часть:       0.84
Вещественная часть: 2.2
Мнимая часть:       6.16
Результат: z.real= 3.500000  z.imag=7.000000
```

6.2. Структуры, массивы и указатели

Массивы и структуры в качестве элементов структур. В примерах предыдущего параграфа мы использовали в качестве элементов структур символьные массивы. Никаких особенных сложностей в этом нет. Массив, как и любые другие данные, может быть элементом структуры, и для обращения к каждому элементу такого массива потребуется индексирование. Приведем примеры. Пусть нужна структура, описывающая размещение в пространстве материальных точек (точечных масс). Элементами такой структуры могут быть:

- значение массы (**double**);
- массив координат (**float** [3]).

Проиллюстрируем основные особенности работы с такими структурами. Введя величину и координаты точечной массы, вычислим ее удаление от начала координат (модуль радиус-вектора).

```
#include <stdio.h>
#include <math. h>  /* Для функции sqrt( ) */
void main( )
{
   struct {
           double mass;
           float coord [3];
          }point={12.3, {1.0, 2.0, -3.0}};
   int i;
   float s=0.0;
   for (i=0, i<3; i++)
         s+=point.coord [i]* point.coord [i];
   printf("\n Длина радиус-вектора: %f", sqrt(s));
}
```

Результат выполнения программы:

```
Длина радиус-вектора: 3.741657
```

Для структуры point структурный тип введен как безымянный. Структура point получает значения своих элементов в результате инициализации. В списке начальных значений использовано вложение инициализаторов, так как элементом структуры является массив. Можно обратить внимание на индексирование при обращении к элементу внутреннего массива coord[3]. Индексы записываются после имени массива-элемента, а не после имени структуры.

Так как элементами структур могут быть данные любых типов, то естественным образом допускается вложение структур, то есть элементом структуры может быть другая (и только другая, не та же самая!) структура. В качестве примера еще раз рассмотрим структурный тип для представления сведений о студенте. Дополнительно к тем элементам, которые уже были (имя, фамилия, курс), введем в качестве элемента структуру с именем (названием) структурного типа **struct** birth. В ней будут сохраняться сведения о рождении – место рождения и дата рождения. В свою очередь, дату рождения представим структурой (типа date) с элементами число, месяц, год.

В тексте программы перечисленные структурные типы должны быть размещены в такой последовательности, чтобы использованный в определении структурный тип был уже определен ранее:

```
/* Определение структурных типов: */
struct date { /* тип "дата" */
        int day;    /* число */
        int month; /* месяц */
        int year;   /* год */
            };
struct birth { /* тип "данные о рождении" */
        char place [20] /* место рождения */
        struct date dd; /* дата рождения */
            };
struct student { /* тип "студент" */
        char name [15]; /* имя */
        char surname [20]; /* фамилия */
        int year; /* курс */
        struct birth bir;/* данные о рождении*/
            };
/* Работа с конкретным студентом: */
void main( )
{
/* Конкретная структура: */
```

```
struct student stud= {"Павел", "Иванов", 1,
                      "Петербург", 22, 11, 1888 };
printf("\n Введите курс студента: ");
scanf("%d", &stud.year);
printf("Сведения о студенте:");
printf("\n Фамилия: %s", stud.surname);
printf("\n Имя:%s", stud.name);
printf("\n Курс: %d-й", stud.year);
printf("\n Место рождения: %d", stud.bir.place);
printf("\n Дата рождения: %d.%d.%d",
          stud.bir.dd.day, stud.bir.dd.month, stud.bir.dd.year);
}
```

Результаты выполнения программы:

```
Введите курс студента:3
Сведения о студенте:
Фамилия: Иванов
Имя: Павел
Курс: 3-й
Место рождения: Петербург
Дата рождения: 22.11.1888
```

Конкретная структура-объект stud типа **struct** student получает значение элементов при инициализации. Затем с помощью scanf( ) изменяется элемент stud.year (Иванов Павел перешел уже на 3-й курс!). Содержимое структуры выводится на дисплей, причем для доступа к элементам вложенных структур используются уточненные имена с нужным количеством «точек».

Массивы структур. Определяются массивы структур так же, как и массивы других типов данных. Естественное отличие – служебное слово **struct** в названии структурного типа (если обозначение типа не введено с помощью **typedef**). Для введенных выше структурных типов можно определить:

```
struct goods list [MAX_GOODS];
complex set [80];
```

Эти предложения определяют list как массив, состоящий из MAX_GOODS-элементов – структур типа goods, a set – массив структур типа complex, состоящий из 80 элементов. (Идентификатор MAX_GOODS должен быть именем константы.)

Имена list и set не являются именами структур, это имена массивов, элементами которых являются структуры. list[0] есть структура типа goods, list[1] – вторая структура типа goods из массива list[ ] и т. д.

Для доступа к компонентам структур, входящих в массив структур, используются уточненные имена с индексированием первого имени (имени массива). Например:

- set[4].real – первый компонент (с именем real) типа **double** структуры типа complex, входящей в качестве пятого (от начала) элемента в массив структур set[ ];
- list[0].price – второй компонент (с именем price) типа **long** структуры типа goods, входящей в качестве первого элемента (с нулевым индексом) в массив структур list[ ].

На рис. 6.2 условно изображена схема размещения в памяти массива list[ ]. Его элементы (как и элементы любого другого массива) размещаются в основной памяти подряд в соответствии с возрастанием индекса.

Определение массива структур:

struct goods list [MAX_GOODS];

list[0].name	list[0].price	list[0].percent	list[0].vol	list[0].date	list[0]
char*	**long**	**float**	**int**	**char[9]**	
list[1].name	list[1].price	list[1].percent	list[1].vol	list[1].date	list[1]
char*	**long**	**float**	**int**	**char[9]**	
...	...	...	...	...	
list[99].name	list[0].price	list[0].percent	list[0].vol	list[0].date	list[99]
char*	**long**	**float**	**int**	**char[9]**	

Рис. 6.2. *Массив структур (MAX_GOODS==100)*

Еще раз обратите внимание на индексирование в уточненном имени при обращении к компонентам структур, принадлежащих массиву. Индекс записывается непосредственно после имени массива структур. Тем самым из массива выделяется нужная структура, а уже с помощью точки и последующего имени идентифицируется соответствующий компонент структуры. Будет ошибкой поместить индекс после всего уточненного имени. Например, запись:

```
list.percent[8]    /* Ошибка !   */
```

– ошибочное имя; percent – переменная типа **float**, а не массив.

Как и массивы других типов, массив структур при определении может быть инициализирован. Инициализатор массива структур должен содержать в фигурных скобках список начальных значений структур массива. В свою очередь, каждое начальное значение для структуры – это список значений ее компонентов (также в фигурных скобках).

В следующей программе вводится и инициализируется массив структур, каждая из которых описывает точечную массу (материальную точку). Для определенной таким образом системы материальных точек вычисляется центр масс (координаты x_c, y_c, z_c центра масс).

Если $m_1, m_2, ..., m_n$ – массы материальных точек; x_i, y_i, z_i – координаты отдельной точки (i=1, n), то имеем:

$$m = \sum_{i=1}^{n} m_i$$ – общая масса системы точек.

Координаты центра масс:

$$x_c = \frac{\sum_{i=1}^{n} x_i m_i}{m}; \quad y_c = \frac{\sum_{i=1}^{n} y_i m_i}{m}; \quad z_c = \frac{\sum_{i=1}^{n} z_i m_i}{m}.$$

Текст программы с массивом структур:

```
#include <stdio.h>
/* Определение структурного типа: */
struct particle {
                double mass;
                float coord [3];
               };
void main ( )
{   /* Определение массива структур: */
   struct particle mass_point[ ]={
                     20.0, {2.0, 4.0, 6.0},
                     40.0, {6.0, -2.0, 8.0},
                     10.0, {1.0, 3.0, 2.0}
                                };
   int N; /* N - число точек */
   /* Структура для формирования результатов: *
   struct particle center= { /* центр масс */
                     0.0, {0.0, 0.0, 0.0}
                          };
```

```
   int i;
   N=sizeof(mass_point) / sizeof(mass_point[0]);
   for (i=0; i<N; i++)
    {
     center.mass+=mass_point[i].mass;
     center.coord[0] += mass_point[i].coord[0] * mass_point[i].mass;
     center.coord[1] += mass_point[i].coord[1] * mass_point[i].mass;
     center.coord[2] += mass_point[i].coord[2] * mass_point[i].mass;
    }
   printf("\n Координаты центра масс:");
   for (i=0; i<3; i++)
     {
       center.coord[i]/=center.mass;
       printf("\n Координата %d: %f", (i+1), center. coord [i]);
     }
}
```

Результат выполнения программы:

```
Координаты центра масс:
Координата 1: 4.142857
Координата 2: 0.428571
Координата 3: 6.571429
```

Указатели на структуры. Эти указатели определяются, как и указатели на данные других типов. Пример:

```
struct goods *p_goods;
struct studest *p_stu, *p_stu2;
```

Указатели на структуры могут вводиться и для безымянных структурных типов:

```
struct { /* Безымянный структурный тип */
      char processor [10]; /* Тип процессора */
      int frequency; /* Частота в ГГц */
      int memory; /* Память в Гбайт */
      int disk; /* Емкость диска в Гбайт */
      } *point_IBM, *point_2;
```

Если название структурного типа введено с помощью **typedef**, то при определении указателей название этого типа используется без предшествующего служебного слова **struct**:

```
complex *cc, *ss, comp;
```

При определении указателя на структуру он может быть инициализирован. Наиболее корректно в качестве инициализирующего значения применять адрес структурного объекта того же типа, что и тип определяемого указателя:

```
struct particle
      {
         double mass;
         float coord [3];
      } dot [3], point, *pinega;
/* Инициализация указателей: */
struct particle *p_d=&dot [1],
*pinta=&point;
```

Значение указателя на структуру может быть определено и с помощью обычного присваивания:

```
pinega=&dot [0];
```

Указатели как средство доступа к компонентам структур. Указатель на структуру, настроенный на конкретную структуру (конкретный объект) того же типа, обеспечивает доступ к ее элементам двумя способами:

(* *указатель_на_структуру*) **.** *имя_элемента*

или

указатель_на_структуру **->** *имя_элемента*

Первый способ традиционный. Он основан на обратимости операций разыменования '*' и получения адреса '&'. Обозначив знак равенства последовательностью '==', можно таким образом формально напомнить эти правила на конкретном примере:
если

```
pinega = = &dot [0],
```

то

```
*pinega = = dot[0].
```

Доступ к элементам структуры dot[0] с помощью разыменования адресующего его указателя pinega можно в соответствии с приведенными соотношениями реализовать с помощью таких конструкций:

```
(*pinega).mass = = dot[0].mass
(*pinega).coord [0] = = dot[0].coord[0]
(*pinega).coord [1] = = dot[0].coord[1]
(*pinega).coord [2] = = dot[0].coord[2]
```

Важным является наличие скобок, ограничивающих действие операции разыменования (*pinega). Скобки необходимы, так как бинарная операция «точка» имеет более высокий ранг, чем унарная операция разыменования (см. табл. 1.4 приоритетов операций в главе 1).

Присвоить элементам структуры dot[0] значения можно, например, с помощью таких операторов присваивания:

```
(*pinega).mass = 33.3;
(*pinega).coord[1] = 12.3;
```

Второй способ доступа к элементам структуры с помощью «настроенного» на нее указателя предусматривает применение специальной операции «стрелка» (**->**). Операция «стрелка» имеет самый высший ранг (см. табл. 1.4) наряду со скобками и операцией «точка». Операция «стрелка» обеспечивает доступ к элементу структуры через адресующий ее указатель того же структурного типа. Формат применения операции в простейшем случае:

указатель_на_структуру **->** *имя_элемента*

Операция «стрелка» двуместная. Применяется для доступа к элементу (компоненту), задаваемому правым операндом, той структуры, которую адресует левый операнд. В качестве левого операнда должен быть указатель на структуру: в качестве правого – обозначение (имя) компонента этой структуры.

Операция «стрелка» (**->**) иногда называется операцией косвенного выбора компонента (элемента) структурированного объекта, адресуемого указателем.

Примеры:

```
pinega -> mass     эквивалентно (*pinega).mass
pinega -> coord[0] эквивалентно (*pinega).coord[0]
```

Если в программе (см. выше) определена структура point типа **struct** particle и на нее настроен указатель:

```
struct particle *pinta = &point;
```

то будут справедливы следующие равенства:

```
pinta -> mass == (*pinta).mass
pinta -> coord == (*pinta).coord
```

Изменить значения элементов структуры point в этом случае можно, например, такими операторами:

```
pinta -> mass = 18.4;
for (i=0; i<3; i++)
pinta -> coord[i] = 0.1*i;
```

Тип результата выражения с операцией «стрелка» (**->**) совпадает с типом правого операнда, то есть того элемента структуры, на который «нацелена» стрелка.

Операции над указателями на структуры. Эти операции не отличаются от операций над другими указателями на данные. (Исключение составляет операция «стрелка» (**->**), но ее мы уже рассмотрели). Если присвоить указателю на структуру конкретного структурного типа значение адреса одного из элементов массива структур того же типа, то, изменяя значение указателя (например, с помощью операций ++ или ––), можно равномерно «перемещаться» по массиву структур.

В качестве иллюстрации рассмотрим следующую задачу. Вычислить сумму заданного количества комплексных чисел, представленных в программе массивом array[ ] структур.

```
#include <stdio.h>
void main ( )
{
struct complex
   { /* Определение структурного типа complex*/
      float x; /* Вещественная часть */
      float y; /* Мнимая часть */
   } array [ ] = {1.0, 2.0, 3.0, -4.0, -5.0, -6.0, -7.0, -8.0};
 struct complex summa = {0.0, 0.0};
```

```
 struct complex * point = & array [0];
 int k, i;
 k = sizeof (array) / sizeof (array [0]);
 for (i=0; i<k; i++)
    {
       summa.x += point -> x;
       summa.y += point -> y;
       point++;
    }
printf(" \n Сумма: real=%f,\t imag=%f", summa.x, summa.y);
}
```

Результат выполнения программы:

```
Сумма: real= -8.000000,   imag=-16.000000
```

В программе введен тип **struct** complex. Определены: массив структур array[ ], структура summa, где формируется результат, и указатель point, в начале программы настроенный на первый (нулевой) элемент массива структур, а затем в цикле «пробегающий» по всему массиву. Доступ к структуре summa реализован с помощью уточненных имен (операция «точка»). Доступ к элементам структур, входящих в массив, осуществляется через указатель point и с помощью операции «стрелка» **(->)**.

Указатели на структуры как компоненты структур. В соответствии с синтаксисом языка компонентами структур могут быть данные любых типов, за исключением структур того же типа, что и определяемый структурный тип. Например, элементом структуры, как уже говорилось, может быть структура, тип которой уже определен. Обычна конструкция:

```
struct mix {int N; double * d;};
struct hole {
              struct mix exit;
              float b;
             };
```

Здесь ранее определенная структура типа **struct** mix является элементом структуры типа **struct** hole.

В то же время элементом структуры может быть указатель на структуру того же типа, что и определяемый структурный тип:

```
struct element {
     /* Структурный тип «химический элемент» */
     int number /* Порядковый номер *
     double mass; /* Атомный вес */
     char name[16] /* Название элемента */
     struct element * next;
};
```

В структурном типе «химический элемент» есть компонент next – указатель на структуру того же типа. С его помощью можно формировать списки (цепочки) интересующих нас элементов. Например, можно связать все элементы одной группы периодической таблицы химических элементов либо представить в виде единого списка всю таблицу Д. И. Менделеева.

При определении структурных типов может потребоваться организация взаимных перекрестных связей между структурами двух или более разных типов. В этом случае невозможно выполнить требование об определенности всех связываемых структурных типов. Выходом из тупиковой ситуации служит применение указателей на структуры. Например:

```
struct part { /* Структурный тип */
              double modul;
              struct cell * element_cell;
              struct part * element_part;
             };
struct cell { /* Структурный тип */
              long summa;
              struct cell * one;
              struct part * two;
             }
```

В данном примере для организации перекрестных ссылок в структурах типа **struct** part использованы в качестве элементов указатели на объекты типа **struct** cell. В структуры типа **struct** cell входят указатели на структуры типа **struct** part.

6.3. Структуры и функции

Для «взаимоотношения» структур и функций имеются две основные возможности: структура может быть возвращаемым функцией значением и структура может использоваться в параметрах функ-

ции. Кроме того, в обоих случаях могут использоваться указатели на объекты структурных типов. Итак, рассмотрим все перечисленные варианты.

```
/* Определение структурного типа: */
struct person {char * name; int age; };
```

Функция может возвращать структуру как результат:

```
/* Прототип функции: */
struct person f (int N)
```

Функция может возвращать указатель на структуру:

```
/* Прототип функции: */
struct person * ff (void);
```

Параметром функции может быть структура:

```
/* Прототип функции: */
void fff (struct person str);
```

Параметром функции может быть указатель на объект структурного типа:

```
/* Прототип функции: */
void ffff (struct person * pst);
```

При вызове функции fff( ) выделяется память для параметра, то есть для вспомогательного объекта типа **struct** person. В этот объект переносится значение аргумента, заменяющего параметр – структуру str. Далее выполняются действия, предусмотренные операторами тела функции fff( ). Эти действия не могут изменять структуру, использованную в качестве аргумента.

В случае, когда параметром служит указатель на объект структурного типа, действиями внутри тела функции ffff( ) можно изменить ту структуру вызывающей функции, которая адресуется аргументом pst.

Имитация абстрактных типов данных. При решении конкретных прикладных задач бывает удобным выделить набор типов данных, наиболее полно соответствующих понятиям и объектам предметной области. Такие типы данных обычно отсутствуют в языке програм-

мирования, но их можно сконструировать и ввести как производные типы. Одновременно с такими специализированными данными (точнее, с производными типами для их представления) для каждого типа вводится набор операций, удобный для обработки этих данных. Тип данных и набор операций над объектами этого типа совместно определяют абстрактный тип данных, ориентированный на представление понятий предметной области решаемой задачи. В языке Си программист имеет возможность вводить абстрактные типы данных с помощью структур, отображающих собственно данные, и функций, определяющих операции над данными.

В качестве примера введем абстрактный тип данных для рациональных дробей. Выше в §6.1 введен и поименован с помощью **typedef** как fraction структурный тип для представления рациональных дробей.

Определим функции, реализующие операции над рациональными дробями:

- **input( )** – ввод дроби;
- **out( )** – вывод дроби на дисплей;
- **add( )** – сложение;
- **sub( )** – вычитание;
- **mult( )** – умножение;
- **divide( )** – деление.

Этот набор операций можно легко расширить, введя, например, функции для приведения дробей к одному знаменателю, для сокращения дроби, для выделения целой части неправильной дроби и т. д. Но для демонстрации особенностей применения структур в функциях при определении абстрактного типа данных перечисленного набора операций вполне достаточно.

Оформим определение структурного типа «рациональная дробь» и набор прототипов перечисленных выше функций для операций над дробями в виде следующего файла:

```
/* Файл fract.h - абстрактный тип "рациональная дробь"  */
typedef struct rational_fraction
   {
    int numerator; /* Числитель */
    int denominator; /* Знаменатель */
   } fraction; /* Обозначение структурного типа */

/* Прототипы функций для работы с дробями: */
void input (fraction * pd);
```

```
void out (fraction dr);
fraction add (fraction dr1, fraction dr2);
void sub (fraction dr1, fraction dr2, fraction * pdr);
fraction * mult (fraction dr1, fraction dr2);
fraction divide (fraction *pd1, fraction *pd2);
```

Обратите внимание, что применение **typedef** позволило упростить название структурного типа. Вместо конструкции **struct** rational_fraction в прототипах функций и далее в программе используется более короткое обозначение fraction.

Определения функций, реализующих перечисленные операции над рациональными дробями, объединим в один файл следующего вида:

```
/*Файл fract.c - определения функций для работы с дробями    */
#include <stdio.h>
#include <stdlib.h> /* Для exit() и malloc() */
void input (fraction * pd) /* Ввод дроби */
{
   int N;
   printf("\n Числитель:");
   scanf("%d", pd -> numerator);
   printf("Знаменатель:");
   scanf("%d", &N);
   if (N == 0)
      {
         printf("\n Ошибка! Нулевой знаменатель!");
         exit (0); /* Завершение программы */
      }
   pd -> denominator=N;
}
/* Изобразить дробь на экране: */
void out (fraction dr)
{
   printf("Рациональная дробь:");
   printf("%d/%d", dr.numerator, dr.denominator);
}
/* Сложение дробей: */
fraction add (fraction dr1, fraction dr2)
{
   fraction dr;
   dr.numerator = dr1.numerator * dr2.denominator
                + dr1.denominator * dr2.numerator;
```

```
   dr.denominator = dr1.denominator * dr2.denominator;
   return dr;
}
/* Вычитание дробей: */
void sub (fraction dr1, fraction dr2, fraction * pdr)
{
   pdr -> numerator=dr1.numerator * dr2.numerator
            - dr2.numerator * dr1.denominator;
   pdr -> denominator=dr1.denominator * dr2.denominator;
}
/* Умножение дробей: */
fraction * mult (fraction dr1, fraction dr2)
{
   fraction * mul;
   mul=(fraction *) malloc (sizeof (fraction));
   mul -> numerator=dr1.numerator * dr2.numerator;
   mul -> denominator=dr1.denominator * dr2.denominator;
   return mul;
}
/* Деление дробей: */
fraction divide (fraction * pd1, fraction * pd2)
  {
    fraction d;
    d.numerator = pd1 -> numerator * pd2 -> denominator;
    d.denominator = pd1 -> denominator * pd2 -> numerator;
    return d;
}
```

Приведенный набор функций демонстрирует все возможные сочетания возвращаемых значений и параметров функций при работе со структурами. Обратите внимание, что функции add( ), divide( ) возвращают структуру типа fraction, которую нужно «разместить» в соответствующей структуре вызывающей программы. Функция sub( ) предусматривает передачу результата через параметр-указатель fraction * pdr. Функция mult( ) формирует динамический объект типа fraction и возвращает его адрес. В основной программе после обращения к mult( ) корректно будет освободить динамическую память с помощью функции free( ).

Следующая программа демонстрирует работу с рациональными дробями с помощью введенных функций:

```
#include "fract.h" /* Файл прототипов */
#include "fract.c" /* Файл определений функций */
```

```
void main ( )
{
   fraction a, b, c; /* Определили три дроби-структуры */
   fraction *p; /* Указатель на дробь */
   printf("\n Введите дроби:");
   input(&a);
   input(&b);
   c = add(a, b);        /* Сложение дробей */
   out(c);               /* Вывод дроби */
   p = mult(a, b);       /* Умножение дробей */
   out(*p);              /* Вывод дроби, адресуемой указателем */
   free(p);              /* Освободить память */
   c = divide(&a, &b);   /* Деление дробей */
   out (c);              /* Вывод дроби */
}
```

Результаты выполнения программы:

```
Введите дроби:
Числитель:        5     <Enter>
Знаменатель:      4     <Enter>
Числитель:        2     <Enter>
Знаменатель:      3     <Enter>
Рациональная дробь:       23/12
Рациональная дробь:       10/12
Рациональная дробь:       15/8
```

Обратите внимание на необходимость явного освобождения памяти после выполнения функции mult( ), внутри тела которой для результата (для структуры типа fraction) явно выделяется память, и указатель на этот участок памяти передается в вызывающую программу. После печати результата out(*p) библиотечная функция free(p) освобождает память.

6.4. Динамические информационные структуры

Статическое и динамическое представление данных. В языках программирования типы данных и определения переменных (как объектов заданных типов) нужны для того, чтобы определить требования к памяти для каждого из этих объектов и фиксировать диапазон (пределы) значений, которые могут принимать

эти объекты (переменные). Для базовых типов (**int**, **long**, **double** и т. д.) требования к памяти и диапазоны возможных значений определяются реализацией языка (компилятора). Из базовых типов формируются производные, такие как массивы и структурные типы. Для производных типов требования к памяти заложены в их определениях. Таким образом, определив массив или структурный тип, программист зафиксировал требования к памяти в самом определении. Например, на основании определения **double** array[18] в памяти для массива array[ ] будет выделено не менее 18* **sizeof** (**double**) байт.

С помощью определения

```
struct mixture
      {
         int ii;
         long ll;
         char cc [8];
      };
```

не только задается состав структурного типа с названием **struct** mixture, но и утверждается, что каждый объект структурного типа **struct** mixture потребует для размещения не менее

sizeof (**int**) + **sizeof** (**long**) + 8* **sizeof** (**char**)

байт. Точный объем позволяет определить операция

sizeof (**struct** mixture).

Вводимые таким образом объекты позволяют представить только статические данные. Однако во многих задачах требуется использовать более сложные данные, представление которых (конфигурация, размеры, состав) может изменяться в процессе выполнения программы. О таких изменяемых данных говорят, используя принятый в информатике термин *динамические информационные структуры.*

Односвязный список. Наиболее простая динамическая информационная структура – это односвязный список, элементами (звеньями) которого служат объекты, например такого структурного типа:

struct *имя_структурного_типа*
{
    элементы_структуры; /* данные */
    struct *имя_структурного_типа* * *указатель*;
};

В каждую структуру такого типа входит указатель на объект того же типа, что и определяемая структура.

Чтобы продемонстрировать некоторые особенности обработки простых динамических информационных структур, разработаем программу, в которой определим структурный тип для представления звеньев односвязного списка и решим такую простейшую задачу: «Ввести с клавиатуры произвольное количество структур, объединяя их в односвязный список, а затем вывести на экран дисплея содержимое введенного списка в порядке формирования его звеньев».

Анализ динамических информационных структур удобнее всего выполнять с помощью их графического изображения. На рис. 6.3 приведены схема односвязного списка и обозначения, которые будут использованы в программе. Для работы со списком понадобятся три указателя: beg – на начало списка; end – на последний элемент, уже включенный в список; rex – указатель для «перебора» элементов списка от его начала.

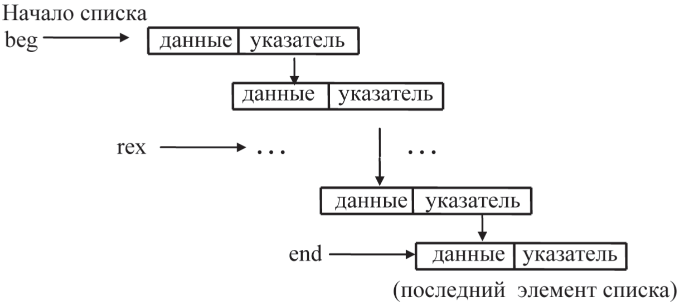


Рис. 6.3. *Односвязный динамический список*

Следующая программа решает сформулированную задачу.

```
#include <stdlib.h>
#include <stdio.h>
/* Определение структурного типа "Звено списка":*/
struct cell {
             char sign [10];
             int weight;
             struct cell * pc;
            };
```

```
void main()
{
   /* Указатель для перебора звеньев списка: */
   struct cell * rex;
   struct cell * beg=NULL; /* Начало списка */
   struct cell * end=NULL; /* Конец списка */
   printf("\nВведите значения структур:\n");
   /* Цикл ввода и формирования списка */
   do
   {    /* Выделить память для очередного звена списка: */
    rex=(struct cell *)malloc(sizeof(struct cell));
    /* Ввести значения элементов звена: */
    printf("sign=");
    scanf("%s",& rex->sign);
    printf("weight=");
    scanf("%d",& rex->weight);
    if (rex->weight == 0)
    {
       free(rex);
       break; /* Выход из цикла ввода списка */
    }
    /* Включить звено в список: */
    if (beg==NULL && end==NULL)/* Список пуст*/
         beg=rex;
    else    /* Включить звено в уже существующий список */
         end->pc=rex;
    end=rex;
    end->pc=NULL;
   }
   while(1); /* Конец ввода списка */
   /* Напечатать список */
   printf("\nСодержимое списка:");
   rex=beg;
   while (rex!=NULL)
   {
     printf("\nsign=%s\tweight=%d",rex->sign,rex->weight);
     rex=rex->pc;
   }
}
```

Пример выполнения программы:

```
Введите данные о структурах:
Sign=sigma    <Enter>
weight=16     <Enter>
```

```
Sign=omega    <Enter>
weight=44     <Enter>
Sign=alfa     <Enter>
weight=0      <Enter>

Содержимое  списка:
Sign=sigma   weight=16
Sign=omega   weight=44
```

В программе ввод данных, то есть заполнение односвязного списка структур, выполняется в цикле. Условием окончания цикла служит нулевое значение, введенное для элемента **int** weight, очередной структуры.

Формирование списка структур происходит динамически. Указатели beg, end инициализированы нулевыми значениями – вначале список пуст.

Обратите внимание на преобразование типов (**struct** cell *) при выделении памяти для структуры типа **struct** cell. Функция **malloc( )** независимо от типа параметров всегда возвращает указатель типа **void** *, а слева от знака присваивания находится указатель rex типа **struct** cell *.

Используется явное приведение типов, хотя это не обязательно – тип **void** * совместим по операции присваивания с указателем на объект любого типа.

Остальные особенности программы поясняются комментариями в ее тексте.

Рекурсия при обработке списка. Даже такая простая структура, как динамический односвязный список, представляет собой конструкцию, которую удобно обрабатывать с помощью рекурсивных процедур. Рассмотрим ту же задачу об односвязном списке, но оформим формирование списка и вывод списка на экран дисплея в виде рекурсивных функций. Сделаем это с целью показать на этом простом примере особенности рекурсивной обработки динамических информационных структур.

В каждом звене списка содержатся полезная информация (данные) и ссылка (адрес) на следующее звено списка. Если такая ссылка нулевая, то список пройден до конца. Для начала просмотра списка нужно знать только адрес его первого элемента.

Рассмотрим, какие действия должны выполнять функция рекурсивного формирования списка и функция его рекурсивного просмотра с выводом данных о звеньях списка на экран дисплея.

Функция рекурсивного формирования (заполнения) списка ниже в программе имеет прототип:

```
struct cell * input (void);
```

Она возвращает указатель на сформированный и заполненный данными с клавиатуры динамический список. Предполагается, что при каждом вызове этой функции формируется новый список. Функция заканчивает работу, если для очередного звена списка введено нулевое значение переменной weight. В противном случае заполненная с клавиатуры структура подключается к списку, а ее элементу **struct** cell * pc присваивается значение, которое вернет функция input( ), рекурсивно вызванная из тела функции. После этого функция возвращает адрес очередного звена списка, то есть значение указателя **struct** cell * pc.

Прежде чем рассматривать текст функции input( ), помещенный в приведенной ниже программе, еще раз сформулируем использованные в ней конструктивные решения.

1. При каждом обращении к этой функции в основной памяти формируется новый список, указатель на начало которого возвращает функция.
2. Если для очередного звена списка вводится нулевое значение элемента weight, то звено не подключается к списку, процесс формирования списка заканчивается. Такой набор данных, введенных для элемента очередного звена, считается терминальным.
3. Если нулевое значение элемента weight введено для первого звена списка, то функция input( ) возвращает значение **NULL**.
4. Список определяется рекурсивно как первое (головное) звено, за которым следует присоединяемый к нему список.

Текст функции input( ) на языке Си учитывает перечисленные конструктивные решения. Для структуры типа **struct** cell выделяется память. После того как пользователь введет значения данных, выполняется их анализ. В случае терминальных значений библиотечная функция **free(p)** освобождает память от ненужного звена списка, и выполняется оператор **return NULL**;. В противном случае элементу связи (указатель **struct** cell * p) присваивается результат рекурсивного вызова функции input( ). Далее все повторяется для нового экземпляра этой функции.

Функция рекурсивного просмотра и печати списка имеет такой прототип:

```
void output (struct cell * p);
```

Исходными данными для ее работы служит адрес начала списка (или адрес любого его звена), передаваемый как значение указателя – параметра **struct** cell * p. Если параметр имеет значение **NULL** – список исчерпан, исполнение функции прекращается. В противном случае выводятся на экран значения элементов той структуры, которую адресует параметр, и функция output( ) рекурсивно вызывается с параметром p **->** pc. Тем самым выполняется продвижение к следующему элементу списка. Конец известен – функция печатает «Список исчерпан!» и больше не вызывает сама себя, а завершает работу.

Текст программы с рекурсивными функциями:

```
/* Определение структурного типа "Звено списка":*/
struct cell
{
 char sign[10];
 int weight;
 struct cell * pc;
};
#include <stdlib.h>
#include <stdio.h>
/* Функция ввода и формирования списка: */
struct cell * input(void)
{
   struct cell * p;
   p=(struct cell *)malloc(sizeof(struct cell));
   printf("sign=");
   scanf("%s",& p->sign);
   printf("weight=");
   scanf("%d",& p->weight);
   if (p -> weight == 0)
     {
      free (p);
      return NULL;
     }
   p -> pc = input(); /* Рекурсивный вызов */
   return p;
}
```

```
/* Функция "печати" списка: */
void output(struct cell *p)
{
   if (p == NULL)
   {
    printf("\nСписок исчерпан!");
    return;
   }
   printf("\nsign=%s\tweight=%d",p->sign,p->weight);
   output(p -> pc); /* Рекурсивный вызов */
}
void main()
{
   struct cell * beg=NULL; /* Начало списка */
   printf("\nВведите данные структур:\n");
   beg=input();   /* Ввод списка. */
   /*Напечатать список: */
   printf("\nСодержимое списка:");
   output(beg);
}
```

Возможный результат выполнения программы:

```
Введите данные структур:
Sign=Zoro      <Enter>
weight=1938    <Enter>
Sign=Zodiac    <Enter>
weight=1812    <Enter>
Sign=0         <Enter>
weight=0       <Enter>

Содержимое списка:
Sign=Zoro      weight=1938
Sign=Zodiac    weight=1812
Список исчерпан!
```

6.5. Объединения и битовые поля

Объединения. Со структурами в «близком родстве» находятся объединения, которые вводятся (описываются, определяются) с помощью служебного слова **union**.

Объединение можно рассматривать как структуру, все элементы которой имеют нулевое смещение от ее начала. При таком размеще-

нии разные элементы занимают в памяти один и тот же участок. Тем самым объединения обеспечивают возможность доступа к одному и тому же участку памяти с помощью переменных (и/или массивов и структур) разного типа. Необходимость в такой возможности возникает, например, при выделении из внутреннего представления целого числа его отдельных байтов. Для иллюстрации сказанного введем такое объединение:

```
union {
        char  hh[2];
        int    ii;
      } CC;
```

Здесь:

- **union** – служебное слово, вводящее тип данных «объединение» или объединяющий тип данных;
- CC – имя конкретного объединения;
- символьный массив **char** hh[2] и целая переменная **int** ii – элементы (компоненты) объединения.

Схема размещения объединения CC в памяти ЭВМ приведена на рис. 6.4.

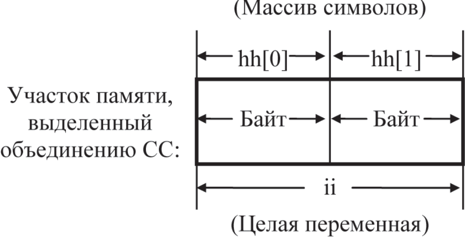


Рис. 6.4. *Схема размещения объединения в памяти*

Для обращения к элементу объединения используются те же конструкции, что и для обращения к элементу структуры:

имя_объединения **.** *имя_элемента*
(* *указатель_на_объединение* **)** **.** *имя_элемента*
указатель_на_объединение **–>** *имя_элемента*

Смысловое отличие объединения от структуры состоит в том, что записать информацию в объединение можно с помощью одного из его элементов, а выбрать данные из того же участка памяти можно с помощью другого элемента того же объединения. Например, оператор

```
CC.ii = 15;
```

записывает значение 15 в объединение, а с помощью конструкций CC.hh[0], CC.hh[1] можно получить отдельные байты внутреннего представления целого числа 15.

Как и данные других типов, объединение – это конкретный объект, которому выделено место в памяти. Размеры участка памяти, выделяемого для объединения, должны быть достаточны для размещения самого большого элемента объединения. В нашем примере элементы **int** ii и **char** hh[2] имеют одинаковую длину, но это не обязательно. Основное свойство объединения состоит в том, что все его элементы размещаются от начала одного и того же участка памяти. А размеры участка памяти, отводимого для объединения, определяются размером самого большого из элементов. Например, для объединения

```
union
    {
      double   dd;
      float   aa;
      int   jj;
    }  uu;
```

размеры объекта-объединения uu равны размерам самого большого из элементов, то есть:

```
sizeof(uu) == sizeof(double)
```

Объединяющий тип. В рассмотренных примерах определены объединения, но явно не обозначены объединяющие типы. Именованный объединяющий тип вводится с помощью определения такого вида:

union *имя_объединяющего_типа*
{
определения_элементов
};

где **union** – спецификатор типа; *имя_объединяющего_типа* – выбираемый программистом идентификатор; *определение_элементов* – совокупность описаний объектов, каждый из которых служит прототипом одного из элементов объединений вводимого объединяющего типа.

Конструкция **union** *имя_объединяющего_типа* играет роль имени типа, то есть с ее помощью можно вводить объединения-объекты.

Как и для структурных типов, с помощью **typedef** можно вводить обозначения объединяющих типов, не требующие применения **union**:

typedef union *имя_объединяющего_типа*
{
определения_элементов
} *обозначение_объединяющего_типа ;*

Пример:

```
typedef union uni
      {
        double  dou;
        int  in[4];
        char  ch[8];
      } uni_name;
```

Имея такое определение, можно вводить конкретные объединения двумя способами:

```
union uni snow, ray;
uni_name yet, zz4;
```

Объединения не относятся ни к скалярным данным, ни к данным агрегирующих типов.

Битовые поля. Битовое поле может быть только элементом структуры или объединения и вне объектов этих типов не встречается. Битовые поля не имеют адресов и не могут объединяться в массивы. Назначение битовых полей – обеспечить удобный доступ к отдельным битам данных. Кроме того, с помощью битовых полей можно формировать объекты с длиной внутреннего представления, не кратной байту. Это позволяет плотно «упаковывать» информацию, что экономит память.

Описание структуры с битовыми полями должно иметь такой формат:

struct { *тип_1 имя_поля_1* **:** *ширина_поля_1*;
тип_2 имя_поля_2 **:** *ширина_поля_2*;
...
} *имя_структуры* ;

где *тип_i* – тип поля, который может быть только **int**, возможно, со спецификатором **unsigned** или **signed**; *ширина_поля* – целое неотрицательное десятичное число, значение которого зависит от реализации компилятора.

Разрешается поле без имени (для чего указываются только двоеточие и ширина), с помощью которого в структуру вводятся неиспользуемые биты (промежуток между значимыми полями).

Для обращения к полям используются те же конструкции, что и для обращения к обычным элементам структур:

имя_структуры **.** *имя_поля_i*
(указатель_на_структуру)* **.** *имя_поля_i*
указатель_на_структуру –> *имя_поля_i*

Например, для структуры хх с битовыми полями:

```
struct {
        int a:10;
        int b:14;
      } xx;
```

правомочны такие операторы:

```
xx.a=1;   xx.b=48;     или      xx.a=xx.b=0;
```

От реализации зависит порядок размещения в памяти полей одной структуры. Поля могут размещаться как слева направо, так и справа налево. Для архитектуры Intel поля, которые размещены в начале описания структуры, имеют младшие адреса. Таким образом, для примера:

```
struct {
       int x:5;
       int y:3;  } hh;
hh.x=4;  hh.y=2;
```

в одном байте формируется последовательность битов, приведенная на рис. 6.5.

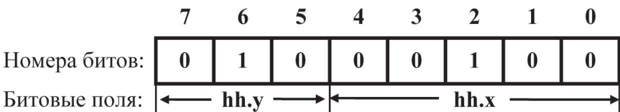


Рис. 6.5. *Размещение битовых полей в байте*

Вместо служебного слова **struct** может употребляться **union**. В этом случае определяется объединение с битовыми полями.

В качестве иллюстрации возможностей объединений и структур с битовыми полями рассмотрим следующую программу. В ней вводятся значения двух целых переменных m и n, и остатки от их деления на 16 заносятся соответственно в четыре младших и четыре старших разряда одного байта. Таким образом выполняется некоторая кодировка введенных значений переменных m и n. Затем печатается изображение содержимого сформированного байта. Особенности программы объяснены в комментариях и поясняются результатами выполнения. Обратите внимание на использование объединений. В функции cod( ) запись данных производится в элементы (битовые поля) структуры hh, входящей в объединение un, а результат выбирается из того же объединения un в виде одного байта. В функции binar( ) происходит обратное преобразование – в нее как значение параметра передается байт, содержимое которого побитово «расшифровывается» за счет обращения к отдельным полям структуры byte, входящей в объединение cod.

Текст программы:

```
/* Структуры, объединения и битовые поля */
#include <stdio.h>
void main (void)
{
   unsigned char k;
   int m,n;
   void binar(unsigned char);   /* Прототип функции */
   unsigned char cod(int,int);  /* Прототип функции */
   printf("\n m=");
   scanf("%d",&m);
   printf(" n=");
   scanf("%d",&n);
   k=cod(m,n);
   printf(" cod=%u",k);
```

```
   binar(k);
}
/* Упаковка в один байт остатков от деления на 16 двух целых чисел */
unsigned char cod(int a, int b)
{
   union
         {
            unsigned char z;
            struct
               {
                 unsigned int x :4; /* Младшие биты */
                 unsigned int y :4; /* Старшие биты */
               } hh;
         } un;
   un.hh.x=a%16;
   un.hh.y=b%16;
   return un.z;
}
/* binar - двоичное представление байта */
void binar(unsigned char ch)
{
   union
        {
          unsigned char ss;
          struct
             {
               unsigned a0 :1; unsigned a1 :1;
               unsigned a2 :1; unsigned a3 :1;
               unsigned a4 :1; unsigned a5 :1;
               unsigned a6 :1; unsigned a7 :1;
             } byte;
        } cod;
   cod.ss=ch;
   printf("\n Номера битов:         "
          "7    6    5    4    3    2    1    0");
   printf("\n Значения битов:      %d    %d    "
          "%d    %d    " %d    %d    %d    %d",
          cod.byte.a7, cod.byte.a6, cod.byte.a5,
          cod.byte.a4, cod.byte.a3, cod.byte.a2,
          cod.byte.a1, cod.byte.a0);
}
```

Результаты выполнения программы на современном ПК:

- первый вариант:

```
m=1     <Enter>
n=3     <Enter>
cod=49
Номера  битов:       7     6     5     4     3     2     1     0
Значения  битов:     0     0     1     1     0     0     0     1
```

- второй вариант:

```
m=0     <Enter>
n=1     <Enter>
cod=16

Номера  битов:       7     6     5     4     3     2     1     0
Значения  битов:     0     0     0     1     0     0     0     0
```

Контрольные вопросы

1. Дайте определение структуры.
2. Что является спецификатором структурного типа?
3. Вспомните формат определения конкретных структур.
4. Назовите все схемы определения структур.
5. Каким образом производится инициализация структур?
6. Разрешается ли присваивание структур?
7. Каким образом можно сравнить структуры?
8. Как осуществляется доступ к элементам структуры?
9. Что такое «уточненное имя»?
10. Поясните смысл операции «точка».
11. Можно ли операцию получения адреса & применить к уточненному имени?
12. Допускается ли вложение структур?
13. Как определить массив структур?
14. Каким образом осуществляется доступ к компонентам структур, входящим в массив структур?
15. Могут ли вводиться указатели на структуры для безымянных структурных типов?
16. Поясните способы доступа к элементам структуры с помощью указателя.
17. Какие операции допустимы над указателями на структуры?

18. Объясните применение в качестве элемента структуры указателя на структуру того же типа, что и определяемый структурный тип.
19. Перечислите варианты использования структур при вызове функции.
20. Каким образом в языке Си можно вводить абстрактные типы данных?
21. Поясните способ создания динамических информационных структур.
22. Дайте определение объединения.
23. Приведите определение для именованного объединяющего типа.
24. Что такое битовые поля и в объектах каких типов их можно использовать?
25. Каким образом можно получить доступ к любым разрядам внутреннего представления переменной?
26. Можно ли ввести в битовое поле значение из входного потока?

Глава 7
Ввод и вывод

В языке Си отсутствуют средства ввода-вывода. Все операции ввода-вывода реализуются с помощью функций, находящихся в библиотеке языка Си, поставляемой в составе конкретной системы программирования Си.

Особенностью языка Си, который впервые был применен при разработке операционной системы UNIX, является отсутствие заранее спланированных структур файлов. Все файлы рассматриваются как неструктурированная последовательность байтов. При таком подходе удалось распространить понятие файла на различные устройства. В UNIX конкретному устройству соответствует так называемый «специальный файл», а одни и те же функции библиотеки языка Си используются как для обмена данными с файлами, так и для обмена с устройствами.

> **Примечание**
>
> В UNIX эти же функции используются также для обмена данными между процессами (выполняющимися программами).

Библиотека языка Си поддерживает два уровня ввода-вывода: потоковый ввод-вывод и ввод-вывод нижнего уровня.

7.1. Потоковый ввод-вывод

На уровне потокового ввода-вывода обмен данными производится побайтно. Такой ввод-вывод возможен как для собственно устройств побайтового обмена (печатающее устройство, дисплей), так и для файлов на диске, хотя устройства внешней памяти, строго говоря, являются устройствами поблочного обмена, то есть за одно обращение к устройству производится считывание или запись фиксированной порции данных. Чаще всего минимальной порцией

данных, участвующей в обмене с внешней памятью, являются блоки в 512 байт. При вводе с диска (при чтении из файла) данные помещаются в буфер операционной системы, а затем побайтно или определенными порциями передаются программе пользователя. При выводе данных в файл они накапливаются в буфере, а при заполнении буфера записываются в виде единого блока на диск за одно обращение к последнему. Буферы операционной системы реализуются в виде участков основной памяти. Поэтому пересылки между буферами ввода-вывода и выполняемой программой происходят достаточно быстро, в отличие от реальных обменов с физическими устройствами.

Функции библиотеки ввода-вывода языка Си, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный ввод и вывод. Таким образом, *поток* – это файл либо внешнее устройство вместе с предоставляемыми средствами буферизации.

При работе с потоком можно производить следующие действия:

- открывать и закрывать потоки (связывать указатели на потоки с конкретными файлами);
- вводить и выводить: символ, строку, форматированные данные, порцию данных произвольной длины;
- анализировать ошибки потокового ввода-вывода и условие достижения конца потока (конца файла);
- управлять буферизацией потока и размером буфера;
- получать и устанавливать указатель (индикатор) текущей позиции в потоке.

Для того чтобы можно было использовать функции библиотеки ввода-вывода языка Си, в программу необходимо включить заголовочный файл **stdio.h** (**#include <stdio.h>**), который содержит прототипы функций ввода-вывода, а также определения констант, типов и структур, необходимых для работы функций обмена с потоком.

На рис. 7.1 показаны возможные информационные обмены исполняемой программы на локальной (не сетевой) ЭВМ.

7.1.1. Открытие и закрытие потока

Прежде чем начать работать с потоком, его необходимо инициализировать, то есть открыть. При этом поток связывается в исполняемой программе со структурой предопределенного типа **FILE**.

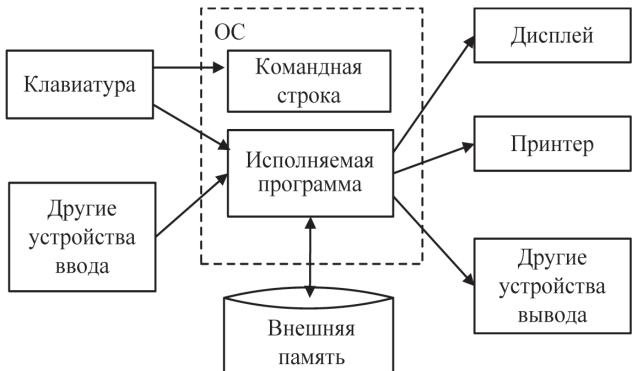


Рис. 7.1. *Информационные обмены исполняемой программы на локальной ЭВМ*

Определение структурного типа **FILE** находится в заголовочном файле **stdio.h**. В структуре **FILE** содержатся компоненты, с помощью которых ведется работа с потоком, в частности указатель на буфер, указатель (индикатор) текущей позиции в потоке и другая информация.

При открытии потока в программу возвращается указатель на поток, являющийся указателем на объект структурного типа **FILE**. Этот указатель идентифицирует поток во всех последующих операциях.

Указатель на поток, например fp, должен быть объявлен в программе следующим образом:

```
#include <stdio.h>
FILE *fp;
```

Указатель на поток приобретает значение в результате выполнения функции открытия потока:

fp = **fopen** (*имя_файла*, *режим_открытия*);

Параметры *имя_файла* и *режим_открытия* являются указателями на массивы символов, содержащих соответственно имя файла, связанного с потоком, и строку режимов открытия. Однако эти параметры могут задаваться и непосредственно в виде строк при вызове функции открытия файла:

```
fp = fopen("t.txt", "r");
```

где `t.txt` – имя некоторого файла, связанного с потоком; `r` – обозначение одного из режимов работы с файлом (тип доступа к потоку).

Поток можно открыть в текстовом либо двоичном (бинарном) режиме.

Стандартно файл, связанный с потоком, можно открыть в одном из следующих шести текстовых режимов:

- «**w**» – новый текстовый (см. ниже) файл открывается для записи. Если файл уже существовал, то предыдущее содержимое стирается, файл создается заново;
- «**r**» – существующий текстовый файл открывается только для чтения;
- «**a**» – текстовый файл открывается (или создается, если файла нет) для добавления в него новой порции информации (добавление в конец файла). В отличие от режима «**w**», режим «**a**» позволяет открывать уже существующий файл, не уничтожая его предыдущей версии, и писать в продолжение файла;
- «**w+**» – новый текстовый файл открывается для записи и последующих многократных исправлений. Если файл уже существует, то предыдущее содержимое стирается. Последующие после открытия файла запись и чтение из него допустимы в любом месте файла, в том числе запись разрешена и в конец файла, то есть файл может увеличиваться («расти»);
- «**r+**» – существующий текстовый файл открывается как для чтения, так и для записи в любом месте файла; однако в этом режиме невозможна запись в конец файла, то есть недопустимо увеличение размеров файла;
- «**a+**» – текстовый файл открывается или создается (если файла нет) и становится доступным для изменений, то есть для записи и для чтения в любом месте; при этом, в отличие от режима «**w+**», можно открыть существующий файл и не уничтожать его содержимого; в отличие от режима «**r+**», в режиме «**a+**» можно вести запись в конец файла, то есть увеличивать его размеры.

В текстовом режиме прочитанная из потока комбинация символов CR (значение 13) и LF (значение 10), то есть управляющие коды «возврат каретки» и «перевод строки», преобразуется в один символ новой строки '\n' (значение 10, совпадающее с LF). При записи в поток в текстовом режиме осуществляется обратное преобразование, то есть символ новой строки '\n' (LF) заменяется последовательностью CR и LF.

Если файл, связанный с потоком, хранит не текстовую, а произвольную двоичную информацию, то указанные преобразования не нужны и могут быть даже вредными. Обмен без такого преобразования выполняется при выборе двоичного или бинарного режима, который обозначается буквой **b**. Например, «**r+b**» или «**wb**». В некоторых компиляторах текстовый режим обмена обозначается буквой **t**, то есть записывают «**a+t**» или «**rt**».

Если поток открыт для изменений, то есть в параметре режима присутствует символ «+», то разрешен как вывод в поток, так и чтение из него. Однако смена режима (переход от записи к чтению и обратно) должна происходить лишь после установки указателя потока в нужную позицию (см. §7.1.3).

При открытии потока могут возникнуть следующие ошибки: указанный файл, связанный с потоком, не найден (для режима «чтение»); диск заполнен или диск защищен от записи и т. п. Необходимо также отметить, что при выполнении функции **fopen( )** происходит выделение динамической памяти. При ее отсутствии устанавливается признак ошибки «Not enough memory» (Недостаточно памяти). В перечисленных случаях указатель на поток приобретает значение **NULL**. Заметим, что указатель на поток в любом режиме, отличном от аварийного, никогда не бывает равным **NULL**.

Приведем типичную последовательность операторов, которая используется при открытии файла, связанного с потоком:

```
if ((fp = fopen("t.txt",   "w")) == NULL)
   {
    perror("ошибка при открытии файла t.txt \n");
    exit(0);
   }
```

где **NULL** – нулевой указатель, определенный в файле **stdio.h**.

Для вывода на экран дисплея сообщения об ошибке при открытии потока используется стандартная библиотечная функция **perror( )**, прототип которой в **stdio.h** имеет вид:

```
void perror(const char * s);
```

Функция **perror( )** выводит строку символов, адресуемую аргументом, за которой размещается сообщение об ошибке. Содержимое и формат сообщения определяются реализацией системы про-

граммирования. Текст сообщения об ошибке выбирается функцией **perror( )** на основании номера ошибки. Номер ошибки заносится в переменную **int errno** (определенную в заголовочном файле **errno.h**) рядом функций библиотеки языка Си, в том числе и функциями ввода-вывода.

После того как файл открыт, с ним можно работать, записывая в него информацию или считывая ее (в зависимости от режима).

Открытые на диске файлы после окончания работы с ними рекомендуется закрыть явно. Для этого используется библиотечная функция

int fclose (*указатель_на_поток*);

Открытый файл можно открыть повторно (например, для изменения режима работы с ним) только после того, как файл будет закрыт с помощью функции **fclose( )**.

7.1.2. Стандартные потоки и функции для работы с ними

Когда программа начинает выполняться, автоматически открываются пять потоков, из которых основными являются:

- стандартный поток ввода (на него ссылаются, используя предопределенный указатель на поток **stdin**);
- стандартный поток вывода (**stdout**);
- стандартный поток вывода сообщений об ошибках (**stderr**).

По умолчанию стандартному потоку ввода **stdin** ставится в соответствие клавиатура, а потокам **stdout** и **stderr** соответствует экран дисплея.

Для ввода-вывода данных с помощью стандартных потоков в библиотеке языка Си определены следующие функции:

- **getchar( )/putchar( )** – ввод-вывод отдельного символа;
- **gets( )/puts( )** – ввод-вывод строки;
- **scanf( )/printf( )** – ввод-вывод в режиме форматирования данных.

Ввод-вывод отдельных символов. Одним из наиболее эффективных способов осуществления ввода-вывода одного символа является использование библиотечных функций **getchar( )** и **putchar( )**. Прототипы функций **getchar( )** и **putchar( )** имеют следующий вид:

int getchar(void);
int putchar(int c);

Функция **getchar( )** считывает из входного потока один символ и возвращает этот символ в виде значения типа **int**.

Функция **putchar( )** выводит в стандартный поток один символ и возвращает этот символ в точку вызова в виде значения типа **int**.

Обратите внимание на то, что функция **getchar( )** вводит очередной символ в виде значения типа **int**. Это сделано для того, чтобы гарантировать успешность распознавания ситуации «достигнут конец файла». Дело в том, что при чтении из файла с помощью функции **getchar**( ) может быть достигнут конец файла. В этом случае операционная система в ответ на попытку чтения символа передает функции **getchar**( ) значение **EOF** (End of File). Константа **EOF** определена в заголовочном файле **stdio.h** и в разных операционных системах имеет значение 0 или –1. Таким образом, функция **getchar**( ) должна иметь возможность прочитать из входного потока не только символ, но и целое значение. Именно с этой целью функция **getchar**( ) всегда возвращает значение типа **int**.

Применение в программах константы **EOF** вместо конкретных целых значений, возвращаемых при достижении конца файла, делает программу мобильной (независимой от конкретной операционной системы).

В случае ошибки при вводе функция **getchar( )** также возвращает **EOF**.

Строго говоря, функции **getchar( )** и **putchar( )** функциями не являются, а представляют собой макросы, определенные в заголовочном файле **stdio.h** следующим образом:

```
#define  getchar()     getc(stdin)
#define  putchar(c)    putc  ((c),  stdout)
```

Здесь переменная **c** определена как **int** c.

Функции **getc( )** и **putc( )**, с помощью которых определены функции **getchar( )** и **putchar( )**, имеют следующие прототипы:

int getc (**FILE** *stream);
int putc (**int** c, **FILE** *stream);

Указатели на поток, задаваемые в этих функциях в качестве параметра, при определении макросов **getchar( )** и **putchar( )** имеют соответственно предопределенные значения **stdin** (стандартный ввод) и **stdout** (стандартный вывод).

Интересно отметить, что функции **getc( )** и **putc( )**, в свою очередь, также реализуются как макросы. Соответствующие строки легко найти в заголовочном файле **stdio.h**.

Не разбирая подробно этих макросов, заметим, что вызов библиотечной функции **getc( )** приводит к чтению очередного байта информации из стандартного ввода с помощью системной функции (ее имя зависит от операционной системы) лишь в том случае, если буфер операционной системы окажется пустым. Если буфер непуст, то в программу пользователя будет передан очередной символ из буфера. Это необходимо учитывать при работе с программой, которая использует функцию **getchar( )** для ввода символов.

При наборе текста на клавиатуре коды символов записываются во внутренний буфер операционной системы. Одновременно они отображаются (для визуального контроля) на экране дисплея. Набранные на клавиатуре символы можно редактировать (удалять и набирать новые). Фактический перенос символов из внутреннего буфера в программу происходит при нажатии клавиши <Enter>. При этом код клавиши <Enter> также заносится во внутренний буфер. Таким образом, при нажатии на клавишу 'A' и клавишу <Enter> (завершение ввода) во внутреннем буфере оказываются код символа 'A' и код клавиши <Enter>.

Об этом необходимо помнить, если вы рассчитываете на ввод функцией **getchar( )** *одиночного* символа.

Приведем фрагмент программы, в которой функция **getchar( )** используется для временной приостановки работы программы с целью анализа промежуточных значений контролируемых переменных.

```
#include    <stdio.h>
int  main()
{
   printf("a");
   getchar();              /*  #1  */
   printf("b");
   getchar();              /*  #2  */
   printf("c");
   return  0;
}
```

Сначала на экран выводится символ 'a', и работа программы приостанавливается до ввода очередного (в данном случае – любого) символа. Если нажать, например, клавишу <q> и затем клавишу

<Enter> (для завершения ввода), то на следующей строке появятся символы **bc**, и программа завершит свою работу. Первая (в программе) функция **getchar( ) (#1)** прочитала из внутреннего буфера код символа 'q', и следующая за ней функция **printf**( ) вывела на экран символ 'b'. Остановки работы программы не произошло, потому что вторая функция **getchar( ) (#2)** прочитала код клавиши <Enter> из внутреннего буфера, а не очередной символ с клавиатуры. Произошло это потому, что к моменту вызова функции **getchar( )** внутренний буфер не был пуст.

Приведенная программа будет работать правильно, если в момент остановки программы нажимать только клавишу <Enter>.

Функция **putchar( )** служит для вывода на устройство стандартного вывода одного символа, заданного в качестве параметра. Ниже приведены примеры:

```
    int  c;
1   c  =  getchar(  );
2   putchar(c);
3   putchar('A');
4   putchar('\007');
5   putchar('\t');
```

В строке 2 фрагмента программы на экран дисплея выводится символ, введенный в предыдущей строке функцией **getchar( )**. В строке 3 выводится символ 'A', в строке 4 выводится управляющий символ, заданный кодом 007 (звуковой сигнал). В последней строке выводится неотображаемый (управляющий) символ табуляции, перемещающий курсор в следующую позицию табуляции.

Приведем в качестве примера применения функций **getchar( )** и **putchar( )** программу копирования данных из стандартного ввода в стандартный вывод, которую можно найти практически в любом пособии по программированию на Си:

```
#include   <stdio.h>
int  main()
{
   int  c;
   while   ((c  =  getchar())  !=  EOF)
         putchar(c);
   return  0;
}
```

Эта же программа с использованием функций **getc( )** и **putc( )** будет выглядеть так:

```
#include   <stdio.h>
int  main()
{
   int  c;
   while   ((c  =  getc(stdin))  !=  EOF)
         putc(c,  stdout);
   return  0;
}
```

Для завершения приведенной выше программы копирования необходимо ввести с клавиатуры сигнал прерывания **Ctrl**+**C** (одновременно нажать клавиши <Ctrl> и <C>).

В [1, §1.5] приведены примеры использования функций **getchar( )** и **putchar( )** для подсчета во входном потоке числа отдельных введенных символов, строк и слов.

Ввод-вывод строк. Одной из наиболее популярных операций ввода-вывода является операция ввода-вывода строки символов. В библиотеку языка Си для обмена со стандартными потоками ввода-вывода включены функции ввода-вывода строк **gets( )** и **puts( )**. Прототипы этих функций имеют следующий вид:

char * gets(**char** * s); /* Функция ввода */
int puts(**char** * s); /* Функция вывода */

Обе функции имеют только один аргумент – указатель **s** на массив символов. Если строка прочитана удачно, функция **gets( )** возвращает адрес того массива s, в который производился ввод строки. Если произошла ошибка, то возвращается **NULL**.

Функция **puts( )** выводит в стандартный выходной поток символы из массива, адресованного аргументом. В случае успешного завершения возвращает последний выведенный символ, который всегда является символом '\n'. Если произошла ошибка, то возвращается **EOF**.

Приведем простейший пример использования этих функций.

```
#include  <stdio.h>
char  str1[  ]  =  "Введите  фамилию  сотрудника:";
int  main()
{
   char  name[80];
```

```
   puts(str1);
   gets(name);
   return 0;
}
```

Напомним, что любая строка символов в языке Си должна заканчиваться терминальным символом '\0'. В последний элемент массива str1 терминальный символ будет записан автоматически при инициализации массива. Для функции **puts( )** наличие нуль-символа в конце строки является обязательным. В противном случае, то есть при отсутствии в символьном массиве символа '\0', программа может завершиться аварийно, так как функция **puts( )** в поисках нуль-символа будет перебирать всю доступную память байт за байтом, начиная в нашем примере с адреса str1. Об этом необходимо помнить, если в программе происходит формирование строки для вывода ее на экран дисплея. Функция **gets( )** завершает свою работу при вводе символа '\n', который автоматически передается от клавиатуры в ЭВМ при нажатии на клавишу <Enter>. При этом сам символ '\n' во вводимую строку не записывается. Вместо него в строку помещается терминальный символ '\0'. Таким образом, функция **gets( )** производит ввод «правильной» строки, а не просто последовательности символов.

Здесь следует обратить внимание на следующую особенность ввода данных с клавиатуры. Функция **gets( )** начинает обработку информации от клавиатуры только после нажатия клавиши <Enter>. Таким образом, она «ожидает», пока не будет набрана нужная информация и нажата клавиша <Enter>. Только после этого начинается ввод данных в программу.

Форматный ввод-вывод. В состав стандартной библиотеки языка Си включены функции форматного ввода-вывода. Использование таких функций позволяет создавать программы, способные обрабатывать данные в заданном формате, а также осуществлять элементарный синтаксический анализ вводимой информации уже на этапе ее чтения. Функции форматного ввода-вывода предназначены для ввода-вывода отдельных символов, строк, целых восьмеричных, шестнадцатеричных, десятичных чисел и вещественных чисел всех типов.

Для работы со стандартными потоками в режиме форматного ввода-вывода определены две функции:

- **printf( )** – форматный вывод;
- **scanf( )** – форматный ввод.

Форматный вывод в стандартный выходной поток. Прототип функции **printf( )** имеет вид:

int printf(const char *format, . . .);

При обращении к функции **printf( )** возможны две формы задания первого параметра:

int printf (*форматная_строка, список_аргументов*);
int printf (*указатель_на_форматную_строку, список_аргументов*);

В обоих случаях функция **printf( )** преобразует данные из внутреннего представления в символьный вид в соответствии с форматной строкой и выводит их в выходной поток. Данные, которые преобразуются и выводятся, задаются как аргументы функции **printf( )**.

Возвращаемое значение функции **printf( )** – число напечатанных символов; а в случае ошибки – отрицательное число.

Форматная_строка ограничена двойными кавычками и может включать произвольный текст, управляющие символы и спецификации преобразования данных. Текст и управляющие символы из форматной строки просто копируются в выходной поток. Форматная строка обычно размещается в списке аргументов функции, что соответствует первому варианту вызова функции **printf( )**. Второй вариант предполагает, что первый аргумент – это указатель типа **char ***, а сама форматная строка определена в программе как обычная строковая константа или переменная.

В список аргументов функции **printf( )** включают выражения, значения которых должны быть выведены из программы. Частные случаи этих выражений – переменные и константы. Количество аргументов и их типы должны соответствовать последовательности спецификаций преобразования в форматной строке. Для каждого аргумента должна быть указана точно одна *спецификация преобразования.*

Если аргументов недостаточно для данной форматной строки, то результат зависит от реализации (от операционной системы и от системы программирования). Если аргументов больше, чем указано в форматной строке, «лишние» аргументы игнорируются.

Список_аргументов (с предшествующей запятой) может отсутствовать.

Спецификация преобразования имеет следующую форму:

% флаги ширина_поля.точность модификатор спецификатор

Символ % является признаком спецификации преобразования. В спецификации преобразования обязательными являются только два элемента: признак % и *спецификатор*.

Спецификация преобразования **не должна** содержать внутри себя пробелов. Каждый элемент спецификации является одиночным символом или числом.

Предупреждение. Спецификации преобразований должны соответствовать типу аргументов. При несовпадении **не будет** выдано сообщений об ошибках во время компиляции или выполнения программы, но в выводе результатов выполнения программы может содержаться произвольная информация.

Начнем рассмотрение составных элементов спецификации преобразования с обязательного элемента – *спецификатора*, который определяет, как будет интерпретироваться соответствующий аргумент: как символ, как строка или как число (табл. 7.1).

Таблица 7.1. Спецификаторы форматной строки для функции форматного вывода

Спецификатор	Тип аргумента	Формат вывода
d	**int, char, unsigned**	Десятичное целое со знаком
i	**int, char, unsigned**	Десятичное целое со знаком
u	**int, char, unsigned**	Десятичное целое без знака
o	**int, char, unsigned**	Восьмеричное целое без знака
x	**int, char, unsigned**	Шестнадцатеричное целое без знака; при выводе используются символы «0…9a…f»
X	**int, char, unsigned**	Шестнадцатеричное целое без знака; при выводе используются символы «0…9A…F
f	**double, float**	Вещественное значение со знаком в виде: *знак_числа*dddd.dddd, где dddd – одна или более десятичных цифр. Количество цифр перед десятичной точкой зависит от величины выводимого числа, а количество цифр после десятичной точки зависит от требуемой точности. *Знак_числа* при отсутствии модификатора '+' изображается только для отрицательного числа

***Таблица 7.1. Спецификаторы форматной строки для функции форматного вывода** (окончание)*

Специ-фикатор	Тип аргу-мента	Формат вывода
e	**double, float**	Вещественное значение в виде: *знак_числа*m.dddd**e***знак*xxx, где m.dddd – изображение мантиссы числа; m – одна десятичная цифра; dddd – последовательность десятичных цифр; **e** – признак порядка; *знак* – знак порядка; xxx – десятичные цифры для представления порядка числа; *знак_числа* при отсутствии модификатора '+' изображается только для отрицательного числа
E	**double, float**	Идентичен спецификатору «e», за исключением того, что признаком порядка служит «E»
g	**double, float**	Вещественное значение со знаком печатается в формате спецификаторов «f» или «e» в зависимости от того, какой из них более компактен для данного значения и точности. Формат спецификатора «e» используется тогда, когда значение показателя меньше –4 или больше заданной точности. Конечные нули отбрасываются, а десятичная точка появляется, если за ней следует хотя бы одна цифра
G	**double, float**	Идентичен формату спецификатора «g», за исключением того, что признаком порядка служит «E»
c	**int, char, unsigned**	Одиночный символ
s	**char ***	Символьная строка. Символы печатаются либо до первого терминального символа ('\0'), или печатается то количество символов, которое задано в поле *точность* спецификации преобразования
p	**void ***	Значение адреса. Печатает адрес, указанный аргументом (представление адреса зависит от реализации)

Приведем примеры использования различных спецификаторов. В каждой строке с вызовом функции **printf( )** в качестве комментариев приведены результаты вывода. Переменная code содержит код возврата функции **printf( )** – число напечатанных символов при выводе значения переменной f.

```
#include <stdio.h>
int main()
{
   int number = 27;
   float f = 123.456;
```

```
    char str[4] = "abc";
    char c =  'a';
    int code;
    printf("%d\n", number);       /*     27      */
    printf("%o\n", number);       /*     33      */
    printf("%x\n", number);       /*     1b      */
    code = printf("%f\n", f);     /* 123.456001  */
    printf("code = %d\n", code); /* code = 11   */
    printf("%e\n",  f);          /* 1.23456e+02 */
    printf("%c\n", c);           /*      a      */
    printf("%s\n", str);         /*     abc     */
    return 0;
}
```

Необязательные элементы спецификации преобразования управляют другими параметрами форматирования.

Флаги управляют выравниванием вывода и печатью знака числа, пробелов, десятичной точки, префиксов восьмеричной и шестнадцатеричной систем счисления. Флаги могут отсутствовать, а если они есть, то могут стоять в любом порядке. Смысл флагов следующий:

- «–» – выводимое изображение значения прижимается к левому краю поля. По умолчанию, то есть при отсутствии этого флага, происходит выравнивание по правой границе поля;
- + – используется для обязательного вывода знака числа. Без этого флага знак выводится только при отрицательном значении;
- пробел – используется для вставки пробела на месте знака перед положительными числами;
- # – если этот флаг используется с форматами «o», «x» или «X», то любое ненулевое значение выводится с предшествующим 0, 0x или 0X соответственно. При использовании флага # с форматами «f», «g», «G» десятичная точка будет выводиться, даже если в числе нет дробной части.

Примеры использования флагов:

- «%+d» – вывод знака '+' перед положительным целым десятичным числом;
- «% d» – добавление (вставка) пробела на месте знака перед положительным числом (использован флаг **пробел** после символа %);
- «%#o» – печать ведущих нулей в изображениях восьмеричных чисел.

Ширина_поля, задаваемая в спецификации преобразования положительным целым числом, определяет минимальное количество позиций, отводимое для представления выводимого значения. Если число символов в выводимом значении меньше, чем *ширина_поля*, выводимое значение дополняется пробелами до заданной минимальной длины. Если *ширина_поля* задана с начальным нулем, не занятые значащими цифрами выводимого значения позиции слева заполняются нулями.

Если число символов в изображении выводимого значения больше, чем определено в *ширине_поля,* или *ширина_поля* не задана, печатаются все символы выводимого значения.

Точность указывается с помощью точки и необязательной последовательности десятичных цифр (отсутствие цифр эквивалентно 0).

Точность задает:

- минимальное число цифр, которые могут быть выведены при использовании спецификаторов **d, i, o, u, x** или **X**;
- число цифр, которые будут выведены после десятичной точки при спецификаторах **e**, **E** и **f**;
- максимальное число значащих цифр при спецификаторах **g** и **G**;
- максимальное число символов, которые будут выведены при спецификаторе **s**.

Непосредственно за точностью может быть указан *модификатор*, который определяет тип ожидаемого аргумента и задается следующими символами:

- **h** – указывает, что следующий после h спецификатор **d**, **i**, **o**, **x** или **X** применяется к аргументу типа **short** или **unsigned short**;
- **l** – указывает, что следующий после **l** спецификатор **d**, **i**, **o**, **x** или **X** применяется к аргументу типа **long** или **unsigned long**;
- **L** – указывает, что следующий после **L** спецификатор **e**, **E**, **f**, **g** или **G** применяется к аргументу типа **long double**.

Примеры указания *ширины_поля* и *точности*:

- %d – вывод десятичного целого в поле, достаточном для представления всех его цифр и знака;
- %7d – вывод десятичного целого в поле из 7 позиций;
- %f – вывод вещественного числа с целой и дробной частями (выравнивание по правому краю; количество цифр в дробной части определяется реализацией и обычно равно 6);
- %7f – вывод вещественного числа в поле из 7 позиций;

- %.3f – вывод вещественного числа с тремя цифрами после десятичной точки;
- %7.3f – вывод вещественного числа в поле из 7 позиций и тремя цифрами после десятичной точки;
- %.0f – вывод вещественного числа без десятичной точки и без дробной части;
- %15s – печать в выводимой строке **не менее** 15 символов;
- %.12s – печать строки длиной **не более** 12 символов («лишние» символы не выводятся);
- %12.12 – печать всегда в поле из 12 позиций, причем лишние символы не выводятся, используются всегда 12 позиций. Таким образом, точное количество выводимых символов можно контролировать, задавая и *ширину_поля*, и *точность* (максимальное количество символов);
- %–15s – выравнивание выводимой строки по левому краю;
- %08f – вывод вещественного числа в поле из 8 позиций. Не занятые значащими цифрами позиции заполняются нулями.

Функция форматного вывода **printf( )** предоставляет множество возможностей по форматированию выводимых данных. Рассмотрим на примере построение столбцов данных и их выравнивание по левому или правому краям заданного поля.

В программе, приводимой ниже, на экран дисплея выводится список товаров в магазине. В каждой строке в отдельных полях указываются: номер товара **(int)**, код товара **(int)**, наименование товара (строка символов) и цена товара **(float)**.

```
#include  <stdio.h>
int main()
{
   int i;
   int number[3]={1, 2, 3}; /* Номер товара */
   int code[3]={10, 25670, 120}; /* Код товара */
   /* Наименование товара: */
   char design[3][30]={{"лампа"}, {"стол"}, {"очень большое кресло"}};
   /* Цена: */
   float price[3]={1152.70, 2400.00, 4824.00};
   for (i=0; i<=2; i++)
      printf("%-3d  %5d  %-20s  %8.2f\n",
           number[i], code[i], design[i], price[i]);
   return 0;
}
```

Результат выполнения программы:

```
1      10    лампа                  1152.70
2   25670    стол                   2400.00
3     120    очень большое кресло   4824.00
```

Форматная строка в параметрах функции **printf( )** обеспечивает следующие преобразования выводимых значений:

1) переменная number[i] типа **int** выводится в поле шириной 3 символа и прижимается к левому краю (%–3d);
2) переменная code[i] типа **int** выводится в поле шириной 5 символов и прижимается (по умолчанию) к правому краю (%5d);
3) строка из массива design[i] выводится в поле шириной 20 символов и прижимается к левому краю (%–20s). Если в данной спецификации преобразования будет указано меньшее, чем 20, количество позиций, то самая длинная строка будет все равно выведена, однако последний столбец не будет выровнен;
4) переменная price[i] типа **float** выводится в поле шириной 8 символов, причем после десятичной точки выводятся 2 символа, и выводимое значение прижимается к правому краю.

Между полями, определенными спецификациями преобразований, выводится столько пробелов, сколько их явно задано в форматной строке между спецификациями преобразования. Таким образом, добавляя пробелы между спецификациями преобразования, можно производить форматирование всей выводимой таблицы.

Напомним, что любые символы, которые появляются в форматной строке и не входят в спецификации преобразования, копируются в выходной поток. Этим обстоятельством можно воспользоваться для вывода явных разделителей между столбцами таблицы. Например, для этой цели можно использовать символ '*' или '|'. В последнем случае форматная строка в функции **printf( )** будет выглядеть так: «%–3d | %5d | %–20s | %8.2f\n», и результат работы программы будет таким:

```
1   |    10 |  лампа                 |  1152.70
2   | 25670 |  стол                  |  2400.00
3   |   120 |  очень большое кресло  |  4824.00
```

Форматный ввод из входного потока. Форматный ввод из входного потока осуществляется функцией **scanf( )**. Прототип функции **scanf( )** имеет вид:

int scanf(const char * format, . . .);

При обращении к функции **scanf( )** возможны две формы задания первого параметра:

int scanf (*форматная_строка, список_аргументов*);
int scanf (*указатель_на_форматную_строку,*
список_аргументов);

Функция **scanf( )** читает последовательности кодов символов (байты) из входного потока и интерпретирует их в соответствии с *форматной_строкой* как целые числа, вещественные числа, одиночные символы, строки. В первом варианте вызова функции форматная строка размещается непосредственно в списке аргументов. Во втором варианте вызова предполагается, что первый аргумент – это указатель типа **char ***, адресующий собственно форматную строку. Форматная строка в этом случае должна быть определена в программе как обычная строковая константа или переменная.

После преобразования во внутреннее представление данные записываются в области памяти, определенные аргументами, которые следуют за форматной строкой. Каждый аргумент должен быть **указателем** на переменную, в которую будет записано очередное значение данных и тип которой соответствует типу, указанному в спецификации преобразования из форматной строки.

Если аргументов недостаточно для данной форматной строки, то результат зависит от реализации (от операционной системы и от системы программирования). Если аргументов больше, чем требуется в *форматной строке*, «лишние» аргументы игнорируются.

Последовательность кодов символов, которую функция **scanf( )** читает из входного потока, как правило, состоит из полей (строк), разделенных символами промежутка или обобщенными пробельными символами. Поля просматриваются и вводятся функцией **scanf( )** посимвольно. Чтение поля прекращается, если встретился пробельный символ или в спецификации преобразования точно указано количество читаемых символов (см. ниже).

Функция **scanf( )** завершает работу, если исчерпана *форматная строка*. При успешном завершении **scanf( )** возвращает количество прочитанных полей (точнее, количество объектов, получивших значения при вводе). Значение **EOF** возвращается при возникновении ситуации «конец файла»; значение –1 – при возникновении ошибки преобразования данных.

Форматная строка ограничена двойными кавычками и может включать:

- пробельные символы, отслеживающие разделение входного потока на поля. Пробельный символ указывает на то, что из входного потока надо считывать, но не сохранять все последовательные пробельные символы вплоть до появления непробельного символа. Один пробельный символ в форматной строке соответствует любому количеству последовательных пробельных символов во входном потоке;
- обычные символы, отличные от пробельных и символа '%'. Обработка обычного символа из форматной строки сводится к чтению очередного символа из входного потока. Если прочитанный символ отличается от обрабатываемого символа форматной строки, функция завершается с ошибкой. Несовпавший символ и следующие за ним входные символы остаются непрочитанными;
- спецификации преобразования.

Спецификация преобразования имеет следующую форму:

*% * ширина_поля модификатор спецификатор*

Все символы в спецификации преобразования являются необязательными, за исключением символа '%', с которого она начинается (он и является признаком спецификации преобразования), и *спецификатора*, позволяющего указать ожидаемый тип элемента во входном потоке (табл. 7.2).

Необязательные элементы спецификации преобразования имеют следующий смысл:

- ***** – звездочка, следующая за символом процента, запрещает запись значения, прочитанного из входного потока по адресу, задаваемому аргументом. Последовательность кодов из входного потока прочитывается функцией **scanf()**, но не преобразуется и не записывается в переменную, определенную очередным аргументом.
- **Ширина_поля** – положительное десятичное целое, определяющее максимальное число символов, которое может быть прочитано из входного потока. Фактически может быть прочитано меньше символов, если встретится пробельный символ или символ, который не может быть преобразован по заданной спецификации.

- **Модификатор** – позволяет задать длину переменной, в которую предполагается поместить *вводимое* значение. Модификатор может принимать следующие значения:
  - **L** – означает, что соответствующий спецификации преобразования аргумент должен указывать на объект типа **long double**;
  - **l** – означает, что аргумент должен быть указателем на переменную типа **long**, **unsigned long** или **double**;
  - **h** – означает, что аргумент должен быть указателем на тип **short**.

Спецификация преобразования имеет следующую форму:

*%   *   ширина_поля   модификатор   спецификатор*

Все символы в спецификации преобразования являются необязательными, за исключением символа '%', с которого она начинается (он и является признаком спецификации преобразования), и *спецификатора*, позволяющего указать ожидаемый тип элемента во входном потоке (табл. 7.2).

Необязательные элементы спецификации преобразования имеют следующий смысл:

- ** * –** звездочка, следующая за символом процента, запрещает запись значения, прочитанного из входного потока по адресу, задаваемому аргументом. Последовательность кодов из входного потока прочитывается функцией **scanf()**, но не преобразуется и не записывается в переменную, определенную очередным аргументом.
- **Ширина_поля –** положительное десятичное целое, определяющее максимальное число символов, которое может быть прочитано из входного потока. Фактически может быть прочитано меньше символов, если встретится пробельный символ или символ, который не может быть преобразован по заданной спецификации.
- **Модификатор** – позволяет задать длину переменной, в которую предполагается поместить *вводимое* значение. Модификатор может принимать следующие значения:
  - **L** – означает, что соответствующий спецификации преобразования аргумент должен указывать на объект типа **long double**;
  - **l** – означает, что аргумент должен быть указателем на переменную типа **long**, **unsigned long** или **double**;

- **h** – означает, что аргумент должен быть указателем на тип **short**.

Таблица 7.2. Спецификаторы форматной строки для функции форматного ввода

Спецификатор	Ожидаемый тип вводимых данных	Тип аргумента
d	Десятичное целое	**int ***
o	Восьмеричное целое	**int ***
x	Шестнадцатеричное целое	**int ***
i	Десятичное, восьмеричное или шестнадцатеричное целое	**int ***
u	Десятичное целое без знака	**unsigned int ***
e, f, g	Вещественное значение вида: [+|-]dddd [E|e[+|-]dd], состоящее из необязательного знака (+ или –), последовательности из одной или более десятичных цифр, возможно, содержащих десятичную точку, и необязательного порядка (признак «e» или «E», за которым следует целое значение, возможно, со знаком)	**float ***
le, lf lg	Для ввода значений переменных типа **double** используются спецификаторы «%le», «%lf», «%lg»	**double ***
Le, Lf, Lg	Для ввода значений переменных типа **long double** используются спецификаторы «%Le», «%Lf», «%Lg»	**long double ***
c	Очередной читаемый символ должен всегда восприниматься как значимый символ. Пропуск начальных пробельных символов в этом случае подавляется. (Для ввода ближайшего, отличного от пробельного, символа необходимо использовать спецификацию «%1s».)	**char ***
s	Строка символов, ограниченная справа и слева пробельными символами. Для чтения строк, не ограниченных пробельными символами, вместо спецификатора **s** следует использовать набор символов в квадратных скобках. Символы из входного потока читаются до первого символа, отличного от символов в квадратных скобках. Если же первым символом в квадратных скобках задан символ '^', то символы из входного потока читаются до первого символа из квадратных скобок	Указатель **char *** на массив символов, достаточный для размещения входной строки, и терминального символа ('\0'), который добавляется автоматически

Контроль заключается в том, что во входном потоке должна присутствовать именно строка *«code:»* (без кавычек). Строка *строка1* используется для комментирования вводимых данных, может иметь произвольную длину и пропускается при вводе. Отметим, что ***строка1*** и ***строка2*** не должны содержать внутри себя пробелов. Текст программы приводится ниже:

```
#include  <stdio.h>
int main()
{
   int i;
   int ret; /* Код возврата функции scanf() */
   char c, s[80];
   ret=scanf("code: %d %*s %c %s", &i, &c, s);
   printf("\n  i=%d  c=%c  s=%s",  i,  c,  s);
   printf("\n  \t  ret = %d\n", ret);
   return 0;
}
```

Рассмотрим форматную строку функции **scanf( )**:

```
"code: %d  %*s  %c  %s"  <Enter>
```

Строка *«code:»* присутствует во входном потоке для контроля вводимых данных и поэтому указана в форматной строке. Спецификации преобразования задают следующие действия:

- %d – ввод десятичного целого;
- %*s – пропуск строки (*строка1* в приведенной выше форме ввода);
- %c – ввод одиночного символа;
- %s – ввод строки.

Приведем результаты работы программы для трех различных наборов входных данных.

1. Последовательность символов исходных данных:

```
code:  5  поле2  D  asd  <Enter>
```

Результат выполнения программы:

```
i=5  c=D  s=asd
     ret=3
```

Значением переменной ret является код возврата функции **scanf( )**. Число 3 говорит о том, что функция **scanf( )** ввела данные без ошибки и было обработано 3 входных поля (строки «code:» и «поле2» пропускаются при вводе).

2. Последовательность символов исходных данных:

```
code:   5   D   asd         <Enter>
```

Результат выполнения программы:

```
i=5   c=a   s=sd
ret=3
```

Обратите внимание на то, что во входных данных пропущена строка перед символом D, использующаяся как комментарий. В результате символ D из входного потока был (в соответствии с форматной строкой) пропущен, а из строки «asd» в соответствии с требованием спецификации преобразования %c был введен символ 'a' в переменную **c**. Остаток строки «asd» (sd) был введен в массив символов s. Код возврата (ret=3) функции **scanf( )** говорит о том, что функция завершила свою работу без ошибок и обработала 3 поля.

3. Последовательность символов исходных данных:

```
cod:   5   поле2   D   asd        <Enter>
```

Результат выполнения программы:

```
i=40   c=     s=  )
ret=0
```

Вместо предусмотренной в форматной строке последовательности символов в данных входного потока допущена ошибка (набрано слово cod: вместо code:). Функция **scanf( )** завершается аварийно (код возврата равен 0). Функция **printf( )** в качестве результата напечатала случайные значения переменных i, c и s.

Необходимо иметь в виду, что функция **scanf( )**, обнаружив какую-либо ошибку при преобразовании данных входного потока, завершает свою работу, а необработанные данные остаются во входном потоке. Если функция **scanf( )** применяется для организации диалога с пользователем, то, будучи активирована повторно, она дочитает из входного потока «остатки» предыдущей порции данных, а не начнет анализ новой порции, введенной пользователем.

Предположим, что программа запрашивает у пользователя номер дома, в котором он живет. Программа обрабатывает код возврата функции **scanf( )** в предположении, что во входных данных присутствует одно поле.

```
#include  <stdio.h>
int main()
{
   int number;
   printf("Введите номер дома: ");
   while (scanf("%d", &number) != 1)
   printf("Ошибка. Введите номер дома: ");
   printf("Номер вашего дома %d\n",  number);
   return 0;
}
```

При правильном вводе данных (введено целое десятичное число) результат будет следующий:

```
Введите номер дома: 25   <Enter>
Номер вашего дома 25
```

Предположим, что пользователь ошибся и ввел, например, следующую последовательность символов «@%». (Эти символы будут введены, если нажаты клавиши '2' и '5', но на верхнем регистре, то есть одновременно была нажата клавиша <Shift>.) В этом случае получится следующий результат:

```
Введите номер дома: @%   <Enter>
Ошибка. Введите номер дома:
Ошибка. Введите номер дома:
......
```

Ошибочный символ @ прерывает работу функции **scanf( )**, но сам остается во входном потоке, и вновь делается попытка его преобразования при повторном вызове функции в теле цикла **while**. Однако эта и последующие попытки ввода оказываются безуспешными, и программа переходит к выполнению бесконечного цикла. Необходимо до предоставления пользователю новой попытки ввода номера дома убрать ненужные символы из входного потока. Это предусмотрено в следующем варианте той же программы:

```
#include  <stdio.h>
int main()
{
   int number;
   printf("Введите номер дома: ");
   while (scanf("%d",  &number) != 1)
      {
         /* Ошибка. Очистить входной поток: */
         while (getchar() != '\n')
         ;
         printf("Ошибка. Введите номер дома: ");
      }
   printf("Номер вашего дома %d\n",  number);
   return 0;
}
```

Теперь при неправильном вводе данных результат будет таким:

```
Введите номер дома: @%      <Enter>
Ошибка. Введите номер дома: 25   <Enter>
Номер вашего дома 25
```

7.1.3. Работа с файлами на диске

Так же как это делается при работе со стандартными потоками ввода-вывода **stdin** и **stdout**, можно осуществлять работу с файлами на диске. Для этой цели в библиотеку языка Си включены следующие функции:

- **fgetc( )**, **getc( )** – ввод (чтение) одного символа из файла;
- **fputc( ), putc( )** – запись одного символа в файл;
- **fprintf( )** – форматированный вывод в файл;
- **fscanf( )** – форматированный ввод (чтение) из файла;
- **fgets( )** – ввод (чтение) строки из файла;
- **fputs( )** – запись строки в файл.

Двоичный (бинарный) режим обмена с файлами. Двоичный режим обмена организуется с помощью функций **getc( )** и **putc( )**, обращение к которым имеет следующий формат:

c = **getc**(fp);
putc(c, fp);

где fp – указатель на поток; c – переменная типа **int** для приема очередного символа из файла или для записи ее значения в файл.

Прототипы функции:

int getc (**FILE** *stream);
int putc **(** **int** c, **FILE** *stream);

В качестве примера использования функций **getc( )** и **putc( )** рассмотрим программы ввода данных в файл с клавиатуры и программу вывода их на экран дисплея из файла.

Программа ввода читает символы с клавиатуры и записывает их в файл. Пусть признаком завершения ввода служит поступивший от клавиатуры символ '#'. Имя файла запрашивается у пользователя. Если при вводе последовательности символов была нажата клавиша <Enter>, служащая разделителем строк при вводе с клавиатуры, то в файл записываются управляющие коды «Возврат каретки» (CR – значение 13) и «Перевод строки» (LF – значение 10). Код CR в дальнейшем при выводе вызывает перевод курсора в начало строки. Код LF служит для перевода курсора на новую строку дисплея. Значения этих кодов в тексте программы обозначены соответственно идентификаторами CR и LF, то есть CR и LF – именованные константы. Запись управляющих кодов CR и LF в файл позволяет при последующем выводе файла отделить строки друг от друга.

В приводимых ниже программах используются уже рассмотренные выше функции **getchar( )**, **putchar( )** для посимвольного обмена со стандартными потоками **stdin**, **stdout**.

```
/* Программа ввода символов */
#include  <stdio.h>
int main()
{
   FILE *fp;   /* Указатель на поток */
   char c;
   /* Восьмеричный код "Возврат каретки": */
   const char CR='\015';
   /* Восьмеричный код "Перевод строки":   */
   const  char LF = '\012';
   char fname[20]; /* Массив для имени файла */
   /* Запрос имени файла: */
  puts("введите имя файла: \n");
  gets(fname);
   /* Открыть файл для записи: */
  if ((fp = fopen(fname,"w")) == NULL)
      {
         perror(fname);
```

```
        return 1;
      }
   /* Цикл ввода и записи в файл символов: */
   while  ((c = getchar()) != '#')
       {
          if  (c == '\n')
             {
                putc(CR, fp);
                putc(LF, fp);
             }
          else  putc(c, fp);
       }
   /* Цикл ввода завершен; закрыть файл: */
   fclose(fp);
   return 0;
}
```

Следующая программа читает поток символов из ранее созданного файла и выводит его на экран дисплея:

```
/* Программа вывода символьного файла на экран дисплея */
#include <stdio.h>
int main()
{
   FILE *fp; /* Указатель на поток */
   char c;
   char fname[20]; /* Массив для имени файла */
   /* Запрос имени файла:  */
   puts("введите имя файла: \n ");
   gets(fname);
   /* Открыть файл для чтения: */
   if ((fp = fopen(fname,"r")) == NULL)
       {
          perror(fname);
          return 1;
       }
   /* Цикл чтения из файла и вывода символов на экран: */
   while ((c = getc(fp)) != EOF)
           putchar(c);
   fclose(fp); /* Закрыть файл */
   return 0;
}
```

Программу чтения символов из файла можно усовершенствовать, добавив возможность вывода информации на экран порциями (кадрами):

```
/* Усовершенствованная программа вывода
   символьного файла на экран дисплея по кадрам */
#include  <stdio.h>
int main()
{
   FILE *fp;   /* Указатель на поток) */
   char c;
   /* Восьмеричный код "Перевод строки": */
   const char LF='\012';
   int MAX=10; /* Размер кадра */
   int nline;   /* Счетчик строк */
   char fname[10]; /* Массив для имени файла */
   /* Запрос имени файла: */
   puts("введите имя файла \n");
   gets(fname);
   /* Открыть файл для чтения: */
   if ((fp = fopen(fname,»r»)) = = NULL)
   {
      perror(fname);
      return 1;
   }
   while (1)
   {   /* Цикл вывода на экран */
      nline = 1;
      while (nline<MAX) /* Цикл вывода кадра */
      {
         c = getc(fp);
         if (c == EOF)
         {
            fclose(fp);
            return 0;
         }
         if (c == LF) nline++;
               putchar(c);
      } /* Конец цикла вывода кадра*/
      getchar(); /* Задержка вывода до нажатия любой клавиши */
   } /* Конец цикла вывода */
} /* Конец программы */
```

В этой программе после вывода очередного кадра из MAX строк для перехода к следующему кадру необходимо нажать любую клавишу.

Используя двоичный обмен с файлами, можно сохранять на диске информацию, которую нельзя непосредственно вывести на экран дисплея (целые и вещественные числа во внутреннем представлении).

Строковый обмен с файлами. Для работы с текстовыми файлами, кроме перечисленных выше, удобно использовать функции **fgets( )** и **fputs( )**. Прототипы этих функций в файле **stdio.h** имеют следующий вид:

```
int fputs(const char *s, FILE *stream);
char * fgets(char *s, int n, FILE *stream);
```

Функция **fputs( )** записывает строку (на которую указывает s) в файл, определенный указателем stream на поток, и возвращает неотрицательное целое. При ошибках функция возвращает **EOF**.

Функция **fgets( )** читает из определенного указателем stream файла не более (n–1) символов и записывает их в строку, на которую указывает s. Функция прекращает чтение, как только прочтет (n–1) символов или встретит символ новой строки '\n', который переносится в строку s. Дополнительно в конец каждой строки записывается признак окончания строки '\0'. В случае успешного завершения функция возвращает указатель s. При ошибке или при достижении конца файла, при условии, что из файла не прочитан ни один символ, возвращается значение **NULL**. В этом случае содержимое массива, который адресуется указателем s, остается без изменений.

Проиллюстрируем возможности указанных функций на программе, которая переписывает текст из одного файла в другой. На этом же примере еще раз проиллюстрируем особенность языка Си – возможность передачи информации в выполняемую программу из командной строки (см. §5.8).

В стандарте языка Си определено, что любая программа на языке Си при запуске на выполнение получает от операционной системы информацию в виде значений двух аргументов – **argc** (сокращение от argument count – счетчик аргументов) и **argv** (сокращение от argument vector). Первый из них определяет количество строк, передаваемых в виде массива указателей argv. По принятому соглашению, argv[0] – это всегда указатель на строку, содержащую полное имя файла с выполняемой программой. Остальные элементы

массива argv[1],...,argv[argc–1] содержат ссылки на параметры, записанные через пробелы в командной строке после имени программы. (Третий параметр **char** * envp[ ] мы уже рассмотрели в §5.8, но он используется весьма редко.)

Предположим, что имя выполняемой программы copyfile.exe, и мы хотим с ее помощью переписать содержимое файла f1.dat в файл f2.txt. Вызов программы из командной строки имеет вид (> – «приглашение» от операционной системы):

```
>copyfile.exe   f1.dat   f2.txt <Enter>
```

Текст программы может быть таким:

```
#include   <stdio.h>
main(int argc, char *argv[ ])
{
   char cc[256];/* Массив для обмена с файлами */
   FILE *f1, *f2; /* Указатели на  потоки */
   if (argc != 3) /* Проверка командной строки */
   {
      printf("\n Формат вызова программы: ");
      printf("\n copyfile.exe"
      printf("\n файл-источник    файл-приемник");
      return 1;
   }
   if ((f1 = fopen(argv[1], "r")) == NULL)
   /* Открытие входного файла */
   {
      perror(argv[1]);
      return 1;
   }
   if ((f2 = fopen(argv[2], "w")) == NULL)
   /* Открытие выходного файла */
   {
      perror(argv[2]);
      return 1;
   }
   while (fgets(cc, 256, f1) != NULL)
         fputs(cc, f2);
   fclose(f1);
   fclose(f2);
   return 0;
}
```

Обратите внимание, что значение argc явно не задается, а определяется автоматически. Так как argv[0] определено всегда, то значение argc не может быть меньше 1. При отсутствии в командной строке двух аргументов (имен файлов ввода-вывода) значение argc оказывается не равным трем, и поэтому на экран выводится поясняющее сообщение:

```
Формат вызова программы:
copyfile.exe      файл-источник      файл-приемник
```

Если имя входного файла указано неудачно или нет места на диске для создания выходного файла, то выдаются соответствующие сообщения:

```
Ошибка при открытии файла f1.dat
```

или

```
Ошибка при открытии файла f2.txt
```

Режим форматного обмена с файлами. В некоторых случаях информацию удобно записывать в файл в виде, пригодном для непосредственного (без преобразования) отображения на экран дисплея, то есть в символьном виде. Например, для сохранения результатов работы программы, чтобы затем распечатать их на бумагу (получить «твердую» копию), или когда необходимо провести трассировку программы – вывести значения некоторых переменных во время выполнения программы для их последующего анализа. В этом случае можно воспользоваться функциями форматного ввода (вывода) в файл **fprintf( )** и **fscanf( )**, которые имеют следующие прототипы:

int fprintf (*указатель_на_поток, форматная-строка,*
список-переменных);
int fscanf (*указатель_на_поток, форматная_строка,*
список_адресов_переменных);

Как и в функциях **printf( )** и **scanf( )**, форматная строка может быть определена вне вызова функции, а в качестве аргумента в этом случае будет указатель на нее.

От рассмотренных выше функций **printf( )**, **scanf( )** для форматного обмена с дисплеем и клавиатурой функции **fprintf( )** и **fscanf( )** отличаются лишь тем, что в качестве первого параметра в них необ-

ходимо задавать указатель на поток, с которым производится обмен. В качестве примера приведем программу, создающую файл int.dat и записывающую в него символьные изображения чисел от 1 до 10 и их квадратов [9]:

```
#include   <stdio.h>
int  main()
{
   FILE  *fp;  /*  Указатель  на  поток  */
   int  n;
   if  ((fp  =  fopen("int.dat","w"))  ==  NULL)
     {
        perror("int.dat");
        return  1;
     }
   for  (n=1;  n<11;  n++)
      fprintf(fp,  "%d   %d\n",  n,   n*n);
   fclose(fp);
   return  0;
}
```

В результате работы этой программы в файле int.dat формируется точно такая же «картинка», как и в том случае, если бы мы воспользовались вместо функции **fprintf( )** функцией **printf( )** для вывода результатов непосредственно на экран дисплея. В этом легко убедиться, воспользовавшись соответствующей обслуживающей программой операционной системы для вывода файла int.dat на экран дисплея. Несложно и самим написать такую программу, используя для форматного чтения данных из файла функцию **fscanf( )**:

```
#include   <stdio.h>
int  main()
{
   FILE  *fp;  /*  Указатель  на  поток  */
   int  n,  nn,  i;
   if  ((fp  =  fopen("int.dat","r"))  ==  NULL)
     {
        perror("int.dat");
        return  1;
     }
   for  (i=1;  i<11;  i++)
      {
         fscanf(fp,  "%d  %d",  &n,   &nn);
```

```
        printf(" %d %d \n",n, nn);
    }
fclose(fp);
return 0;
}
```

Позиционирование в потоке. В начале главы были рассмотрены посимвольный, построчный и форматный обмены с файлами, организованными в виде потока байтов. Эти средства позволяли записать в файл данные и читать из него информацию только последовательно.

Операция чтения (или записи) для потока всегда производится, начиная с текущей позиции в потоке. Начальная позиция чтения/записи в потоке устанавливается при открытии потока и может соответствовать начальному или конечному байту потока в зависимости от режима открытия потока. При открытии потока в режимах «**r**» и «**w**» указатель текущей позиции чтения/записи в потоке устанавливается на начальный байт потока, а при открытии в режиме «**a**» – в конец файла (за конечным байтом). При выполнении каждой операции ввода-вывода указатель текущей позиции в потоке перемещается на новую текущую позицию в соответствии с числом прочитанных (записанных) байтов.

Рассмотрим средства позиционирования в потоке, позволяющие перемещать указатель (индикатор) текущей позиции в потоке на нужный байт. Эти средства дают возможность работать с файлом на диске как с обычным массивом, осуществляя доступ к содержимому файла в произвольном порядке.

В библиотеку языка Си включена функция **fseek( )** для перемещения (установки) указателя текущей позиции в потоке на нужный байт потока (файла). Она имеет следующий прототип:

int fseek (*указатель_на_поток, смещение, начало_отсчета)*;

Смещение задается переменной или выражением типа **long** и может быть отрицательным, то есть возможно перемещение по файлу в прямом и обратном направлениях. Начало отсчета задается одной из предопределенных констант, размещенных в заголовочном файле **stdio.h**:

- **SEEK_SET** (имеет значение 0) – начало файла;
- **SEEK_CUR** (имеет значение 1) – текущая позиция;
- **SEEK_END** (имеет значение 2) – конец файла.

Здесь следует напомнить некоторые особенности данных типа **long**. Этот тип определяется для целочисленных констант и переменных, которым в памяти должно быть выделено больше места, чем данным типа **int**. Обычно переменной типа **long** выделяется 4 байта, чем и определен диапазон ее значений. Описание данных типа **long**:

```
long A, p, Z[16];
```

Константа типа **long** записывается в виде последовательности десятичных цифр, вслед за которыми добавляется разделитель **L** или **l**. Примеры констант типа **long**:

```
0l        0L        10L          688L        33l
```

В текстах программ лучше использовать **L**, чтобы не путать **l** с цифрой 1.

Функция **fseek( )** возвращает 0, если перемещение в потоке (файле) выполнено успешно, в противном случае возвращается ненулевое значение.

Приведем примеры использования функции **fseek( )**. Перемещение к началу потока (файла) из произвольной позиции:

```
fseek(fp, 0L, SEEK_SET);
```

Перемещение к концу потока (файла) из произвольной позиции:

```
fseek(fp, 0L, SEEK_END);
```

В обоих примерах смещение задано явно в виде нулевой десятичной константы 0**L** типа **long**.

При использовании сложных типов данных (таких как структура) можно перемещаться в потоке (файле) на то количество байтов, которое занимает этот тип данных. Пусть, например, определена структура:

```
struct str
    {...
      ...
    } st;
```

Тогда при следующем обращении к функции **fseek( )** указатель текущей позиции в потоке будет перемещен на одну структуру назад относительно текущей позиции:

```
fseek(fp, -(long)sizeof(st), SEEK_CUR);
```

Для иллюстрации работы с файлом в режиме произвольного доступа рассмотрим пример небольшого трехъязычного словаря, содержащего названия цифр от 1 до 9 на английском, немецком и французском языках. Для обслуживания словаря необходимы две программы: программа загрузки словаря в файл на диске и программа поиска слова в словаре (в ней используется функция **fseek( )**).

«База данных» словаря организована в файле vac.dat. Функция загрузки запрашивает названия для цифр от 1 до 9. Названия конкретной цифры (в предположении, что каждое название не превышает 9 букв) на всех трех языках записываются в одной строке через пробелы. Сформированная строка по нажатии клавиши <Enter> прочитывается в массив символов buf[ ]. Текст программы:

```
#include <stdio.h>
int main()
{
   int i, k;
   int n;
   char buf[30];   /* Буфер */
   char *c;
   int *pti;
   FILE *fp; /* Указатель на поток */
   /* Открыть файл для записи */
   if ((fp = fopen("vac.dat","w")) == NULL)
   {
      perror("vac.dat");
      return 1;
   }
   for (i=1; i<=9; i++)
   { /* Очистить буфер: */
    for (k=0; k<30; k++)
          buf[k] = ' ';
    /* Запрос названий цифр */
    n = i;
    pti = &n;
    printf("\n Введите названия цифры %d:\n", i);
```

```
    scanf("%s  %s  %s",  buf,  buf+10,  buf+20);
    /* Запись в файл цифры */
    c = (char *)pti;
    for (k=0; k <2; k++)
         putc(*c++, fp);
    /* Запись в файл названий цифр */
    c = buf;
    for (k=0; k<30; k++)
         putc(*c++, fp);
  }
  fclose(fp);
  return 0;
}
```

Введенные данные (представление цифры в формате целого числа и названия цифр на трех языках) побайтно выводятся в файл vac.dat, образуя в нем «записи», структура которых показана на рис. 7.2.

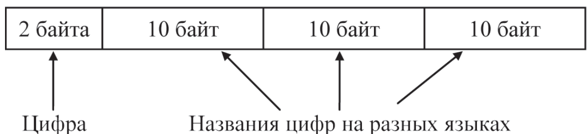


Рис. 7.2. *Состав записи «Цифра»*

Следующая программа перевода (trans) запрашивает сначала язык, на котором нужно выводить названия цифр, а затем в цикле запрашиваются цифры. Язык идентифицируется первой буквой своего русского наименования: 'а' или 'А' – для английского; 'н' или 'Н' – для немецкого; 'ф' или 'Ф' – для французского. Признаком завершения программы служит ввод цифры 0. Текст программы trans:

```
/* Программа перевода trans.c */
#include <stdio.h>
int main()
{
  int i, k;
  char *c;
  long pos; /* Смещение в файле */
  char buf[30];
  char ln;
  FILE *fp; /* Указатель на поток */
```

```
   int lang; /* Индикатор "язык" */
   /* Открыть файл: */
   if ((fp = fopen("vac.dat","r")) == NULL)
   {
      perror("vac.dat");
      return 1;
   }
   /* Запросить язык: */
   lang = -1;
   while (lang == -1)
   {
      puts("\n Введите первую букву названия"
           " языка: (а,н,ф)");
      scanf("%c", &ln);
      if ((ln =='а')||(ln =='А')) lang = 0;
      else
         if ((ln =='н')||(ln =='Н')) lang = 1;
         else
            if ((ln =='ф')||(ln =='Ф')) lang = 2;
   }
   while (1)  /* Цикл перевода */
   {
      c = buf;
      puts("введите цифру (0 - для завершения):");
      scanf("%d", &k);
      if  (k = = 0)
      {
         fclose(fp);
         return 0;
      }
      /* Вычислить смещение */
      pos = (k-1)*32+2+lang*10;
      fseek(fp, pos, SEEK_SET);
      /* Выбрать перевод */
      for (i=0; i<=9; i++)
          *c++=getc(fp);
      c++;
      *c = '\0';
      printf("%d->%s\n", k, buf);
   }
   fclose(fp);
   return 0;
}
```

При вычислении позиции (pos) в файле, с которой начинается строка перевода, участвуют следующие составляющие:

- k–1 – выбирает подстроку (длиной 32 байта), в которой содержится перевод;
- 2 – учитывает длину поля цифры (2 байта);
- lang*10 – задает смещение до требуемого значения цифры в подстроке перевода.

На рис. 7.3 показаны составляющие смещения для искомого поля с переводом на немецкий язык.

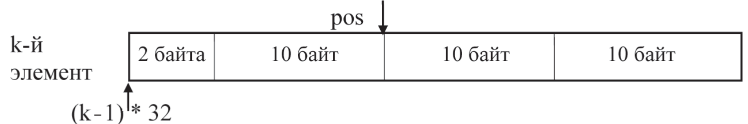


Рис. 7.3. *Составляющие смещения для записей в файле словаря*

Предлагаем самостоятельно разработать функцию поиска в файле заданной подстроки и на ее основе написать программу перевода названий цифр с одного языка на другой. В табл. 7.3 приводятся названия цифр от 1 до 9 на трех языках.

Таблица 7.3. Трехъязычный словарь «Цифры»

№	Английский язык	Немецкий язык	Французский язык
1	ONE	EINS	UN
2	TWO	ZWEI	DEUX
3	THREE	DREI	TROIS
4	FOUR	VIER	QUATRE
5	FIVE	FüNF	CINQ
6	SIX	SECHS	SIX
7	SEVEN	SIEBEN	SEPT
8	EIGHT	ACHT	HUIT
9	NINE	NEUN	NEUF

Кроме рассмотренной функции **fseek( )**, в библиотеке функций языка Си находятся следующие функции для работы с указателями текущей позиции в потоке:

- **long ftell(**FILE ***)** – получить значение указателя текущей позиции в потоке;

- **void rewind(**FILE ***)** – установить указатель текущей позиции в потоке на начало потока.

Необходимо иметь в виду, что недопустимо использовать функции работы с указателем текущей позиции в потоке для потока, связанного не с файлом, а с устройством. Поэтому применение описанных выше функций с любым из стандартных потоков приводит к неопределенным результатам.

7.2. Ввод-вывод нижнего уровня

Ввод-вывод, ориентированный на поток, обычно применяется для выполнения достаточно стандартных операций ввода-вывода. Применение рассмотренных выше функций обмена с потоками гарантирует успешность переноса программы (в отношении ввода-вывода) в различные операционные системы.

Функции ввода-вывода более низкого уровня позволяют пользоваться средствами ввода-вывода операционной системы непосредственно. При этом не выполняются буферизация и форматирование данных. Программы, использующие низкоуровневый ввод-вывод, переносимы в рамках некоторых систем программирования Си, в частности относящихся к UNIX. Учитывая близость функций низкоуровневого ввода-вывода к средствам ввода-вывода операционной системы, можно рекомендовать их применение для разработки собственной подсистемы ввода-вывода, например ориентированной на работу со сложными структурами данных (списки, деревья, сложные записи и т. п.).

При низкоуровневом открытии файла с ним связывается не указатель файла (потока), а дескриптор (handle) файла. Дескриптор является целым значением, характеризующим размещение информации об открытии файла во внутренних таблицах операционной системы. Дескриптор файла используется при последующих операциях с файлом.

В библиотеку языка Си включены следующие основные функции ввода-вывода нижнего уровня:

- **open( )/close( )** – открыть/закрыть файл;
- **creat( )** – создать файл;
- **read( )/write( )** – читать/писать данные;
- **sopen( )** – открыть файл в режиме разделения, то есть для одновременного доступа со стороны нескольких процессов (работающих программ);

- **eof( )** – проверить достижение конца файла;
- **lseek( )** – изменить текущую позицию в файле;
- **tell( )** – получить значение текущей позиции в файле.

Для работы с функциями нижнего уровня в программу должны включаться соответствующие заголовочные файлы. Имена этих файлов могут быть различными в разных операционных системах. Поэтому перед написанием программ, использующих функции ввода-вывода нижнего уровня, или при переносе программ в другую операционную систему необходимо ознакомиться с документацией по библиотеке Си для конкретной операционной системы.

Функции нижнего уровня, в отличие от функций для работы с потоком, не требуют включения в программу заголовочного файла **stdio.h**. Однако этот файл содержит определения ряда констант (например, признак конца файла **EOF**), которые могут оказаться полезными. В случае применения этих констант файл **stdio.h** должен быть включен в программу.

7.2.1. Открытие/закрытие файла

До выполнения операций ввода-вывода в файл (из файла) на низком уровне необходимо открыть или создать файл одной из следующих функций: **open( )**, **sopen( )** или **creat( )**.

Функция **sopen( )** используется в том случае, когда необходимо дать возможность одновременного доступа к файлу для нескольких выполняющихся программ. Разумеется, речь идет о доступе к файлу в режиме чтения. Обычно файл блокируется для доступа со стороны других выполняющихся программ, и именно функция **sopen( )** необходима для разрешения одновременного доступа.

При открытии файла в программу возвращается дескриптор файла, значение которого является целочисленным. В отличие от дескриптора, указатель на поток есть указатель на структуру типа **FILE**, определенного в заголовочном файле **stdio.h**.

Формат вызова функции **open( )**, в результате выполнения которой приобретает значение дескриптор файла:

fd = **open** (*имя_файла, флаги, права_доступа*);

В программе дескриптор файла fd должен быть определен как **int** fd. Параметр *имя_файла* является указателем на массив символов, содержащий имя файла.

Второй параметр *флаги* определяет режим открытия файла, который является выражением, сформированным (с помощью '|' – по-

битовой операции ИЛИ) из одной или более предопределенных констант, размещенных в заголовочном файле **fcntl.h**. В некоторых реализациях UNIX эти константы находятся в файле **sys/file.h**.

> **Примечание**
>
> Обратите внимание на то, что в UNIX при образовании полного имени файла применяется символ '/' (прямой слэш).

Приведем в алфавитном порядке список констант (флагов), задающих режим открытия файла:

- **O_APPEND** – открыть файл для добавления (для записи в конец файла);
- **O_BINARY** – открыть файл в бинарном режиме (см. §7.1.1);
- **O_CREAT** – создать и открыть новый файл;
- **O_EXCL** – этот флаг позволяет избежать непреднамеренного уничтожения уже существующего файла. Если этот флаг указан вместе с флагом **O_CREAT** и файл уже существует, то функция открытия файла завершается с ошибкой;
- **O_RDONLY** – открыть файл только для чтения;
- **O_RDWR** – открыть файл и для чтения, и для записи;
- **O_TEXT** – открыть файл в текстовом режиме (см. §7.1.1);
- **O_TRUNC** – открыть существующий файл и стереть его содержимое (подготовить для записи новой информации).

Обратите внимание на то, что режим открытия файла должен быть задан обязательно, так как его значение по умолчанию не устанавливается.

Третий параметр – *права_доступа* – должен применяться лишь в режиме открытия файла **O_CREAT**, то есть только при создании нового файла.

В операционных системах семейства Windows для задания параметра *права_доступа* используются следующие предопределенные константы:

- **S_IWRITE** – разрешить запись в файл;
- **S_IREAD** – разрешить чтение из файла;
- **S_IREAD|S_IWRITE** – разрешить и чтение, и запись.

Перечисленные константы размещены в заголовочном файле **stat.h**, находящемся в каталоге **sys**. Обычно его подключение осуществляется директивой **#include <sys\stat.h>**.

Если параметр *права_доступа* не указан, то устанавливается разрешение только на чтение из файла. Чаще всего в операционных системах семейства Windows этот параметр не используется.

В ОС UNIX в силу того, то она является многопользовательской, система защиты файлов более развита. Права доступа к файлам устанавливаются для трех категорий пользователей:

- владелец файла;
- участник группы пользователей;
- прочие пользователи.

Права доступа к конкретному файлу устанавливаются владельцем файла специальными командами. Права доступа отображаются при просмотре оглавления каталога командой **ls –l** в виде символьной строки, которая формируется по следующему правилу: для каждой группы пользователей в строке прав доступа выделяются три символа, каждый из которых может принимать следующие значения:

- **r** – разрешено чтение из файла;
- **w** – разрешена запись в файл;
- **x** – разрешено выполнение файла (для файлов, хранящих исполняемую программу).

Символы **r**, **w**, **x** задаются строго на своих местах в указанном порядке **(rwx)**. Если какой-либо из типов доступа к файлу запрещен, на месте соответствующего символа записывается символ '–' (минус). Таким образом, если для владельца файла разрешены все виды доступа к файлу **(rwx)**, для участника группы пользователей – только чтение и выполнение **(r-x)**, а для прочих пользователей – только выполнение **(--x)**, то строка прав доступа будет выглядеть так:

```
rwxr-x--x
```

От этой строки символов легко перейти к собственно параметру *права_доступа*, являющемуся целым числом. Если на соответствующем месте в строке указан символ, отличный от '**-**', то записывают '1', иначе записывают '0'. Получившееся двоичное число (111101001) переводят в восьмеричное, записав в виде восьмеричной цифры каждую группу из трех двоичных цифр, начиная с **самой правой** группы: 0751. Это число и следует указать в качестве параметра *права_доступа* в функции **open( )**.

Приведем несколько примеров открытия файла.

1. Открыть файл для чтения:

```
fd = open("t.txt", O_RDONLY);
```

2. Открыть существующий файл для записи новых данных:

```
fd = open("new.txt",O_WRONLY|O_CREAT|O_TRUNC,0600);
```

Параметр *права_доступа*, заданный восьмеричной константой 0600 (для UNIX), в символьном изображении имеет вид **rw----**, то есть для владельца файла разрешены чтение и запись, для двух других категорий пользователей не разрешен ни один из видов доступа к файлу. Если файл с именем new.txt существует, то перед записью новых данных он будет усечен до нулевого размера (очищен). При открытии файла с указанными параметрами в других ОС параметр *права_доступа* опускается.

3. Открыть файл для добавления:

```
fd=open("t.txt",O_WRONLY|O_APPEND|O_CREAT,0600);
```

4. Открыть файл для чтения и записи:

```
fd = open("t.txt", O_RDWR);
```

5. Создать новый файл для записи:

```
if ((fd = open("tmpfile",O_WRONLY|O_CREAT|O_EXCL, 0666)) == -1)
          puts("tmpfile уже существует\n");
```

В операционной системе UNIX такая последовательность операторов открывает новый файл для записи. Если файл не существует, то он создается. Иначе функция завершается неудачей. Флаг **O_EXCL** специально задан для предотвращения непреднамеренного уничтожения уже существующего файла. Этот флаг используется совместно с **O_CREAT**. Права доступа **(rw-rw-rw-)** разрешают чтение и запись в файл для всех категорий пользователей. В других ОС параметр *права_доступа* должен быть опущен.

Приведем более полный пример создания файла (MS-DOS, Windows):

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

```
#include <io.h>
#include <errno.h>
#include <fcntl.h>
#include <sys\stat.h>
void main( )
{
   int fd; /* Дескриптор файла */
   if ((fd = open("tmpfile",
                        O_RDWR|O_CREAT|O_EXCL,
                        S_IREAD|S_IWRITE)) < 0)
   {
      if (errno == EEXIST)
          fprintf(stderr, "файл tempfile"
                          " уже существует\n");
      exit(errno);
   }
}
```

Создаваемый файл в соответствии с выбранными флагами открывается для чтения и записи. Права доступа **(rw-------)** позволяют только владельцу работать с файлом (читать и писать).

Для идентификации ошибок, возникающих при открытии файла, используется именующее выражение (переменная) **errno**, определенное в заголовочном файле **errno.h**. При выполнении функций стандартной библиотеки в область памяти, именуемой **errno**, записываются коды ошибок. Предопределенная в **errno.h** константа **EEXIST** означает, что файл, указанный в функции **open( )**, уже существует. В этом примере для вывода сообщения об ошибке применена функция форматного вывода в файл **fprintf( )**, в которой использован предопределенный дескриптор файла **stderr** стандартного потока для вывода сообщений об ошибках.

Кроме функции **open( )**, для открытия файла можно использовать функцию **creat( )**, упомянутую в начале параграфа. Функция **creat( )** полностью эквивалентна такому вызову функции **open( )**:

open (*имя_файла*, **O_CREAT|O_TRUNC|O_WRONLY**);

Функция **creat( )** создает новый файл и открывает его для записи. Наличие в библиотеке наряду с функцией **open( )** функции **creat( )** вызвано требованиями совместимости с ранними версиями UNIX, имевшими только три основных режима открытия файла (**O_RDONLY**, **O_WRONLY**, **O_RDWR**), что вынуждало использовать для создания нового файла специальную функцию **creat( )**.

Так же как и при использовании потоков, в начале работы каждой программы автоматически открываются файлы стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках. Эти файлы имеют значения дескрипторов файлов 0, 1 и 2, которые можно использовать при обменах на нижнем уровне со стандартными файлами.

Необходимо иметь в виду, что в каждой операционной системе имеется ограничение на количество одновременно открытых в программе файлов. Обычно их число устанавливается от 20 до 40. Во время работы программы, в которой обрабатывается большое количество файлов, необходимо своевременно закрывать ненужные файлы. Для закрытия файла на нижнем уровне служит функция **close( )**, прототип которой имеет вид:

int close (*дескриптор_файла*);

Функция **close( )** при успешном завершении возвращает 0. В случае ошибки возвращается –1.

При завершении программы все открытые файлы автоматически закрываются.

7.2.2. Чтение и запись данных

Ввод-вывод данных на нижнем уровне осуществляется функциями **read( )** и **write( )**. Прототипы этих функций имеют следующий вид:

int read(**int** fd, **char** *buffer, **unsigned int** count);
int write(**int** fd, **char** *buffer, **unsigned int** count);

Обе функции возвращают целое число – количество действительно прочитанных или записанных байтов.

Функция **read( )** читает количество байтов, заданное третьим параметром count, из файла, открытого с дескриптором файла fd, в буфер, определенный указателем buffer. При достижении конца файла функция **read( )** возвращает значение 0. В случае возникновения ошибки при чтении из файла функция **read( )** возвращает значение –1.

Операция чтения, так же как и для потокового ввода-вывода, начинается с текущей позиции в файле. После завершения операции чтения текущая позиция будет определять первый непрочитанный символ.

Если файл открыт в текстовом режиме, то происходят точно такие же преобразования при вводе последовательности символов CR

и LF в символ '\n' (LF), как и при работе с потоком. Указанное преобразование приводит к тому, что в возвращаемом значении вместо двух символов CR и LF учитывается только один символ '\n' (LF).

Функция **write( )** записывает последовательность байтов, количество которых задано третьим параметром count, в файл, открытый с дескриптором файла fd, из буфера, определенного указателем buffer. Запись производится с текущей позиции. Если файл открыт в текстовом режиме, то количество реально записанных байтов может превышать count за счет преобразований всех символов '\n' в последовательности символов CR, LF.

Если при выполнении операции записи возникла ошибка, то функция **write( )** возвращает значение –1, а глобальная переменная **errno** получает одно из следующих значений, заданных предопределенными константами в заголовочном файле **errno.h**:

- **EACCES** – файл защищен для записи (доступен только для чтения);
- **ENOSPC** – исчерпано свободное пространство на внешнем устройстве;
- **EBADF** – недействительный дескриптор файла.

Приведем два примера применения функций низкоуровневого ввода-вывода.

Пример 1. Копирование последовательности отдельных символов из стандартного ввода в стандартный вывод:

```
#include   <io.h>
int  main(  )
{
   char   c[2];
   while  ((read(0,  c,  1)) > 0)
   write(1,  c,  1);
   return  0;
}
```

В текст программы включается заголовочный файл **io.h**, содержащий прототипы функций **read( )** и **write( )**. При вызове этих функций для файлов стандартного ввода и стандартного вывода используются соответственно значения дескрипторов стандартных файлов 0 и 1. Прочитанный символ и код клавиши <Enter>, который служит признаком завершения набора вводимой последовательности символов, записываются в одномерный массив c[ ]

из 2 байтов, откуда они затем функцией **write( )** выводятся на экран дисплея.

Запустив программу на выполнение, можно вводить одиночные символы с клавиатуры, завершая ввод каждого из них нажатием на клавишу <Enter>. Результат работы программы может выглядеть так:

```
V  <Enter>
v
w  <Enter>
w
e  <Enter>
e
<Ctrl+C>
```

Первый символ из пары одинаковых символов (v-v; w-w и т. д.) – это символ, введенный с клавиатуры и выведенный системой ввода-вывода на экран (стандартный режим ввода данных с клавиатуры). Второй символ пары выведен на устройство стандартного вывода функцией **write( )**. Программа копирования завершает работу при вводе сигнала прерывания (одновременном нажатии клавиш <Ctrl> и <C>).

Пример 2. Копирование произвольного файла.

Программа получает имена файлов из командной строки при запуске и позволяет копировать произвольные файлы (см. аналог этой программы в §7.1.3).

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(int argc, char *argv[ ])
{
   int  fdin, fdout; /* Дескрипторы файлов */;
   int n; /* Количество прочитанных байтов */
   char buff[BUFSIZ];
   if ( argc != 3)
   {
    printf("Формат вызова программы:");
    printf("\n %s  файл_источник  файл_приемник", argv[0]);
    return 1;
   }
   if ((fdin = open(argv[1], O_RDONLY)) == -1)
   {
```

```
    perror(argv[1]);
    return 1;
  }
  if ((fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC)) == -1)
  {
    perror(argv[2]);
    return 1;
  }
  /* Файлы открыты - можно копировать */
  while ((n = read(fdin, buff, BUFSIZ))  >  0 )
       write(fdout, buff, n);
  return 0;
}
```

Константа **BUFSIZ** (размер буфера для потокового ввода-вывода) определена в заголовочном файле **stdio.h**. Ее значение обычно равно 512 байт.

Пусть программа откомпилирована и создан исполняемый файл с именем, например, copyf.exe. Вызов программы из командной строки будет иметь вид:

```
>copyf    f1.dat    f2.dat     <Enter>
```

где `f1.dat` – файл-источник; `f2.dat` – файл-приемник.

Если ошибок при исполнении программы нет, файл будет скопирован, но никаких сообщений на экране дисплея не появится.

7.2.3. Произвольный доступ к файлу

В приведенных выше программах с функциями ввода-вывода низкого уровня обмен с файлом осуществлялся последовательно. При необходимости файл можно читать на низком уровне и в произвольном порядке. Так же как это делалось при работе с потоками, можно изменять значение указателя текущей позиции чтения/записи в файле. Для этой цели служит функция **lseek( )**. Прототип этой функции имеет следующий вид:

```
long lseek(int fd, long offset, int origin);
```

Функция **lseek( )** изменяет текущую позицию в файле, связанном с дескриптором fd, на новую, определяемую смещением (второй параметр – offset) относительно выбранной точки отсчета (третий параметр – origin).

Точка отсчета задается одной из предопределенных констант, размещенных в заголовочном файле **io.h** или файле **unistd.h** (UNIX):

- **SEEK_SET** (имеет значение 0) – начало файла;
- **SEEK_CUR** (имеет значение 1) – текущая позиция;
- **SEEK_END** (имеет значение 2) – конец файла.

При удачном завершении функция **lseek( )** возвращает новую текущую позицию чтения/записи, представляющую собой смещение от начала файла. Попытка переместиться за пределы файла считается ошибкой. Код ошибки заносится в глобальную переменную **errno**, определенную в заголовочном файле **errno.h**.

Для определения текущей позиции в файле можно использовать функцию **tell( )**, прототип которой имеет следующий вид:

```
long tell(int fd);
```

Приведем примеры использования функции **lseek( )**.

Пример 1. Установка текущей позиции в файле на его начало:

```
lseek(fd, 0L, SEEK_SET);
```

Пример 2. Установка текущей позиции для последующего добавления данных в файл (позиция в конце файла):

```
lseek(fd, 0L, SEEK_END);
```

Пример 3. Модификация записей в существующем файле.

Запись – это последовательность байтов в файле, представляющая сложный структурный элемент в контексте программы, обрабатывающей эти записи. Логическая структура записи может быть любой: массивом, строкой, структурой и т. д. В следующем фрагменте программы предполагается, что все записи имеют одинаковый размер (в байтах) и размещены в файле подряд.

```
/* Прочитать запись в буфер */
read(fd, buff, sizeof( запись ));
/* Вернуть указатель в файле на место,
   с которого начиналось чтение записи */
lseek(fd, -sizeof( запись ), SEEK_CUR);
/* Откорректировать запись в буфере */
. . . . . .
/* Поместить запись на прежнее место */
write(fd, buff, sizeof( запись ));
```

Буфер buff может быть определен как массив символов, достаточный для размещения одной записи.

Контрольные вопросы

1. Каким образом при отсутствии в языке Си средств ввода-вывода реализуются операции ввода-вывода?
2. Дайте определение файла в языке Си.
3. Что означает термин «потоковый ввод-вывод»?
4. Дайте определение потоку.
5. Какие действия можно производить при работе с потоком?
6. Для чего служит структура предопределенного типа **FILE**?
7. Укажите различия между именем файла и указателем типа **FILE**.
8. Укажите имя функции и перечислите ее параметры, необходимые для открытия потока в текстовом режиме.
9. Припомните типичную последовательность операторов, которая используется при открытии файла, связанного с потоком.
10. Какие изменения происходят в структуре типа **FILE** при открытии файла?
11. Перечислите стандартные потоки и функции для работы с ними.
12. Перечислите функции для ввода-вывода строк и форматного ввода-вывода.
13. Какие функции включены в библиотеку Си для работы с файлами на диске?
14. Для чего служат средства позиционирования в потоке?
15. В чем состоит отличие ввода-вывода низкого уровня от потокового ввода-вывода?
16. Перечислите режимы, в которых можно открыть низкоуровневый файл.
17. Что означает параметр открытия файла «права доступа» и как он применяется?
18. Каким образом осуществляется произвольный доступ к низкоуровнему файлу?
19. Что необходимо сделать, для того чтобы изменить режим обработки файла?
20. Как можно проверить, достигнут ли конец файла?
21. В какую точку потока (на начало, в конец) устанавливается указатель чтения/записи в режимах «r», «w» и «a»?
22. Какие точки отсчета выбирают при смещениях по файлу?

Глава 8
ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ

8.1. Схема подготовки программ

Упрощенная схема подготовки программ, существующая во многих операционных системах, изображена на рис. 8.1 (см. также §2.1, рис. 2.1).

Рис. 8.1. *Упрощенная схема подготовки исполняемой программы*

Текст программы, набранный в любом текстовом редакторе и сохраненный в файле в виде последовательности символов, называется ***исходным модулем***. Имена файлов, содержащих тексты программ и функций на языке Си, обычно имеют расширение «с».

Исходные модули обрабатываются компилятором, в результате чего получается промежуточный модуль, называемый ***объектным***. Объектный модуль состоит из двух основных частей: тела модуля, представляющего собой программу в кодах команд конкретной ЭВМ, и заголовка, содержащего внешние имена (имена переменных, используемых в данном модуле, но определенных в других модулях). Эта информация необходима для построения из набора объектных модулей программы или программной системы, готовой к выполнению.

Объектные модули обрабатываются ***компоновщиком***, который строит исполняемую программу, содержащую только команды ЭВМ. В зависимости от операционной системы исполняемую программу могут называть исполняемым файлом, **exe**-модулем, загрузочным модулем и т. п. При построении исполняемой программы к объектным модулям, полученным из исходных модулей, подключаются объектные модули из одной или нескольких библиотек, содержащих, например, объектные модули функций ввода-вывода, вспомогательных библиотечных функций (обработка строк, массивов, выделение динамической памяти и т. д.).

Существование промежуточного объекта (модуля) в цепочке построения, готовой к выполнению программы, позволяет:

- сделать систему программирования гибкой и удобной в эксплуатации за счет предварительной разработки библиотек объектных модулей, содержащих множество вспомогательных функций;
- разрабатывать и использовать собственные библиотеки объектных модулей, содержащие функции, общие для отдельных частей программной системы;
- при модификации программной системы перестраивать только части, подвергшиеся доработке.

На различных этапах разработки программы может применяться разная технология сборки готовой к выполнению программы, например:

- главная функция и все подготовленные программистом вспомогательные функции находятся в одном текстовом файле;
- главная функция и вспомогательные функции находятся в разных текстовых файлах, но собираются в один исходный модуль с помощью препроцессорных команд **#include**;
- главная функция и вспомогательные функции находятся в разных текстовых файлах, транслируются отдельно, и исполняемая программа собирается компоновщиком из объектных модулей;
- вспомогательные функции транслируются отдельно, отлаживаются на тестовых примерах, и из них формируют личную библиотеку объектных модулей, которая применяется для сборки программ.

Даже небольшая программа обычно содержит значительное количество функций и текстов определений и описаний внешних (глобальных) объектов, размещенных в отдельных файлах.

Развитие любой программы приводит к тому, что в итоге ее структура становится сложной, и программисту трудно удержать в голове логическую схему этой структуры. Постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений, становится невозможным.

Современные операционные системы и системы программирования предлагают пользователю средства, в той или иной степени решающие задачу автоматизации разработки программ.

В этой главе рассмотрим средства подготовки программ в операционных системах семейства как UNIX.

8.2. Подготовка программ в операционной системе UNIX

В UNIX для документирования взаимосвязей между модулями и для автоматизации построения программной системы служит утилита **make**. Исходной информацией для утилиты **make** является описание взаимосвязей модулей, на основе которого утилита **make** собирает программу. При дальнейших модификациях текстов отдельных частей программы описание взаимосвязей модулей позволяет утилите **make** перестраивать при сборке программной системы только ту часть системы, которая подверглась модификации.

Схема подготовки исполняемой программы в UNIX приведена на рис. 8.2.

От традиционной для почти всех операционных систем схемы подготовки исполняемых программ схема на рис. 8.2 отличается тем, что в UNIX трансляция исходного модуля ведется на языке

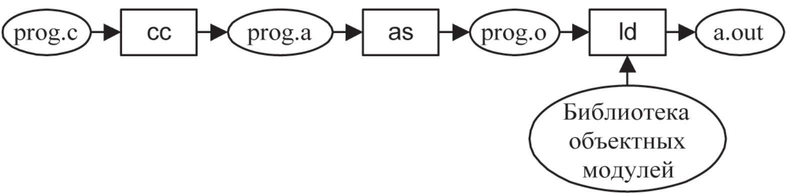


Рис. 8.2. *Схема подготовки исполняемой программы в UNIX:*
**.c – исходный модуль на языке Си;*
**.a – модуль на ассемблере; *.o – объектный модуль;*
a.out – стандартное имя исполняемого модуля
(исполняемой программы); cc – компилятор языка Си;
as – компилятор языка ассемблера;
ld – компоновщик (редактор связей)

ассемблера и исполняемый модуль, если не указано другого, имеет стандартное имя **a.out**. Выбор фиксированного имени объясняется тем, что UNIX в свое время создавалась для удобной разработки и отладки программ. В режиме отладки нет необходимости хранить промежуточные версии исполняемых программ и вполне можно называть их одним именем.

Компилятор Си позволяет за один вызов выполнить всю цепочку преобразования исходного модуля в исполняемую программу. Для задания последовательности преобразований в команде вызова компилятора сначала указываются ключи компилятора и затем параметры компоновщика.

Формат команды **cc**, вызывающей компилятор языка Си, предусматривает задание следующих параметров (в форматах команд операционных систем принято помещать необязательные элементы в квадратные скобки [ ]):

cc [–*ключи*] *исходные_модули* [*ключи_компоновщика*]
[*объектные_модули*] [*библиотеки*]

где

- *ключи* – однобуквенные параметры, задающие режимы работы компилятора. Перед каждым ключом должен стоять знак минус ('–'). После некоторых ключей могут указываться дополнительные параметры. Приведем некоторые (наиболее часто употребляемые) ключи команды **cc**:
  - **–c** – транслировать исходный модуль в объектный;
  - **–p** – провести только препроцессорную обработку исходного модуля;
  - **–s** – транслировать исходный модуль в модуль на языке ассемблера;
  - **–o** *имя_исполняемой_программы* – при необходимости транслировать и задать отличное от стандартного «a.out» имя для исполняемой программы;
- *исходные_модули* – полные имена (с расширением «c») одного или нескольких исходных модулей;
- *объектные_модули* – полные имена (с расширением «o») тех модулей, которые будут использованы при построении исполняемой программы;
- *ключи_компоновщика* – задают режимы работы компоновщика (для нас представляет интерес ключ **–l**, определяющий имя библиотеки объектных модулей):

  –l*библиотека_объектных_модулей*

Примечание

Способ указания имени библиотеки объектных модулей объясняется ниже в разделе «Библиотеки объектных модулей».

Если программа состоит из одного исходного модуля с именем main.c, то для построения исполняемого модуля достаточно выполнить команду (% – «приглашение» от операционной системы):

```
%cc  main.c
```

Исходный модуль будет последовательно преобразован (см. рис. 8.2) в модуль на языке ассемблера, объектный модуль, исполняемый модуль. Исполняемый модуль получит стандартное имя **a.out**. При повторном вызове компилятора языка Си командой **cc** и указании в качестве параметра команды имени другого исходного модуля вновь полученный исполняемый модуль также будет иметь имя **a.out**, но будет соответствовать другому исходному (только что обработанному) модулю.

Для того чтобы определить произвольное имя исполняемого модуля, необходимо в команде вызова компилятора указать ключ **–o** и сразу за ним через пробел задать имя исполняемого модуля:

```
%cc  -o  begin  main.c
```

Построение исполняемого модуля можно провести в два этапа с промежуточным получением объектного модуля:

```
%cc  -c  main.c
%cc  -o  begin  main.o
```

В первой строке применен ключ **–c**, в результате чего процесс обработки исходного модуля прервется, когда будет получен объектный модуль (main.o).

Во второй строке определено имя исполняемого модуля **begin** и в качестве параметра команды **cc** указано имя объектного модуля (main.o), полученного на предыдущем этапе.

8.3. Утилита make

Утилита **make** позволяет документировать взаимосвязи между модулями и освобождает пользователя от рутинной работы по слежению за изменениями в модулях.

Утилита **make** при вызове обновляет (перестраивает) исполняемый модуль, причем транслируются только те функции, в исходные тексты которых были внесены изменения.

Исходными данными для утилиты **make** служит ***файл описаний зависимостей модулей***. Описания зависимостей модулей и соответствующих действий хранятся в так называемом *make*-файле, который по умолчанию называется **makefile** или Makefile. В этом случае он может не указываться при вызове команды **make**. При выборе для *make*-файла (файла зависимостей) произвольного имени часто назначают имена, начинающиеся с заглавной буквы, так как при просмотре содержимого каталога (в UNIX для просмотра используется команда **ls –l**) они будут располагаться в верхней части списка модулей каталога, что облегчает поиск и распознавание *make*-файлов.

Приведем текст файла зависимостей модулей для иллюстративной программы, состоящей из следующих модулей: add_node.c, new_node.c, print.c. В простейшем случае *make*-файл будет выглядеть так:

```
tree:   tree.o    add_node.o    new_node.o    print.o
%cc  -o tree   tree.o   add_node.o   new_node.o   print.o
```

В первой строке первым параметром перед символом ':' указывается имя так называемого ***целевого файла***, а после символа ':' приводится список имен файлов, от которых зависит ***целевой файл***. Эта строка информирует утилиту **make** о необходимости выполнить команду, записанную в следующей строке, если какой-либо из объектных модулей (файлы *.o) имеет более позднюю дату модификации, по сравнению с целевым файлом tree, являющимся в нашем примере исполняемым модулем иллюстративной программы.

Если какой-либо из исходных модулей на препроцессорном уровне включает файл с текстом части программы, то описание зависимостей файлов необходимо дополнить соответствующей строкой по следующему образцу:

```
prog.o  a.o:    def.h
```

Здесь указано, что объектные модули prog.o и a.o зависят от включаемого (директивой **#include**) файла **def.h**.

Помимо описаний зависимостей, утилита **make** использует в процессе своего выполнения набор так называемых встроенных правил.

Одно из основных правил: повторно должны быть откомпилированы лишь те исходные модули, дата последней модификации которых оказалась старше даты создания соответствующего им объектного модуля. Компилируются, конечно, и те исходные модули, для которых соответствующие объектные модули не существуют. Таким образом, утилита **make** осуществляет лишь те минимальные действия, без которых невозможно было бы получить новую версию целевого файла.

Формат файла описаний зависимостей модулей. Файл описаний зависимостей модулей представляет собой текстовый файл, содержащий последовательность строк, которые являются набором *спецификаций взаимозависимостей*, используемых утилитой **make** при ее выполнении. Спецификация взаимозависимостей имеет следующую структуру:

- имя целевого файла;
- последовательность имен файлов, от которых зависит целевой файл;
- последовательность команд UNIX, которая должна быть выполнена, если дата последней модификации хотя бы одного из файлов, от которых зависит целевой файл, старше даты модификации целевого файла.

Формат спецификации взаимозависимостей следующий:

target1[target2 . . .]:[:][depend1 . . .][# . . .]
[(tab)commands][#]

где (квадратные скобки означают необязательность элементов):

- **target ...** – целевые файлы, имена которых разделяются пробелами;
- **:** (или **::**) – разделитель – в первом случае (для ':') последующая последовательность команд UNIX (**commands**) должна содержаться в одной строке файла описаний, а во втором она может располагаться в нескольких строках;
- **depend ...** – последовательность имен файлов, от которых зависит целевой файл (разделяются пробелами);
- **(tab)** – символ «Табуляция», которым предваряются командные строки UNIX;
- commands – команды UNIX, с помощью которых должен быть получен целевой файл;
- **#** – признак начала комментария (комментарий – часть строки от символа '#' до конца строки).

> **Примечание**
>
> Если список имен файлов, от которых зависит целевой файл, не умещается на одной строке, для обозначения переноса части списка на другую строку можно воспользоваться символом '\', например:

```
tree:   tree.o \
        add_node.o \
        new_node.o \
        print.o
        команды UNIX
```

Формат команды make. Команда **make** имеет следующий формат:

make [–f *makefile*][*ключи*][*имена*][*макро_определения*]

Квадратные скобки, выделяющие параметры, означают их необязательность. Параметры в команде отделяются друг от друга пробелами.

Первый параметр с ключом –f задает имя файла зависимостей модулей, если это имя отлично от *makefile* или *Makefile*.

Из множества ключей, определяющих режим работы утилиты **make**, упомянем следующие, необходимые для отладки *make*-файла:

- **–p** – вывести в стандартный поток вывода полный список зависимостей модулей;
- **–i** – игнорировать коды возврата выполненных команд (позволяет отладить сложный *make*-файл);
- **–n** – вывести в стандартный поток вывода строки с командами UNIX, не выполняя их.

Параметр *имена* позволяет задавать имена целевых файлов.

Коротко остановимся на некоторых возможностях утилиты **make**, которые не используются в нашем примере.

Макроопределения. Утилита **make** позволяет использовать при записи спецификаций взаимозависимостей макроопределения. Это дает возможность добиться большей наглядности файла описаний взаимозависимостей и использовать один и тот же файл описаний для различных имен целевых файлов и файлов, от которых зависит целевой файл.

Макроопределения записываются в соответствии со следующим форматом:

имя_макроса = значение

где *имя_макроса* – имя макроса утилиты **make**; *значение* – строка символов, которая подставляется вместо конструкции $(*имя_макроса*) при ее использовании в строках файла взаимозависимостей.

Пример макроопределения:

```
CC=cc
```

После такого макроопределения контекст вида $(CC) будет заменен в строках *make*-файла на cc.

Введем в качестве первой строки в приведенный выше файл описаний взаимозависимостей для целевого файла tree следующее макроопределение:

```
OBJECTS=tree.o   add_node.o   new_node.o   print.o
```

Тогда позже в том же файле для указания объектных модулей, перечисленных справа от знака равенства, можно применять конструкцию $(OBJECTS).

Теперь файл взаимозависимостей для рассматриваемой программы будет выглядеть так:

```
OBJECTS=add_node.o      new_node.o      print.o
tree:  tree.o     $(OBJECTS)
cc   -o   tree     tree.c     $(OBJECTS)
```

Встроенные правила. В процессе выполнения команда **make** использует набор так называемых встроенных правил. Одним из подмножеств этого набора являются правила автоматического установления взаимосвязей между файлами по суффиксам их имен. Например, когда в команде **make** в качестве параметра задано имя файла с суффиксом '.c', автоматически выполняется вызов компилятора языка Си, который строит исполняемый модуль из исходного модуля, находящегося в заданном файле. Таблицу встроенных правил утилиты **make**, соответствующих конкретной реализации UNIX, можно найти в документации по операционной системе. Здесь стандартные встроенные правила мы рассматривать не будем.

Программист может изменить встроенное правило, для чего новый вариант правила необходимо включить в файл описаний зависимостей, то есть во входные данные для **make**.

Простейшее правило, описывающее процедуру подготовки исполняемого модуля из исходного модуля (расширение имени 'c', суффикс имени '.c'), будет выглядеть так:

```
.c:
cc  -o   $@       $*.c
```

где $@ – внутренний макрос утилиты **make**, предназначенный для спецификации полного имени целевого файла; $* – внутренний макрос, определяющий префикс имени файла.

Это правило, включенное в файл описаний зависимостей модулей, заменит внутреннее правило утилиты **make** для суффикса '.c'.

Если теперь ввести команду **make** с указанием имени целевого файла (исполняемой программы)

```
%make     prog
```

то исходный файл prog.c будет скомпилирован, а исполняемый модуль получит имя prog.

С помощью утилиты **make** может быть решено множество задач, связанных с программированием как на языках высокого уровня, так и на командных языках (например, на командном языке UNIX – **csh**), однако основное ее применение – учет взаимозависимостей между исходными текстами модулей в больших программных комплексах.

8.4. Библиотеки объектных модулей

При разработке программной системы объектные модули функций, входящих в ее состав, целесообразно размещать в библиотеках объектных модулей, а в командной строке вызова компилятора указывать эти библиотеки. Использование библиотек позволяет не перечислять непосредственно в командной строке все необходимые для построения исполняемой программы объектные модули. Компоновщик на этапе построения исполняемой программы будет выбирать из библиотеки только те объектные модули функций, которые необходимы.

Стандартные библиотеки. Стандартные библиотеки UNIX хранятся в каталогах **/lib** или **/usr/lib**. Ссылка на библиотеки осуществляется при помощи ключа компоновщика **–l**, который задается в команде **cc** вызова компилятора языка Си после всех ключей и параметров компилятора. Непосредственно за ключом (без пробела) указывается идентификатор библиотеки, например:

```
%cc .   .   .   .   -lx
```

где **x** – часть имени библиотеки (полное имя библиотеки в данном случае libx.a; стандартный префикс для библиотеки «lib»; стандарт-

ное расширение имени для библиотеки '**а**'). Обратите внимание, что libx.a есть название библиотеки, а не отдельного модуля, поэтому суффикс '.a' не обозначает «модуль на ассемблере», как, например, на рис. 8.2.

Стандартная библиотека языка Си просматривается компоновщиком автоматически, то есть указание этой библиотеки в командной строке вызова компилятора Си не требуется. Эта библиотека хранится в файле **/lib/libc.a**.

Библиотека математических функций хранится в файле **/lib/libm.a**, поэтому указание библиотеки в команде вызова компилятора будет таким: **–lm**. Напомним, что ключ **–l**, являющийся ключом компоновщика, необходимо разместить в командной строке вызова компилятора Си после всех других ключей и параметров, так как просмотр библиотеки должен происходить после компиляции, когда становится известно, какие функции из библиотеки нужны для построения исполняемого модуля.

Создание и сопровождение собственных библиотек. В UNIX для создания и сопровождения библиотек объектных модулей служит программа **ar** (archivator – архиватор). Для библиотек в UNIX принят термин «архив»; отсюда в качестве расширения имени библиотеки используется буква 'a'.

При создании библиотеки необходимо иметь в виду, что компоновщик просматривает библиотечный файл только один раз. Если объектный модуль функции ссылается на другие имена из той же библиотеки, то он должен быть расположен в библиотеке до этих объектных модулей. Некоторые версии UNIX содержат программу **ranlib**, которая создает оглавление библиотеки, что позволяет компоновщику **ld** обращаться к элементам библиотеки в произвольном порядке. В некоторых реализациях UNIX функция построения оглавления библиотеки встроена в программы **ar** и **ld**.

Главное назначение программы **ar** – создание и обновление библиотечных файлов, используемых компоновщиком, однако ее можно применять и для любых других, аналогичных целей. Формат команды **ar**:

ar –*ключ* **[***ключ...***]** [*позиционное_имя*] *архив имя. . .*

Параметры в команде отделяются пробелами.

Параметр *позиционное_имя* не является обязательным. Ключей может быть несколько, тогда они записываются без пробелов и только с одним знаком '**–**'.

Параметр *архив* задает имя архивного файла.

Параметр *имя...* является списком имен объектных модулей, которые либо находятся в архиве, либо должны быть туда помещены.

Ключи программы **ar** имеют следующий смысл:

- **–d** – исключить указанные (с помощью параметра *имя...*) файлы из архивного файла;
- **–r** – заменить указанные (параметром *имя...*) файлы в архивном файле. Если вместе с ключом **r** задается необязательный ключ **u**, то заменяются только файлы, имеющие более поздние даты модификации, чем файлы в архиве. Этот же ключ используется для включения в библиотеку новых файлов, то есть файлы, указанные параметром *имя...* и отсутствующие в библиотеке, добавляются в архив. Обычно новые файлы помещаются в конец архива. Если указать имя файла из архива в параметре *позиционное_имя*, то можно новые файлы поместить до или после этого файла. Для этого необходимо соответственно добавить к ключу **–r** необязательные литеры **b** (before) или **a** (after);
- **–t** – вывести в стандартный поток вывода оглавление архивного файла. При задании параметра *имя...* печатается информация только об указанных этим параметром файлах. Этот же ключ используется для включения в библиотеку новых файлов, то есть файлы, указанные параметром *имя...* и отсутствующие в библиотеке, добавляются в архив;
- **–p** – вывести в стандартный поток вывода указанные (параметром *имя...*) файлы из архива;
- **–x** – извлечь из архива указанные (параметром *имя...*) файлы. Если не задан параметр *имя...*, из архива извлекаются все файлы. В любом случае собственно архивный файл не изменяется;
- **–v** – выдавать пояснительные сообщения;
- **–c** – создать архивный файл. Обычно программа **ar** при необходимости создает архивный файл сама. Данный ключ подавляет информационное сообщение, выдаваемое при создании архивного файла.

Для размещения в личной библиотеке объектных модулей функций иллюстративной программы необходимо выполнить следующую команду:

```
%ar   -rv  libtree.a   add_node.o   new_node.o   print.o
```

Здесь libtree.a – имя библиотеки (архивного файла).

Ключ **–r** задает режим замены объектных модулей, хранящихся в библиотеке, на объектные модули, список которых приведен в командной строке вызова архиватора **ar** после имени библиотеки. Так как в момент обработки указанной команды библиотеки не существовало, она будет создана автоматически. Ключ **–v** определяет режим вывода пояснительных сообщений о работе архиватора **ar**.

После занесения объектных модулей в библиотеку необходимо при помощи программы **ranlib** скорректировать (или создать) оглавление библиотеки, выполнив команду

```
%ranlib  libtree.a
```

Рекомендуется ознакомиться с описанием программы **ranlib** до ее использования. Например, справку о ней можно получить так:

```
%man      ranlib
```

Проверим созданную библиотеку, распечатав ее оглавление:

```
%ar   -t   libtree.a
_ _.SYMDEF
add_node.o
new_node.o
print.o
```

Первая строка – вызов программы-архиватора (**ar**). Последние 4 строки – это результат работы команды **ar**. Оглавление библиотеки содержится в разделе библиотеки **_ _.SYMDEF**. Остальные строки – это имена объектных модулей, находящихся в библиотеке. После того как библиотека объектных модулей создана, можно построить исполняемый модуль программы сортировки на основе бинарного дерева с помощью команды:

```
%cc   -o   tree   tree.c   -ltree
```

Здесь транслируется головной модуль (tree.c), объектные модули на этапе компоновки выбираются из библиотеки libtree.a, и строится исполняемая программа с именем tree. Ключ компоновщика **–l** позволяет задать имя библиотеки объектных модулей (полное имя библиотеки: libtree.a).

Предположим, что в исходный текст одной из функций (например, add_node( )) были внесены изменения. Тогда для построения обновленного варианта исполняемой программы необходимо выполнить следующие команды:

```
%cc  -c   add_node.c
%ar  -rv   libtree.a      add_node.o
%cc   -o   tree     tree.o      -ltree
```

На этот раз нет необходимости в повторной трансляции головного модуля (он не подвергался правке), поэтому в последней строке задано имя уже существующего объектного модуля tree.o.

Применим в той же иллюстративной программе утилиту **make** для поддержания личной библиотеки объектных модулей в таком состоянии, когда она всегда содержит объектные модули, полученные из последних версий соответствующих исходных модулей. Выше был приведен пример файла зависимостей утилиты **make**, в котором все объектные модули, используемые для построения исполняемого модуля программы сортировки, указывались в командной строке вызова компилятора. В варианте *make*-файла, ориентированном на применение личной библиотеки объектных модулей, появляются два целевых файла: исполняемая программа сортировки (tree) и библиотека объектных модулей (libtree.a), причем исполняемая программа зависит от содержимого (объектных модулей) библиотеки. Взаимозависимость этих объектов определяется следующим образом:

```
tree: tree.o \
      add_node.o \
      new_node.o \
      print.o \
      libtree.a
libtree.a:  libtree.a(add_node.o) \
            libtree.a(new_node.o) \
            libtree.a(print.o)
```

Обратите внимание, как указаны зависимость целевого файла tree от библиотеки libtree.a и, в свою очередь, ее зависимость от объектных модулей. Объектный модуль из библиотеки libtree.a указывается так:

имя_библиотеки(*имя_модуля*).

Для окончательного оформления *make*-файла используем два встроенных макроса утилиты **make**:

- **$@** – задает имя текущего целевого файла (в нашем случае – последнего (libtree.a));
- **$?** – значение этого макроса вычисляется утилитой **make** – это имена всех модулей, которые подвергались изменениям.

Приведем один из вариантов полного текста *make*-файла (напомним, что его имя по умолчанию – **makefile**), позволяющего обновлять целевые файлы (исполняемую программу и библиотеку объектных модулей) в соответствии с изменениями в исходных текстах функций, входящих в состав иллюстративной программы:

```
tree: tree.o \
      add_node.o \
      new_node.o \
      print.o \
      libtree.a
libtree.a:  libtree.a(add_node.o) \
            libtree.a(new_node.o) \
            libtree.a(print.o)
ar -rv $@   $?
ranlib  $@
cc  -o  tree     tree.c     -ltree
```

Теперь, для того чтобы построить исполняемый модуль, учитывающий все внесенные изменения в любые функции программы сортировки, необходимо просто набрать на клавиатуре команду **make**.

После внесения изменений, например в исходный текст функции new_node( ), и при выполнении команды **make** получим на экране дисплея следующий протокол:

```
%make
cc  -o  -c   new_node
ar  -rv     libtree.a     new_node.o
r  -  new_node.o
ranlib   libtree.a
cc  -o  tree     tree.c   -ltree
```

Строка, следующая за командой вызова архиватора **ar** (r – new_node.o), – это сообщение архиватора о замене объектного модуля new_node.o в библиотеке libtree.a.

Контрольные вопросы

1. Опишите упрощенную схему подготовки программ к выполнению, характерную для большинства операционных систем.
2. Дайте определения исходного и объектного модулей.
3. Объясните необходимость существования объектного модуля.
4. Перечислите возможные технологии сборки готовой к выполнению программы.
5. Для чего служит утилита make в операционных системах семейства UNIX?
6. Опишите схему подготовки исполняемой программы в ОС UNIX.
7. Что является исходными данными для утилиты make?
8. Дайте определение целевого файла.
9. Приведите формат файла описаний зависимостей модулей.
10. Припомните формат команды make.
11. Что позволяет использование библиотек объектных модулей?
12. Опишите процесс трансляции и выполнения программ из командной строки ОС UNIX.

Приложение 1
ТАБЛИЦЫ КОДОВ ASCII

Таблица П1.1. Коды управляющих символов (0–31)

Символ	Код 10	Код 08	Код 16	Клавиши	Значение
nul	0	0	00	^@	Нуль
soh	1	1	01	^A	Начало заголовка
stx	2	2	02	^B	Начало текста
etx	3	3	03	^C	Конец текста
eot	4	4	04	^D	Конец передачи
enq	5	5	05	^E	Запрос
ack	6	6	06	^F	Подтверждение
bel	7	7	07	^G	Сигнал (звонок)
bs	8	10	08	^H	Забой (шаг назад)
ht	9	11	09	^I	Горизонтальная табуляция
lf	10	12	0A	^J	Перевод строки
vt	11	13	0B	^K	Вертикальная табуляция
ff	12	14	0C	^L	Новая страница
cr	13	15	0D	^M	Возврат каретки
so	14	16	0E	^N	Выключить сдвиг
si	15	17	0F	^O	Включить сдвиг
dle	16	20	10	^P	Ключ связи данных
dc1	17	21	11	^Q	Управление устройством 1
dc2	18	22	12	^R	Управление устройством 2
dc3	19	23	13	^S	Управление устройством 3
dc4	20	24	14	^T	Управление устройством 4
nak	21	25	15	^U	Отрицательное подтверждение
syn	22	26	16	^V	Синхронизация
etb	23	27	17	^W	Конец передаваемого блока
can	24	30	18	^X	Отказ
em	25	31	19	^Y	Конец среды
sub	26	32	1A	^Z	Замена
esc	27	33	1B	^[	Ключ

***Таблица П1.1. Коды управляющих символов (0–31)** (окончание)*

Символ	Код 10	Код 08	Код 16	Клавиши	Значение
fs	28	34	1C	^\	Разделитель файлов
gs	29	35	1D	^]	Разделитель группы
rs	30	36	1E	^^	Разделитель записей
us	31	37	1F	^_	Разделитель модулей

В графе «Клавиши» обозначение ^ соответствует нажатию клавиши **Ctrl**, вместе с которой нажимается соответствующая «буквенная» клавиша, формируя код символа.

Таблица П1.2. Символы с кодами 32–127

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
пробел	32	40	20		:	58	72	3A
!	33	41	21		;	59	73	3B
«	34	42	22		<	60	74	3C
#	35	43	23		=	61	75	3D
$	36	44	24		>	62	76	3E
%	37	45	25		?	63	77	3F
&	38	46	26		@	64	100	40
'	39	47	27		A	65	101	41
(	40	50	28		B	66	102	42
)	41	51	29		C	67	103	43
*	42	52	2A		D	68	104	44
+	43	53	2B		E	69	105	45
,	44	54	2C		F	70	106	46
-	45	55	2D		G	71	107	47
.	46	56	2E		H	72	110	48
/	47	57	2F		I	73	111	49
0	48	60	30		J	74	112	4A
1	49	61	31		K	75	113	4B
2	50	62	32		L	76	114	4C
3	51	63	33		M	77	115	4D
4	52	64	34		N	78	116	4E
5	53	65	35		O	79	117	4F
6	54	66	36		P	80	120	50
7	55	67	37		Q	81	121	51
8	56	70	38		R	82	122	52
9	57	71	39		S	83	123	53

Таблица П1.2. Символы с кодами 32–127 *(окончание)*

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
T	84	124	54		j	106	152	6A
U	85	125	55		k	107	153	6B
V	86	126	56		l	108	154	6C
W	87	127	57		m	109	155	6D
X	88	130	58		n	110	156	6E
Y	89	131	59		o	111	157	6F
Z	90	132	5A		p	112	160	70
[	91	133	5B		q	113	161	71
\	92	134	5C		r	114	162	72
]	93	135	5D		s	115	163	73
^	94	136	5E		t	116	164	74
_	95	137	5F		u	117	165	75
`	96	140	60		v	118	155	75
a	97	141	61		w	119	167	77
b	98	142	62		x	120	170	78
c	99	143	63		y	121	171	79
d	100	144	64		z	122	172	7A
e	101	145	65		{	123	173	7B
f	102	146	66		|	124	174	7C
g	103	147	67		}	125	175	7D
h	104	150	68		~	126	176	7E
i	105	151	69		del	127	177	7F

Таблица П1.3. Символы с кодами 128–255 (кодовая таблица 866 – MS-DOS)

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
А	128	200	80		Л	139	213	8B
Б	129	201	81		М	140	214	8C
В	130	202	82		Н	141	215	8D
Г	131	203	83		О	142	216	8E
Д	132	204	84		П	143	217	8F
Е	133	205	85		Р	144	220	90
Ж	134	206	86		С	145	221	91
З	135	207	87		Т	146	222	92
И	136	210	88		У	147	223	93
Й	137	211	89		Ф	148	224	94
К	138	212	8A		Х	149	225	95

Таблица П1.3. Символы с кодами 128–255 (кодовая таблица 866 – MS-DOS) *(продолжение)*

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
Ц	150	226	96		║	186	272	BA
Ч	151	227	97		╗	187	273	BB
Ш	152	230	98		╝	188	274	BC
Щ	153	231	99		╜	189	275	BD
Ъ	154	232	9A		╛	190	276	BE
Ы	155	233	9B		┐	191	277	BF
Ь	156	234	9C		└	192	300	C0
Э	157	235	9D		┴	193	301	C1
Ю	158	236	9E		┬	194	302	C2
Я	159	237	9F		├	195	303	C3
а	160	240	A0		─	196	304	C4
б	161	241	A1		┼	197	305	C5
в	162	242	A2		╞	198	306	C6
г	163	243	A3		╟	199	307	C7
д	164	244	A4		╚	200	310	C8
е	165	245	A5		╔	201	311	C9
ж	166	246	A6		╩	202	312	CA
з	167	247	A7		╦	203	313	CB
и	168	250	A8		╠	204	314	CC
й	169	251	A9		═	205	315	CD
к	170	252	AA		╬	206	316	CE
л	171	253	AB		╧	207	317	CF
м	172	254	AC		╨	208	320	D0
н	173	255	AD		╤	209	321	D1
о	174	256	AE		╥	210	322	D2
п	175	257	AF		╙	211	323	D3
░	176	260	B0		╘	212	324	D4
▒	177	261	B1		╒	213	325	D5
▓	178	262	B2		╓	214	326	D6
│	179	263	B3		╫	215	327	D7
┤	180	264	B4		╪	216	330	D8
╡	181	265	B5		┘	217	331	D9
╢	182	266	B6		┌	218	332	DA
╖	183	267	B7		█	219	333	DB
╕	184	270	B8		▄	220	334	DC
╣	185	271	B9		▌	221	335	DD

Таблица П1.3. Символы с кодами 128–255 (кодовая таблица 866 – MS-DOS) (*окончание*)

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
▐	222	336	DE		я	239	357	EF
▀	223	337	DF		≡ или Ё	240	360	F0
р	224	340	E0		± или ё	241	361	F1
с	225	341	E1		≥	242	362	F2
т	226	342	E2		≤	243	363	F3
у	227	343	E3		⌠	244	364	F4
ф	228	344	E4		⌡	245	365	F5
х	229	345	E5		÷	246	366	F6
ц	230	346	E6		≈	247	367	F7
ч	231	347	E7		°	248	370	F8
ш	232	350	E8		·	249	371	F9
щ	233	351	E9		·	250	372	FA
ъ	234	352	EA		√	251	373	FB
ы	235	353	EB		^{n}	252	374	FC
ь	236	354	EC		^{2}	253	375	FD
э	237	355	ED		■	254	376	FE
ю	238	356	EE			255	377	FF

Таблица П1.4. Символы с кодами 128–255 (кодовая таблица 1251 – MS Windows)

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
Ђ	128	200	80		Џ	143	217	8F
Ѓ	129	201	81		ђ	144	220	90
‚	130	202	82		‘	145	221	91
ѓ	131	203	83		’	146	222	92
„	132	204	84		“	147	223	93
…	133	205	85		”	148	224	94
†	134	206	86		•	149	225	95
‡	135	207	87		–	150	226	96
€	136	210	88		—	151	227	97
‰	137	211	89		˜	152	230	98
Љ	138	212	8A		™	153	231	99
‹	139	213	8B		љ	154	232	9A
Њ	140	214	8C		›	155	233	9B
Ќ	141	215	8D		њ	156	234	9C
Ћ	142	216	8E		ќ	157	235	9D

Таблица П1.4. Символы с кодами 128–255 (кодовая таблица 1251 – MS Windows) *(продолжение)*

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
ћ	158	236	9E		В	194	302	C2
џ	159	237	9F		Г	195	303	C3
	160	240	A0		Д	196	304	C4
Ў	161	241	A1		Е	197	305	C5
ў	162	242	A2		Ж	198	306	C6
Ј	163	243	A3		З	199	307	C7
¤	164	244	A4		И	200	310	C8
Ґ	165	245	A5		Й	201	311	C9
¦	166	246	A6		К	202	312	CA
§	167	247	A7		Л	203	313	CB
Ё	168	250	A8		М	204	314	CC
©	169	251	A9		Н	205	315	CD
Є	170	252	AA		О	206	316	CE
«	171	253	AB		П	207	317	CF
¬	172	254	AC		Р	208	320	D0
-	173	255	AD		С	209	321	D1
®	174	256	AE		Т	210	322	D2
Ї	175	257	AF		У	211	323	D3
°	176	260	B0		Ф	212	324	D4
±	177	261	B1		Х	213	325	D5
І	178	262	B2		Ц	214	326	D6
і	179	263	B3		Ч	215	327	D7
ґ	180	264	B4		Ш	216	330	D8
µ	181	265	B5		Щ	217	331	D9
¶	182	266	B6		Ъ	218	332	DA
·	183	267	B7		Ы	219	333	DB
ё	184	270	B8		Ь	220	334	DC
№	185	271	B9		Э	221	335	DD
є	186	272	BA		Ю	222	336	DE
»	187	273	BB		Я	223	337	DF
ј	188	274	BC		а	224	340	E0
Ѕ	189	275	BD		б	225	341	E1
ѕ	190	276	BE		в	226	342	E2
ї	191	277	BF		г	227	343	E3
А	192	300	C0		д	228	344	E4
Б	193	301	C1		е	229	345	E5

Таблица П1.4. Символы с кодами 128–255 (кодовая таблица 1251 – MS Windows) *(окончание)*

Символ	Код 10	Код 08	Код 16		Символ	Код 10	Код 08	Код 16
ж	230	346	E6		у	243	363	F3
з	231	347	E7		ф	244	364	F4
и	232	350	E8		х	245	365	F5
й	233	351	E9		ц	246	366	F6
к	234	352	EA		ч	247	367	F7
л	235	353	EB		ш	248	370	F8
м	236	354	EC		щ	249	371	F9
н	237	355	ED		ъ	250	372	FA
о	238	356	EE		ы	251	373	FB
п	239	357	EF		ь	252	374	FC
р	240	360	F0		э	253	375	FD
с	241	361	F1		ю	254	376	FE
т	242	362	F2		я	255	377	FF

Приложение 2 Константы предельных значений

Предельные значения вводятся каждой реализацией для данных целочисленных типов и арифметических значений, представляемых в форме с плавающей точкой. Предельные значения определяются набором констант, названия (имена) которых стандартизированы и не зависят от реализаций. Ниже приводятся обозначения констант и их минимальные (по абсолютной величине) допустимые стандартом ANSI значения. В конкретных реализациях абсолютные значения констант могут превышать значения, приведенные в таблицах.

Таблица П2.1. Предельные значения для целочисленных типов – файл limits.h

Имя константы	Значение	Смысл
CHAR_BIT	8	Число битов в байте
SCHAR_MIN	-128	Минимальное значение **signed char**
SCHAR_MAX	127	Максимальное значение **signed char**
UCHAR_MAX	255	Максимальное значение **unsigned char**
CHAR_MIN	'0' SCHAR_MIN	Минимальное значение для **char**
CHAR_MAX	UCHAR_MAX SCHAR_MAX	Максимальное значение для **char**
MB_LEN_MAX	1	Минимальное число байтов в многобайтовом символе
SHRT_MIN	-32768	Минимальное значение для **short**
SHRT_MAX	32767	Максимальное значение для **short**
USHRT_MAX	65535	Максимальное значение **unsigned short**
INT_MIN	-32768	Минимальное значение для **int**
INT_MAX	32767	Максимальное значение для **int**
UINT_MAX	65535	Максимальное значение **unsigned int**
LONG_MIN	-2147483648	Минимальное значение для **long**
LONG_MAX	2147483647	Максимальное значение для **long**
ULONG_MAX	4294967295	Максимальное значение **unsigned long**

В табл. П2.2 префикс **FLT_** соответствует типу **float**; для типа **double** используется префикс **DBL_**.

Таблица П2.2. Константы для вещественных типов – файл float.h

Имя константы	Значение	Смысл
FLT_RADIX	2	Основание экспоненциального представления, например: 2, 16
FLT_DIG	6	Количество верных десятичных цифр
FLT_EPSILON	1E-5	Минимальное x, такое, что $1.0 + x \neq 1.0$ (1.192093E–07)
FLT_MANT_DIG	24	Количество цифр в мантиссе по основанию FLT_RADIX
FLT_MAX	1E+37	Максимальное число с плавающей точкой (3.402823E+38)
FLT_MAX_EXP	128	Максимальное n, такое, что $\mathrm{FLT_RADIX}^n - 1$ представимо в виде числа типа **float**
FLT_MAX_10_EXP	38	Максимальное целое n, такое, что 10^n представимо как **float**
FLT_MIN	1E-37	Минимальное нормализованное число с плавающей точкой типа **float** (1.175494E–38)
FLT_MIN_EXP	-125	Минимальное n, такое, что 10^n представимо в виде нормализованного числа
FLT_MIN_10_EXP	38	Максимальное целое n, такое, что 10^n представимо как **float**
DBL_DIG	10	Количество верных десятичных цифр для типа **double**
DBL_EPSILON	1E-16	Минимальное x, такое, что $1.0 + x \neq 1.0$, где x принадлежит типу **double** (2.220446E–16)
DBL_MANT_DIG	53	Количество цифр по основанию **FLT_RADIX** в мантиссе для чисел типа **double**
DBL_MAX	1E+308	Максимальное число с плавающей точкой типа **double** (1.797693E+308)
DBL_MAX_EXP	1024	Максимальное n, такое, что $\mathrm{FLT_RADIX}^n - 1$ представимо в виде числа типа **double**
DBL_MAX_10_EXP	308	Максимальное целое n, такое, что 10^n представимо как **double**
DBL_MIN	1E-308	Минимальное нормализованное число с плавающей точкой типа **double** (2.225074E–308)
DBL_MIN_EXP	-1021	Минимальное n, такое, что 10^n представимо в виде нормализованного числа типа **double**
DBL_MIN_10_EXP	-307	Минимальное отрицательное целое n, такое, что 10^n – в области определения чисел типа **double**

В скобках для некоторых констант приведены значения из реализации Borland 3.1 C++.

Приложение 3
Стандартная библиотека функций языка Си

Таблица П3.1. Математические функции (файл math.h)

Функция	Прототип и краткое описание действий
abs	`int abs(int i);` Возвращает абсолютное значение целого аргумента `i`
acos	`double acos(double x);` Функция арккосинуса. Значение аргумента должно находиться в диапазоне от –1 до +1
asin	`double asin(double x);` Функция арксинуса. Значение аргумента должно находиться в диапазоне от –1 до +1
atan	`double atan(double x);` Функция арктангенса
atan2	`double atan2(double y, double x);` Функция арктангенса от значения y/x
cabs	`double cabs(struct complex znum);` Вычисляет абсолютное значение комплексного числа `znum`. Определение структуры (типа) `complex` – в файле `math.h`
cos	`double cos(double x);` Функция косинуса. Угол (аргумент) задается в радианах
cosh	`double cosh(double x);` Возвращает значение гиперболического косинуса x
exp	`double exp(double x);` Вычисляет значение e^x (экспоненциальная функция)
fabs	`double fabs(double x);` Возвращает абсолютное значение вещественного аргумента x двойной точности
floor	`double floor(double x);` Находит наибольшее целое, не превышающее значения x. Возвращает его в форме **double**
fmod	`double fmod(double x, double y);` Возвращает остаток от деления нацело **x** на **y**

К автору: какой символ должен быть в выражении v ж 2e?

Таблица П3.1. Математические функции (файл math.h) *(окончание)*

Функция	Прототип и краткое описание действий
hypot	`double hypot(double x, double y);` Вычисляет гипотенузу **z** прямоугольного треугольника по значениям катетов x, y ($z^2 = x^2 + y^2$)
labs	`long labs(long x);` Возвращает абсолютное значение целого аргумента `long x`
ldexp	`double ldexp(double v, int e);` Возвращает значение выражения v ж 2e
log	`double log(double x);` Возвращает значение натурального логарифма (`ln x`)
log10	`double log10(double x);` Возвращает значение десятичного логарифма ($\log_{10} x$)
poly	`double poly(double x, int n, double c[ ]);` Вычисляет значение полинома: $c[n]x^n + c[n - 1]x^{n-1} + ... + c[1]x + c[0]$
pow	`double pow(double x, double y);` Возвращает значение x^y, то есть x в степени y
pow10	`double pow10(int p);` Возвращает значение 10^p
sin	`double sin(double x);` Функция синуса. Угол (аргумент) задается в радианах
sinh	`double sinh(double x);` Возвращает значение гиперболического синуса для x
sqrt	`double sqrt(double x);` Возвращает положительное значение квадратного корня $\sqrt{x}$
tan	`double tan(double x);` Функция тангенса. Угол (аргумент) задается в радианах
tanh	`double tanh(double x);` Возвращает значение гиперболического тангенса для x

Таблица П3.2. Функции и макросы проверки и преобразования символов (файл ctype.h)

Функция	Прототип и краткое описание действий
isalnum	`int isalnum(int c);` Дает значение не нуль, если **c** – код буквы или цифры (A–Z, a–z, 0–9), и нуль – в противном случае
isalpha	`int isalpha(int c);` Дает значение не нуль, если c – код буквы (A–Z, a–z), и нуль – в противном случае

Таблица П3.2. Функции и макросы проверки и преобразования символов (файл ctype.h) *(окончание)*

Функция	Прототип и краткое описание действий
isascii	`int isascii(int c);` Дает значение не нуль, если **c** есть код ASCII, то есть принимает значение от 0 до 127, в противном случае – нуль
iscntrl	`int iscntrl(int c);` Дает значение не нуль, если **c** – управляющий символ с кодами `0x00–0x01F` или `0x7F`, и нуль – в противном случае
isdigit	`int isdigit(int c);` Дает значение не нуль, если **c** – цифра (0–9) в коде ASCII, и нуль – в противном случае
isgraph	`int isgraph(int c);` Дает значение не нуль, если **c** – видимый (изображаемый) символ с кодом (`0x21–0x7E`), и нуль – в противном случае
islower	`int islower(int c);` Дает значение не нуль, если **c** – код символа на нижнем регистре (`a–z`), и нуль – в противном случае
isprint	`int isprint(int c);` Дает значение не нуль,если **c** – печатный символ с кодом (`0x20–0x7E`), и нуль – в противном случае
ispunct	`int ispunct(int c);` Дает значение не нуль, если `c` – символ-разделитель соответствует `iscntrl` или `isspace`, и нуль – в противном случае
isspace	`int isspace(int c);` Дает значение не нуль, если c – обобщенный пробел: пробел, символ табуляции, символ новой строки или новой страницы, символ возврата каретки (0x09–0x0D, 0x20) и нуль – в противном случае
isupper	`int isupper(int c);` Дает значение не нуль, если `c` – код символа на верхнем регистре (`A–Z`), и нуль – в противном случае
isxdigit	`int isxdigit(int c);` Дает значение не нуль, если `c` – код шестнадцатеричной цифры (`0–9`, `A–F`, `a–f`), и нуль – в противном случае
toascii	`int toascii(int c);` Преобразует целое число `c` в символ кода ASCII, обнуляя все биты, кроме младших семи. Результат от 0 до 127
tolower	`int tolower(int c);` Преобразует код буквы `c` к нижнему регистру, остальные коды не изменяются
toupper	`int toupper(int c);` Преобразует код буквы `c` к верхнему регистру, остальные коды не изменяются

Таблица П3.3. Функции ввода-вывода для стандартных файлов (файл stdio.h)

Функция	Прототип и краткое описание действий
getch	`int getch(void);` Считывает один символ с клавиатуры без отображения на экране
getchar	`int getchar(void);` Считывает очередной символ из стандартного входного потока (**stdin**)
gets	`char *gets(char *s);` Считывает строку s из стандартного входного потока (**stdin**)
printf	`int printf(const char *format [, argument,...]);` Функция форматированного вывода в стандартный поток **stdout**
putchar	`int putchar(int c);` Записывает символ c в стандартный поток вывода (**stdout**)
puts	`int puts(const char *s);` Записывает строку s в стандартный поток вывода (**stdout**)
scanf	`int scanf(const char *format[, address, ...]);` Функция форматированного ввода из стандартного потока **stdin**
sprintf	`int sprintf(char *s, const char *format[, argument, ...]);` Функция форматированной записи в строку s
sscanf	`int sscanf(const char *s, const char *format[, address, ...]);` Функция форматированного чтения из строки s
ungetch	`int ungetch(int c);` Возвращает символ c в стандартный поток ввода **stdin**, заставляя его быть следующим считываемым символом

Таблица П3.4. Функции для работы со строками (файлы string.h, stdlib.h)

Функция	Прототип и краткое описание действий
atof	`double atof(const char *str);` Преобразует строку `str` в вещественное число типа **double**
atoi	`int atoi(const char *str);` Преобразует строку `str` в целое число типа **int**
atol	`long atol(const char *str);` Преобразует строку `str` в целое число типа **long**
itoa	`char *itoa(int v, char *str, int baz);` Преобразует целое `v` в строку `str`. При изображении числа используется основание `baz` ($2 \le baz \le 36$). Для отрицательного числа и `baz = 10` первый символ «минус» (`-`)
ltoa	`char *ltoa(long v, char *str, int baz);` Преобразует длинное целое `v` в строку `str`. При изображении числа используется основание `baz` ($2 \le baz \le 36$)

Таблица П3.4. Функции для работы со строками (файлы string.h, stdlib.h) *(продолжение)*

Функция	Прототип и краткое описание действий
strcat	`char *strcat(char *sp, const char *si);` Приписывает строку `si` к строке `sp` (конкатенация строк)
strchr	`char *strchr(const char *str, int c);` Ищет в строке `str` первое вхождение символа `c`
strcmp	`int strcmp(const char *str1, const char *str2);` Сравнивает строки `str1` и `str2`. Результат отрицателен, если `str1 < str2`; равен нулю, если `str1 == str2`, и положителен, если `str1 > str2` (сравнение беззнаковое)
strcpy	`char *strcpy(char *sp, const char *si);` Копирует байты строки `si` в строку `sp`
strcspn	`int strcspn(const char *str1, const char *str2);` Определяет длину первого сегмента строки `str1`, содержащего символы, не входящие во множество символов строки `str2`
strdup	`char *strdup(const char *str);` Выделяет память и переносит в нее копию строки `str`
strlen	`unsigned strlen(const char *str);` Вычисляет длину строки `str`
strlwr	`char *strlwr(char *str);` Преобразует буквы верхнего регистра в строке в соответствующие буквы нижнего регистра
strncat	`char *strncat(char *sp, const char *si, int kol);` Приписывает **`kol`** символов строки `si` к строке `sp` (конкатенация)
strncmp	`int strncmp(const char *str1, const char *str2, int kol);` Сравнивает части строк `str1` и `str2`, причем рассматриваются первые `kol` символов. Результат отрицателен, если `str1 < str2`; равен нулю, если `str1 == str2`, и положителен, если `str1 > str2`
strncpy	`char *strncpy(char *sp, const char *si, int kol);` Копирует `kol` символов строки `si` в строку `sp` («хвост» отбрасывается или дополняется пробелами)
strnicmp	`int strnicmp(char *str1, const char *str2, int kol);` Сравнивает не более `kol` символов строки `str1` и строки `str2`, не делая различия регистров (см. функцию `strncmp()`)
strnset	`char *strnset(char *str, int c, int kol);` Заменяет первые `kol` символов строки `str` символом `c`
strpbrk	`char *strpbrk(const char *str1,const char *str2);` Ищет в строке `str1` первое появление любого из множества символов, входящих в строку `str2`
strrchr	`char *strrchr(const char *str, int c);` Ищет в строке **`str`** последнее вхождение символа `c`

Таблица П3.4. Функции для работы со строками (файлы string.h, stdlib.h) *(продолжение)*

Функция	Прототип и краткое описание действий
strset	`char *strset(char *str, int c);` Заполняет строку `str` заданным символом `c`
strspn	`int strspn(const char *str1, const char *str2);` Определяет длину первого сегмента строки `str1`, содержащего только символы, из множества символов строки `str2`
strstr	`char *strstr(const char *str1, const char *str2);` Ищет в строке `str1` подстроку `str2`. Возвращает указатель на тот элемент в строке `str1`, с которого начинается подстрока `str2`
strtod	`double strtod(const char *str, char **endptr);` **Преобразует символьную строку** `str` в число двойной точности. Если `endptr` не равен `NULL`, то `*endptr` возвращается как указатель на символ, при достижении которого прекращено чтение строки `str`
strtok	`char *strtok(char *str1, const char *str2);` Ищет в строке `str1` лексемы, выделенные символами из второй строки
strtol	`long strtol(const char *str, char **endptr, int baz);` Преобразует символьную строку `str` к значению «длинное число» с основанием `baz` ($2 \le baz \le 36$). Если `endptr` не равен `NULL`, то `*endptr` возвращается как указатель на символ, при достижении которого прекращено чтение строки `str`
strupr	`char *strupr(char *str);` Преобразует буквы нижнего регистра в строке `str` в буквы верхнего регистра
ultoa	`char *ultoa(unsigned long v, char *str, int baz);` Преобразует беззнаковое длинное целое `v` в строку `str`

Таблица П3.5. Функции для выделения и освобождения памяти (файлы alloc.h, stdlib.h)

Функция	Прототип и краткое описание действий
calloc	`void *calloc(unsigned n, unsigned m);` Возвращает указатель на начало области динамически распределенной памяти для размещения `n` элементов по `m` байт каждый. При неудачном завершении возвращает значение `NULL`
coreleft	`unsigned coreleft(void);` – для схем распределения памяти в Turbo C: **tiny**, **small**, **medium**. `unsigned long coreleft(void);` – для других схем распределения памяти. Возвращает значение объема неиспользованной памяти. Функция уникальна для Turbo C, где приняты упомянутые схемы распределения памяти

Таблица П3.5. Функции для выделения и освобождения памяти (файлы alloc.h, stdlib.h) (окончание)

Функция	Прототип и краткое описание действий
free	`void free(void *bl);` Освобождает ранее выделенный блок динамически распределяемой памяти с адресом первого байта `bl`
malloc	`void *malloc(unsigned s);` Возвращает указатель на блок динамически распределяемой памяти длиной s байт. При неудачном завершении возвращает значение `NULL`
realloc	`void *realloc(void *bl, unsigned ns);`
	Изменяет размер ранее выделенной динамической памяти с адресом начала `bl` до размера **ns** байт. Если `bl` равен `NULL`, то функция выполняется как `malloc()`. При неудачном завершении возвращает значение `NULL`

Таблица П3.6. Функции для работы с терминалом в текстовом режиме (файл conio.h)

Функция	Прототип и краткое описание действий
clreol	`void clreol(void);` Стирает символы от позиции курсора до конца строки в текстовом окне
clrscr	`void clrscr(void);` Очищает экран
cgets	`char *cgets(char *str);` Выводит на экран строку `str`
cprintf	`int cprintf(const char *format[, argument,...]);` Выводит форматированную строку в текстовое окно, созданное функцией `window()`
cputs	`int cputs(char *str);` Помещает в символьный массив `str` строку с клавиатуры (консоли)
cscanf	`int cscanf(const char *format[, address, ...]);` Функция форматированного ввода, которая используется при работе с терминалом в текстовом режиме
delline	`void delline(void);` Удаляет строку в текстовом окне (где находится курсор)
gotoxy	`void gotoxy(int x, int y);` Перемещает курсор в позицию текстового окна с координатами `(x, y)`
highvideo	`void highvideo(void);` Повышает яркость символов, выводимых на экран после ее вызова

Таблица П3.6. Функции для работы с терминалом в текстовом режиме (файл conio.h) *(окончание)*

Функция	Прототип и краткое описание действий
movetext	`int movetext(int x0,int y0,int x1,int y2,int x,int y);` Переносит текстовое окно в область экрана, правый верхний угол которой имеет координаты (`x, y`). Координаты угловых точек окна – (`x0, y0`), (`x1, y1`)
normvideo	`void normvideo(void);` Устанавливает нормальную яркость выводимых на экран символов
textattr	`void textattr(int newattr);` Устанавливает атрибуты (фон, цвет) символов, выводимых на экран
textbackground	`void textbackground(int c);` Устанавливает цвет фона по значению параметра `c`
textcolor	`void textcolor(int c);` Устанавливает цвет символов по значению параметра `c`
textmode	`void textmode(int m);` Переводит экран в текстовый режим по значению параметра `m`
wherex	`int wherex(void);` Возвращает значение горизонтальной координаты курсора
wherey	`int wherey(void);` Возвращает значение вертикальной координаты курсора
window	`void window(int x0,int y0,int x1,int y1);` Создает текстовое окно по координатам угловых точек (`x0, y0`), (`x1, y1`)

Функции из табл. П3.6 поддерживаются только на IBM PC и совместимых с ним компьютерах.

Таблица П3.7. Специальные функции

Функция	Прототип и краткое описание действий	Местонахождение прототипа
delay	`void delay(unsigned x);` Приостанавливает выполнение программы на x мсек	`dos.h`
kbhit	`int kbhit(void);` Возвращает ненулевое целое, если в буфере клавиатуры присутствуют коды нажатия клавиш, в противном случае – нулевое значение	`conio.h`

Таблица П3.7. Специальные функции *(продолжение)*

Функция	Прототип и краткое описание действий	Местонахождение прототипа
memcmp	`int memcmp(const void *s1, const void *s2, unsigned n);` Сравнивает посимвольно две области памяти `s1` и `s2` длиной `n` байт. Возвращает значение меньше нуля, если `s1 < s2`, нуль, если `s1 == s2`, и больше нуля, если `s1 > s2`	`mem.h`
memcpy	`void *memcpy(void *p, const void *i, unsigned n);` Копирует блок длиной `n` байт из области памяти `i` в область памяти `p`	`mem.h`
memicmp	`int memicmp (const void *s1, const void *s2, unsigned n);` Подобна `memcmp`, за тем исключением, что игнорируются различия между буквами верхнего и нижнего регистра	`mem.h`
memmove	`void *memmove (void *dest, const void *src, int n);` Копирует блок длиной `n` байтов из `src` в `dest`. Возвращает указатель `dest`	`mem.h`
memset	`void *memset (void *s, int c, unsigned n);` Записывает во все байты области памяти `s` значение `c`. Длина области `s` равна `n` байт	`mem.h`
nosound	`void nosound(void);` Прекращает подачу звукового сигнала, начатую функцией `sound()`	`dos.h`
peek	`int peek(unsigned s, unsigned c);` Возвращает целое значение (слово), записанное в сегменте `s` со смещением `c`	`dos.h`
peekb	`char peekb(unsigned s, unsigned c);` Возвращает один байт, записанный в сегменте `s` со смещением `c`, то есть по адресу `s:c`	`dos.h`
poke	`void poke(unsigned s, unsigned c, int v);` Помещает значение `v` в слово сегмента `s` со смещением `c`, то есть по адресу `s:c`	`dos.h`
pokeb	`void pokeb(unsigned s, unsigned c, char v);` То же, что и `poke`, но помещает один байт `v` по адресу `s:c`	`dos.h`
rand	`int rand(void);` Возвращает псевдослучайное целое число из диапазона 0–215–1), может использовать функцию `srand()`	`stdlib.h`

Таблица П3.7. ***Специальные функции*** *(окончание)*

Функция	Прототип и краткое описание действий	Местонахождение прототипа
signal	`int signal(int sig);` Вызывает программный сигнал с номером `sig`. Используется для обработки исключительных ситуаций в языке Си	`signal.h`
sound	`void sound(unsigned f);` Вызывает звуковой сигнал с частотой `f` Гц	`dos.h`
srand	`void srand(unsigned seed);` Функция инициализации генератора случайных чисел (`rand`); `seed` – любое беззнаковое целое число	`stdlib.h`

Приложение 4
МОДЕЛИ ПРЕДСТАВЛЕНИЯ ЧИСЕЛ НА РАЗЛИЧНЫХ КОМПЬЮТЕРНЫХ ПЛАТФОРМАХ

При разработке программных продуктов необходимо учитывать платформу (аппаратная и программная составляющие), в среде которой реализован компилятор с языка Си, и платформу, в среде которой будет эксплуатироваться созданный программный продукт, так как платформа существенным образом влияет на размерность данных различных типов.

Основными характеристиками платформы являются:

- тип процессора;
- разрядность процессора;
- разрядность приложения (его ориентация на аппаратную платформу);
- размерность данных (количество битов, отводимое для представления данных).

В табл. 1 приведены сведения о разрядности процессоров, приложений и данных.

Таблица 1. Характеристики различных платформ

Процессоры
4 бита 8 бит 12 бит 16 бит 18 бит 24 бита 31 бит 32 бита 36 бит 48 бит **64 бита** 128 бит
Приложения
4 бита 8 бит 16 бит 32 бита 64 бита 128 бит
Размерность данных
4 бита 8 бит 16 бит 32 бита 64 бита 128 бит
полубайт байт октет слово «двойное слово» «суперслово»

В настоящее время наиболее распространены 32-разрядная архитектура процессоров и 32-разрядные приложения. Однако в ближайшем будущем 64-битовые системы могут стать весьма распространенными, и это обстоятельство необходимо учитывать при разработке программных систем.

Точность и диапазон представления чисел зависят от количества битов, отведенных для запоминания этого числа. В табл. 2 приведен пример изменения диапазона целых чисел в зависимости от числа бит представления числа.

Таблица 2. Зависимость значения целого числа от количества битов, отведенное на его представление

Максимальные и минимальные значения для целых				
	8 бит	**16 бит**	**32 бита**	**64 бита**
Минимальное значение (со знаком)	–128	–32768	–2147483648	–9223372036854775808
Максимальное значение (со знаком)	127	32767	2147483647	–9223372036854775807
Минимальное значение (без знака)	0	0	0	0
Максимальное значение (без знака)	255	65535	4294967295	18446744073709551615

Существуют так называемые «модели данных», задающие правила представления чисел для различных платформ.

1. Модель ILP32. Типы int, long и указатель имеют длину 32 бита. Эта модель является стандартной для большинства 32-разрядных компьютеров.
2. Модель ILP32LL. Типы int, long и указатель имеют длину 32 бита. Стандартом C99 введен новый тип - long long, имеющий длину 64 бита. Этот тип позволяет расширить длину представления чисел до минимального значения в 64 бита, но при этом не затрагиваются фундаментальные положения языка.
3. Модель LP64. Тип long и указатель имеют длину 64 бита.
4. Модель ILP64. Типы int, long и указатель имеют длину 64 бита. Тип int получает длину в 64 бита, что является весьма важным для языка.
5. Модель LLP64 – указатели и новый тип long long имеют длину 64 бита. Типы int и long остаются 32-битовыми типами.

При переносе ПО из среды, поддерживающей модель данных ILP32, в среду, где обеспечивается модель ILP64, происходят фун-

даментальные изменения, так как указатели и данные типа int не являются более одного типа, а указатели и данные типа long являются 64-битовыми и выровнены на границу 64 бита. Эти различия могут потенциально повлиять на следующие операции и объекты:

- округление данных (data truncation);
- работа с указателями;
- преобразование операндов к одному типу (перед выполнением операции);
- выравнивание данных и разделение данных;
- работа с константами;
- сдвиг данных (bit shifts) и битовые маски;
- битовые поля;
- перечислимые типы.

В этом учебнике мы будем ориентироваться на 32-разрядную архитектуру процессоров и 32-разрядные приложения как самые распространенные в настоящее время. В табл. 3 приведены типичные длины и диапазоны целых чисел для этой архитектуры.

Таблица 3. Размеры и диапазоны целых чисел на 32-битовой платформе

Типичные размеры и диапазоны для целых на 32-битовой платформе			
Тип	**Ширина (в битах)**	**Минимальное значение**	**Максимальное значение**
signed char	8	–128	
unsigned char	8	0	
short	16	–32,768 32,767	32,767
unsigned short	16	0	65,535
Int	32	–2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
long	32	–2,147,483,648 2,	2,147,483,647
unsigned long	32	0	4,294,967,295
long long	64	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

Литература

1. Керниган Б., Ритчи Д., Фьюер А. Язык программирования Си: задачи по языку Си. – М.: Финансы и статистика, 1985. – 279 с.
2. Керниган Б., Ритчи Д. Язык программирования Си. – М.: Финансы и статистика, 1992. – 272 с.
3. Болски М.И. Язык программирования Си: справочник. – М.: Радио и связь, 1988. – 96 с.
4. Хэнкок Л., Кригер М. Введение в программирование на языке Си. – М.: Радио и связь, 1986. – 192 с.
5. Жешке Р. Толковый словарь стандарта языка Си. – СПб.: Питер, 1994. – 222 с.
6. Уэйт М., Прата С., Мартин Д. Язык Си: руководство для начинающих. – М.: Мир, 1988. – 512 с.
7. Банахан М., Раттер Э. Введение в операционную систему UNIX. – М.: Радио и связь, 1985. – 344 с.
8. Белецкий Я. Энциклопедия языка Си. – М.: Мир, 1992. – 687 с.
9. Подбельский В.В., Фомин С.С. Программирование на языке Си. – 2-е изд. – М.: Финансы и статистика, 1999. – 600 с.
10. Подбельский В.В. Практикум по программированию на языке Си. – М.: Финансы и статистика, 2004. – 576 с.
11. Джехани Н. Программирование на языке Си. – М.: Радио и связь, 1988. – 272 с.
12. Юлин В.А., Булатова И.Р. Приглашение к Си. – Минск: Вышейшая школа, 1990. – 224 с.
13. Уинер Р. Язык Турбо Си. – М.: Мир, 1991. – 380 с.
14. Романовская Л.М., Русс Т.В., Свитковский С.Г. Программирование в среде Си для ПЭВМ ЕС.– М.: Финансы и статистика, 1991. – 352 с.
15. Трой Д. Программирование на языке Си для персонального компьютера IBM PC. – М.: Радио и связь, 1991. – 430 с.
16. Бочков С.О., Субботин Д.М. Язык программирования Си для персонального компьютера. – М.: СП «Диалог», 1990. – 384 с.
17. Дерк Л. С и С++: справочник / пер. с нем. – М.: Восточная книжная компания, 1997. – 592 с.
18. Тондо К., Гимпел С. Язык Си: книга ответов. – М.: Финансы и статистика, 1994. – 160 с.

Предметный указатель

A – Z

ANSI-стандарт, 138
ESC-sequence,
  см. Эскейп-последовательность
l-значение, 32, 35
MS-DOS, 326, 352
UNIX, 6, 8, 156, 163, 283, 322, 336, 343
Windows, 6, 235, 324, 354

А

Абстрактный тип, 264
Агрегирующий тип, 240
Аддитивная операция, 31, 33, 45, 66, 145, 146
Адрес,
  см. Операция получения адреса
Алфавит языка Си, 11
Аргумент макроса, 115, 130, 134, 135
Аргумент функции, 37, 38, 43, 61, 62, 67, 73, 95, 98, 178, 181, 204, 294
Арифметический тип,
  см. Тип арифметический
Арифметическое выражение,
  см. Выражение арифметическое
Арифметическое преобразование,
  см. Операция преобразования

Б

Бинарные операции, 35
Бинарный режим, 287
Битовое поле, 51, 277, 282
Блок, 22, 40, 60, 68, 95, 155, 164, 227, 284, 352
Буфер ввода-вывода
  клавиатуры, 75
  операционной системы, 284, 290
Буферизация, 322

В

Ввод, 43, 56, 73, 87, 89, 165, 200, 288, 292, 322, 328
Вещественная константа,
  см. Константа вещественная
Вложение блоков,
  см. Блок
  инициализаторов, 290
  комментариев,
    см. Комментарий
  операторов цикла, 77, 84, 91
  переключателей,
    см. Оператор **switch**
  составных операторов,
    см. Составной оператор
  структур, 253
  условных операторов,
    см. Условный оператор

Возвращаемое значение,
см. Значение, возвращаемой функцией
Восьмеричная константа,
см. Константа восьмеричная
Вывод, 43, 56, 60, 61, 123, 124, 165, 210, 365
Выражение, 32, 35, 42, 44, 46, 47, 67, 78, 96, 108, 145, 159, 183, 240, 323, 327
арифметическое, 54
леводопустимое,
см. Леводопустимое выражение
логическое, 46, 54, 69, 78
модифицируемое именующее, 35, 47,
см. *l*-значение
первичное, 98
постфиксной формы, 45
префиксной формы, 45
присваивания,
см. Операция присваивания
пустое, 42
условное, 54
целочисленное константное, 125
Вычитание,
см. Операция вычитания

Г

Глобальный объект, 56, 59, 124, 229, 230, 231
переменная, 227, 229, 230, 329, 332

Д

Двоичный режим обмена с файлами,
см. Бинарный режим
Декремент,
см. Операция «декремент»
Деление,
см. Операция деления
Десятичная константа,
см. Константа десятичная
Динамическая память, 230, 231, 232
выделение,
см. Память, динамическое выделение
управление, 159
Динамическое представление данных, 267
Динамические массивы, 154, 155, 157, 159, 161
см. Массив динамический
объекты, 232, 266
односвязные списки, 269
структуры, 223
информационные, 267, 268, 269, 271
Директива препроцессора
#, 11
#define, 26, 28, 105, 115, 117, 123, 129, 243
#elif, 115, 125, 129
#else, 115, 125, 128
#endif, 115, 125, 126
#error, 115, 136
#if, 115, 125, 128, 135
#ifdef, 115, 125, 129
#ifndef, 115, 125, 129
#include, 29, 44, 56, 61, 67, 115, 122, 135, 138
#line, 115, 135
#pragma, 116, 137, 247
#undef, 115, 121, 129

Доступ к адресам
параметров, 212
значению переменной, 21
кодам библиотечных
функций, 58
объекту, 182
отдельным битам, 277
участку памяти, 22
файлу, 331
элементам массива, 220
элементам структур, 249

З

Заголовок переключателя,
см. Оператор **switch**
функции, 96, 176,186
цикла,
см. Цикл
Заголовочный файл, 29, 59,
329
alloc.h, 155, 162, 364
assert.h, 123
conio.h, 365
ctype.h, 123, 360
dos.h, 366
errno.h, 123, 288, 327
float.h, 29, 30, 123, 358
limits.h, 29, 117, 118, 123,
357
locate.h, 124
math.h, 70, 106, 124, 185,
204, 359
mem.h, 367
setjump.h, 124
signal.h, 124, 368
stdarg.h, 43, 124, 216, 218
stddef.h, 124, 147
stdlib.h, 102, 104, 124, 155,
156, 173, 189, 273
stdio.h, 44, 61, 64, 73, 75,
89, 123, 124, 143, 284,
290, 362
string.h, 124, 189, 195, 210,
220, 362
time.h, 124
Зарезервированное слово,
см. Служебное слово
Знаки операций, 12, 30, 44, 114
Значение, возвращаемое
функцией
леводопустимое,
см. *l*-значение
указателя, 145, 146, 148, 152,
154, 188, 204, 218, 260, 272,
321, 331
нулевое,
см. Нулевой указатель

И

Идентификатор, 11, 12, 70,
96, 197
библиотеки, 343
препроцессорный, 117, 118,
127, 243
Имя
директивы, 115
заголовочного файла, 29, 58
исполняемой программы, 337
компонента, 37
константы, 27 129
макроса, 130, 341
массива, 88, 151, 153, 157,
188, 201, 248
объединения, 275
объекта, 74
глобального, 233
структурированного, 37
параметра, 176

переменной, 32, 47, 76, 141, 142, 145
препроцессорного идентификатора, 116
структуры, 247
типа, 38, 247
Индексация,
см. Операция **[ ]**
Инициализатор, 159, 253
Инициализация, 24, 94
массива, 41, 170
переменной, 44
структуры, 43
Инкремент,
см. Операция «инкремент»
Исполняемый оператор,
см. Оператор

К

Класс памяти
auto, 13, 227, 228
extern, 13, 233
register, 13, 227
static, 13, 228
автоматической, 13
Ключевое слово,
см. Служебное слово
Кодировка ASCII,
см. ASCII-код
Команда препроцессора,
см. Директива препроцессора
Комментарий /* */, 11
Компоновка, 57
Константа
арифметическая, 17
вещественная, 16
восьмеричная, 16
десятичная, 16
именованная, 19
литерная,
см. Константа символьная
неарифметическая, 19
см. Нулевой указатель
перечисляемого типа, 13
предельная,
см. Предельные значения констант
предопределенная,
см. Предопределенные константы
препроцессорная, 26
с плавающей точкой,
см. Константа вещественная
символьная, 14, 121
строковая, 21, 294, 301
в нескольких строках, 20
указатель,
см. Указатель-константа
целая, 16, 18, 19, 74, 98, 144, 207
шестнадцатеричная, 16

Л

Леводопустимое выражение, 32
см. *l*-значение
Лексема, 10, 14, 30, 39, 135
препроцессора, 114, 116, 130
строки замещения, 135
Лексический элемент,
см. Лексема
Литерал,
см. Константа
Литерная константа,
см. Константа символьная
Логическая операция,
см. Операция логическое И (ИЛИ, НЕ)

М

Макроопределение, 130, 134
  см. Директива препроцессора
  #define
  va_arg( ), 216, 218, 219
  va_end( ), 216, 218
  va_start( ), 216, 217, 218
Макрос,
  см. Макроопределение
Массив, 37, 39, 87, 120
  динамический, 154, 155
  доступ к элементам,
    см. Доступ к элементам массива
  и указатель, 151
  имя,
    см. Имя массива
  инициализация,
    см. Инициализация массива
  многомерный, 39, 94, 131
  определение,
    см. Определение массива
  параметр, 188, 190
  символьный, 275, 365
  структур, 254, 255, 256, 261, 281
  указателей, 159, 161, 164, 200, 201, 205
    на строки, 235
    на функции, 200, 201, 205
Метка, 43, 70
  case в переключателе, 108
  default в переключателе, 108
Минус,
  см. Операция «минус унарный»
Многомерный массив,
  см. Массив многомерный
Модификатор 63, 73, 303
  см. Служебное слово
  cdecl, 215
  const, 12, 25, 294, 367
  pascal, 215, 216
  volatile, 12, 13
  спецификации
    преобразования, 61, 62, 73, 298

Н

Неоднозначность, 45
Нулевой указатель (**NULL**), 14, 19, 220, 287

О

Обмен с файлами, бинарный,
  см. Бинарный режим
  двоичный,
    см. Бинарный режим
  строковый, 312
  форматный, 314
Обобщенный пробельный символ, 11, 21
Объединение, 13, 274, 275, 276, 279
Объединяющий тип, 13, 275, 276
Объект, 13, 21, 25, 31, 33, 42
Оператор,
  см. Служебное слово
  break, 12, 13, 68, 84, 85, 89
  continue, 12, 13, 68, 85, 86, 87, 90, 171
  do, 12, 13, 14, 78, 79, 80, 83
  else, 14, 69, 70
  for, 12, 14, 42, 68, 78, 79, 80, 81
  goto, 12, 14, 68, 71, 109

if, 12, 14, 69, 70
return, 12, 14, 68, 96, 97
switch, 12, 13, 14, 68, 108, 110, 111
while, 12, 14, 68, 78, 79, 80, 85
безусловного перехода, *см.* Оператор **goto**
возврата из функции, *см.* Оператор **return**
выбора, *см.* Метка **case** в переключателе
выражение, 32, 35, 38, 44
переключатель, *см.* Оператор **switch**
присваивания, *см.* Операция присваивания
пустой, 42, 70, 71, 82
составной, *см.* Составной оператор
условный, *см.* Оператор **if**
цикла, *см.* Цикл
Операционная система
MS-DOS, *см.* MS-DOS
MS Windows, *см.* Windows
UNIX, *см.* UNIX
Операция, 14, 31
#, 44 57
##, 134
defined, 128
( ), 11, 31, 37, 39
[ ], 11, 30, 31, 37
{ }, 11, 39, 40, 96
sizeof, 157, 175, 268
аддитивная, 31, 33, 45, 66, 145, 146
бинарная, 30 259
больше или равно (**>=**), 31, 34, 46, 148
больше, чем (**>**), 11, 31, 34, 46
получения адреса (**&**), 30, 31
вычисления остатка (**%**), 11, 31, 33, 36, 44
вычитания (**-**), 146
декремент (**--**), 32, 45, 152
деления (**/**), 49
доступа к компоненту по имени структурированного объекта, 31, 37, 249
запятая (**,**), 31, 33, 41
индексации, *см.* Операция **[ ]**
инкремент (**++**), 32, 45, 152
логическое И (**&&**), 46
ИЛИ (**||**), 46
НЕ (**!**), 46
меньше или равно (**<=**), 34, 46
меньше, чем (**<**), 34, 46
минус унарный (**-**), 31, 48
мультипликативная, 31, 33
над указателями, 144, 197, 260
не равно (**!=**), 34, 46
отношения, 46, 54, 69, 146
плюс унарный (**+**), 11, 31, 44
поразрядное И (**&**), 54
ИЛИ (**|**), 11, 31, 34, 53
ИСКЛЮЧАЮЩЕЕ (**^**), 31, 34, 48
НЕ (**~**), 31, 32, 52
с присваиванием (**&=**), 31, 36

с присваиванием (|=), 31, 36
постфиксная, 32, 45
префиксная, 32, 45
приведения, *см.* Операция преобразования
приоритет, *см.* Приоритет операций
присваивания (=), 42, 145, 167, 195
разыменования (*), 31, 199, 256
сдвига влево (<<), 34
сдвига вправо (>>), 34
сложения (+), 33, 34
сравнения на равенство (==), 31, 34, 35, 148
умножения (*), 33, 44
условная (?:), 30, 31
явного преобразования типа, 38

Описание
внешнего объекта, 234
и определение, *см.* Определение и описание
символьной переменой, 165
структуры, 277
указателя, 144
функции, 40, 98, 178

Определение, 22, 95, 285
и описание, 40, 60, 178
массива, 187
объединения, 275
переменной, 22
указателя, 143
на структуру, 257
на функцию, 196
функции, 95, 98
с переменным числом аргументов, 211

Остаток, *см.* Операция взятия остатка

Отношения, *см.* Операция отношения

П

Память, выделение
автоматическое, *см.* Класс памяти автоматической
динамическое, 267
локальная, *см.* Класс памяти **auto**
регистровая, *см.* Класс памяти **register**
ЭВМ, 22

Параметр макроса, 115, 130, 134, 135

Параметр функции, 40, 43, 60, 68, 95, 96, 98, 176, 180, 186, 192, 207

Переключатель, *см.* Оператор **switch**

Переменная, 21
автоматической памяти, *см.* Класс памяти **auto**
вещественная, 62
глобальная, *см.* Глобальная переменная
индексированная, *см.* Индексация
как объект, 21
локальная, *см.* Класс памяти **static**
регистровая, *см.* Класс памяти **register**
статическая, *см.* Класс памяти **static**

целочисленная, 22
Перечислимая константа,
см. Константа перечисляемого типа
Перечисляемый тип, 13
Плюс,
см. Операция плюс унарный
Побочные эффекты, 124
Поле битовое,
см. Битовое поле
Поразрядные операции,
см. Операция поразрядное И (ИЛИ, НЕ)
Предельные значения
для вещественных типов, 358
констант, 17, 22, 24, 28, 30, 73, 118
переменных целочисленных типов, 357
Преобразование,
см. Операция преобразования
Предопределенные
дескрипторы файлов, 329
значения указателя, 289
константы, 316, 324, 329
указатели на поток, 285, 286, 292
Препроцессор, 57, 114
директивы,
см. Директивы препроцессора
команды,
см. Директивы препроцессора
Префиксная операция,
см. Операция префиксная
Приведение,
см. Операция преобразования
Приоритет операций, 30, 31, 35, 42, 147, 152, 199, 259
Присваивание,
см. Операция присваивания
множественное, 47
Пробельный символ, 301, 302
обобщенный 11, 21
Производные типы, 141, 239, 264
Прототип, 39, 40, 61, 98, 99, 123, 366,
см. Описание функции

Р

Разделитель, 41, 56, 340
пробельный,
см. Обобщенный пробельный символ
Разыменование указателей (обращение по указателю),
см. Операция разыменования
Ранги операций,
см. Приоритет операций
Рекурсивная функция, 224, 225
Рекурсия, 223, 226, 271

С

Сдвиг влево,
см. Операция сдвига влево
с присваиванием, 31, 36
вправо,
см. Операция сдвига вправо
с присваиванием, 31, 36
Символ '**\0**', 20, 169, 173, 189, 293
'**\n**'? 15, 20, 114,
см. Эскейп-последовательность
подчеркивания, 11, 12, 14

Слово зарезервированное,
см. Служебное слово
ключевое,
см. Служебное слово
Сложение,
см. Операция сложения
Служебное слово, 12, 19, 60, 66, 275
auto,
см. Класс памяти **auto**
break,
см. Оператор **break**
case,
см. Метка **case** в переключателе
cdecl,
см. Модификатор **cdecl**
char,
см. Тип **char**
const,
см. Модификатор **const**
continue,
см. Оператор **continue**
default,
см. Метка **default** в переключателе
delete,
см. Операция **delete**
do,
см. Оператор **do**
double,
см. Тип **double**
else,
см. Оператор **else**
enum,
см. Перечислимые константы
extern,
см. Класс памяти **extern**
far,
см. Модификатор **far**
float,
см. Тип **float**
for,
см. Оператор **for**
goto,
см. Оператор **goto**
if,
см. Оператор **if**
int,
см. Тип **int**
long,
см. Тип **long**
pascal,
см. Модификатор **pascal**
register,
см. Класс памяти **register**
return,
см. Оператор **return**
short,
см. Тип **short**
signed, 12, 13, 22, 23, 51
sizeof,
см. Операция **sizeof**
static,
см. Класс памяти **static**
struct,
см. Структурный тип
switch,
см. Оператор **switch**
typedef,
см. Спецификатор **typedef**
union,
см. Объединяющий тип
unsigned,
см. Тип **unsigned**
void,
см. Тип **void**

volatile,
см. Модификатор **volatile**
while,
см. Оператор **while**
Составной оператор, 68, 69, 95
Спецификатор, 12
typedef, 12, 13, 206, 242
Спецификация преобразования, 149, 165, 294, 295, 302, 303
Список инициализации,
см. Инициализация
Сравнение,
см. Операция сравнения на равенство
Стандартный поток, 288, 289, 341, 345, 362
Строка замещения, 117, 119, 120, 129, 130, 243
форматная, 62, 73, 177, 221, 294, 314
Строковая константа,
см. Константа строковая
Структура, 12, 239, 252, 262,
см. Структурный тип
Структурный тип, 13, 240, 241, 242, 243, 244, 245

Т

Тег,
см. Структурный тип
Тело функции, 60, 96, 176, 177, 197, 229
цикла,
см. Цикл
Тип, 12
char, 165
double, 18, 64
float, 65, 73
int, 17, 26, 37, 62, 96
long, 16, 370
long double, 50
short, 13, 22, 23, 24, 51, 303, 304, 357
unsigned, 295
void, 12, 13, 60, 96, 144, 180, 271
void*, 155
агрегирующий,
см. Агрегирующий тип
арифметический, 17, 24, 29
базовый, 165
беззнаковый,
см. Тип **unsigned**
возвращаемого значения,
см. Значение, возвращаемое функцией
данных, 18, 24, 216, 240, 264, 275, 317
знаковый,
см. Тип **signed**
объединяющий,
см. Объединяющий тип
переменной, 17, 60
перечисляемый,
см. Перечисляемый тип
производный,
см. Производные типы
результата,
см. Тип возвращаемого значения
скалярный,
см. Скалярный тип
структурный,
см. Структурный тип
указателя, 198, 199
void*,
см. Тип **void***
функции, 157, 198

чисел с плавающей точкой,
см. Тип **float**, Тип **double**
Точность простая,
см. Тип
float, 65, 73, 76
двойная,
см. Тип **double**

У

Указатель, 14, 19, 31, 37, 43, 141, 143, 145, 147, 151
значение, 144, 197,
см. Значение указателя
и массив, 164, 180,
см. Массив и указатель
инициализация, 94, 234,
см. Инициализация массива
константа, 151, 153, 197
константный,
см. Указатель-константа
на массив, 150
на объект, 43, 141, 263, 269
на поток, 287, 288, 308, 315, 323
на строку, 173, 211
на структурированный объект, 37
на структуру, 258, 278
на функцию, 197, 198, 202, 204, 207, 208
нулевой,
см. Нулевой указатель
объект, 150
параметр, 181
переменная, 145
пустой,
см. Нулевой указатель
родовой,
см. Тип **void***
Умножение,
см. Операция умножения
Унарные операции, 66, 148
Управляющая последовательность, 15
Условная операция,
см. Операция условная трехместная, 38, 54
Условный оператор,
см. Оператор **if**
Уточненное имя,
см. Имя уточненное

Ф

Файл заголовочный,
см. Заголовочный файл
Форматирование данных при вводе-выводе, 294
Форматная строка, 61, 73, 177, 221, 294, 300, 301, 314
Флаг форматирования, 297
Функция, 176
main( ), 60, 100, 103
описание,
см. Описание функции
определение,
см. Определение функции
прототип,
см. Прототип
рекурсивная,
см. Рекурсивная функция
с переменным количеством аргументов, 211
самовызывающая,
см. Рекурсивная функция
указатель,
см. Указатель на функцию
Функция библиотечная
calloc( ), 155, 164, 219
close( ), 32, 328

creat( ), 322, 323, 327
exit( ), 104, 108, 226
fclose( ), 288
fopen( ), 287
free( ), 155, 156, 164, 230, 266
fseek( ), 316, 317, 318, 321
getchar( ), 166, 167, 288, 289
lseek( ), 323, 331, 332
malloc( ), 155, 157, 158, 164, 173, 271
open( ), 322, 323, 325, 327
perror( ), 287, 288
printf( ), 61, 62, 67, 68, 165, 168, 288, 293, 294
putchar( ), 288, 289, 291, 292, 309
qsort( ), 207, 208, 210
read( ), 322, 328, 329
scanf( ), 73, 74, 76, 165, 169, 183
strcmp( ), 210, 211
strcpy( ), 194
strlen( ), 195
strncat( ), 192
strstr( ), 191, 364
va_arg( ), *см.* Макроопределение **va_arg( )**
va_end( ), *см.* Макроопределение **va_end( )**
va_start( ), *см.* Макроопределение **va_start( )**
write( ), 322, 328, 329, 330

Ц

Целая константа, *см.* Константа целая
Целочисленный тип, *см.* Тип **int (long**, **short**, **unsigned)**
Цикл, 14
  бесконечный, 205, 307
  итерационный, *см.* Оператор **for**
  с постусловием, *см.* Оператор **do**
  с предусловием, *см.* Оператор **while**

Ш

Шестнадцатеричная константа, *см.* Константа шестнадцатеричная

Э

Эскейп-последовательность, 119
Эффекты побочные, *см.* Побочные эффекты

Я

Явное преобразование типа, *см.* Операция явного преобразования типа