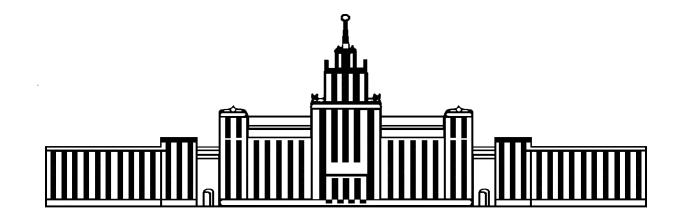
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ



ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

681.3(07) K289

С.Т. Касюк

КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Конспект лекций

Челябинск 2010

Министерство образования и науки Российской Федерации Федеральное агентство по образованию Южно-Уральский государственный университет Кафедра информатики

681.3(07) K289

С.Т. Касюк

КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Конспект лекций

Челябинск Издательский центр ЮУрГУ 2010

УДК 681.3(075.8) ББК Ч23.я7 К289

Одобрено

учебно-методической комиссией факультета экономики и управления

Рецензенты: д.т.н. В.С. Жабреев, к.т.н. В.Л. Федяев

Касюк, С.Т.

К289 Курс программирования на языке Си: конспект лекций/ С.Т. Касюк. — Челябинск: Издательский центр ЮУрГУ, 2010. — 175 с.

Учебное пособие по курсу программирования на языке Си написано в соответствии утвержденной с программой для обучения студентов на кафедре информатики Южно-Уральского государственного университета по специальности «Прикладная информатика в экономике».

Цель пособия состоит в поэтапном формировании у учащихся знаний о методах составления программ и выработке навыков программирования на языке Си.

Весь материал разбит по главам, в которых приведены: 1) базовые конструкции и элементы стандарта ANSI языка программирования Си, 2) описание динамических информационных структур и 3) вопросы сортировки и поиска данных. Имеется много примеров программ на языке Си учебного и практического характеров.

ББК Ч23.я7

ПРЕДИСЛОВИЕ

В учебном пособии представлен конспект лекция ПО курсу программирования на языке Си, который используется для подготовки студентов «Прикладная информатика В экономике» на кафедре информатики Южно-Уральского государственного университета.

Цель настоящего учебного пособия — помочь овладеть учащимся методами составления программ и выработать навыки программирования на языке Си. Для достижения этой цели студенты должны познакомиться с базовыми конструкциями и элементами стандарта ANSI языка Си, изучить динамические информационные структуры и рассмотреть наиболее важные алгоритмы, применяемые в настоящее время для сортировки и поиска данных.

При изложении материала автор стремился учитывать, что пособие ориентировано на студентов младших курсов, только начинающих изучать программирование. По своему объему данных пособие приближается к **учебникам** по языку программирования Си для студентов, информатику и вычислительную технику. Однако оно отличается от них в основном менее подробным изложением материала, поскольку, по мнению автора, изучение разнообразных языковых конструкций и возможностей при начальном освоении программирования затеняет базовые понятия языка Си и «сбивает студентов толку». Поскольку учащихся c ДЛЯ практика программирования имеет огромное значение, в пособии приведено много учебного программ прикладного характеров, снабженных И комментариями.

Учитывая, что студенты часто неверно трактуют работу с указателями, автор стремился как можно чаще приводить иллюстрации содержимого оперативной памяти для предотвращения возникающих на этой почве типичных ошибок.

В пособии даны программные реализации стеков, списков, очередей и деревьев. Кроме того, приведены программы с использованием динамических информационных структур из книги Кернигана Б. и Ритчи Д. «Язык программирования Си». Раздел сортировки и поиска разработан несколько подробнее других разделов, так приведены прикладные алгоритмы обработки данных, примеры сортировки и поиска данных, расчетные характеристики алгоритмов.

При работе над учебным пособием автор стремился использовать только качественные источники и материалы, приведенные в списке литературы.

Глава 1. Язык программирования Си

§1.1. Введение в язык Си

Язык программирования Си — универсальный язык программирования, который завоевал особую популярность у программистов, благодаря сочетанию возможностей языков программирования высокого и низкого уровней.

Язык Си широко применяется при современном профессиональном программировании*. Большинство программистов предпочитают использовать язык Си для своих серьезных разработок потому, что их привлекают такие особенности языка, как свобода выражения мыслей, мобильность и чрезвычайная доступность.

Язык Си наряду с тем, что он позволяет освоить хороший стиль программирования, так же как более простые и менее мощные языки высокого уровня (Бейсик, Паскаль), даёт возможность программисту осуществлять непосредственный доступ к ячейкам памяти и регистрам компьютера, требуя при этом знания особенностей функционирования ЭВМ. В этом Си схож с языком низкого уровня — ассемблером. Поэтому язык Си иногда называют ассемблером высокого уровня, хотя на самом деле он представляет собой гораздо более мощное средство решения трудных задач и создания сложных программных систем.

Позиции языка Си++ в современном мире. Современные языки программирования:

- Си++ язык системного программирования;
- Java язык программирования для Internet и мобильных систем;
- Visual Basic язык разработки Windows-приложений;
- Delphi объектно-ориентированный язык Object Pascal.

Практически все используемые в мире ключевые программные средства, в операционные системы, СУБД. числе компиляторы, TOM телекоммуникаций написаны на Си++. Несколько примеров: а) практически все программные продукты Microsoft (Windows XP, Office XP, Internet Exploer, MS SQL Server и др.), б) ведущие продукты Adobe Systems (Photoshop, Acrobat и др.), в) базовые компиляторы Sun, г) графическая оболочка KDE для Linux, д) многие компоненты Mac OS X и т.д. Не вызывает сомнений подавляющее превосходство Си++ в области встроенных систем и индустрии компьютерных игр (Doom III, StarCraft и др.). На Си++ реализованы ведущие поисковые Webсистемы и крупнейшие Web-порталы: Google, Yahoo, Amazon и др. Создатель Си++ Бьерн Страустроп приводит следующие аргументы в пользу языка: «Си++ является наилучшим языком для многих приложений, где требуется системное программирование, имеются определенные ограничения по выдвигаются серьезные требования к производительности. Одним из примеров служит Google, другим — встроенные системы для миниатюрных устройств».

Язык Си был разработан американцем Деннисом Ритчи в исследовательском центре Computer Science Research Center of Bell Laboratories корпорации AT&T в 1972 г. Первоначальная реализация Си была выполнена на ЭВМ PDP-11 фирмы DEC для создания операционной системы UNIX. Позже он был перенесен в среду многих операционных систем, обособился и существует независимо от любой из них. Программы, написанные на языке Си, как правило, можно перенести в любую другую операционную систему или на другой компьютер либо с минимальными изменениями, либо вовсе без них.

Диалекты языка Си. Первое описание языка Си дал его автор Деннис Ритчи совместно с Брайном Керниганом в книге «Язык программирования Си». Однако, описание не было строгим и содержало ряд неоднозначных моментов. Разработчики трактовали язык по-разному. Фактически, долгое время стандартом языка служила его реализация в UNIX. Сейчас существуют десятки реализаций языка программирования Си. Они поддерживают разные диалекты языка.

В 1983 г. при Американском Институте Национальных Стандартов (American National Standart Institute — ANSI) был создан комитет по стандартизации языка Си. В 1989 г. был утверждён окончательный вариант стандарта. Однако на сегодняшний день большинство реализаций языка Си не поддерживают стандарт в полном объёме.

§1.2. Структура программы

Программа на языке Си состоит из одной или более подпрограмм, называемых функциями. Каждая функция в языке Си имеет свое имя. В любой программе одна из функций обязательно имеет имя *main*.

Имя функции — это коллективное имя группы объявлений и операторов, заключенных в фигурные скобки. За именем функции в круглых скобках указываются параметры функции.

Пример функции

```
/* Первая программа на Си. */
#include <stdio.h>
main()
{
 printf("\n Здравствуй, язык Си!");
 /* Вывод на экран сообщения.*/
}
```

Результат работы программы

```
Здравствуй, язык Си!
```

В этой программе имя функции *main*. При выполнении программы, созданной на языке Си, операционная система компьютера всегда передаёт управление в программу на функцию с именем *main*. Обычно, хотя это не обязательно, функция *main* стоит первой в тексте программы. Следующие за именем функции круглые скобки играют важную роль. В них указываются параметры (аргументы), которые передаются в функцию при обращении к ней. В данном случае операционная система не передаёт в функцию *main* никаких параметров, поэтому список аргументов в круглых скобках пустой. В фигурные скобки « { } » заключены описания и операторы, которые обеспечивают вывод на экран компьютера сообщения «Здравствуй, язык Си!».

В общем случае программа содержит несколько функций.

Пример программы из нескольких функций

```
#директивы препроцессора

main()
{
    ...
}

function_1(...)
{
    ...
}

function_2(...)
{
    ...
}

...
}
```

Функция *main* может вызывать для выполнения любую другую функцию. Функции *function_1*, *function_2*, ..., *function_n* могут вызвать любую функцию, кроме функции *main*. Функцию *main* нельзя вызывать изнутри программы, она является управляющей.

§1.3. Объекты языка Си и их типы

Программа, написанная на языке Си, оперирует с объектами. Они могут быть простыми и сложными. К простым объектам относятся переменные и константы, к сложным — массивы, структуры, очереди, списки и т.д. Каждый объект имеет имя и тип. Обращение к объекту программы осуществляется по его имени (идентификатору).

Имя объекта — это последовательность не более 32 символов а—z, A—Z, 0—9 и «_» (подчеркивания). Начальный символ имени не должен быть цифрой. Несмотря на то, что допускается имя, имеющее до 32 символов, определяющее значение имеют только первые 8 символов.

Помимо имени, каждый объект имеет тип. Указание типа необходимо для того, чтобы было известно, сколько места в оперативной памяти будет занимать данный объект.

Основные типы и размеры данных:

- 1) *char* символьный, 1 байт;
- 2) *int* целый, 2 байта;
- 3) *short* короткий целый, 2 байта;
- 4) *long* длинный целый, 4 байта;
- 5) float числа с плавающей точкой, 4 байта;
- 6) double числа с плавающей точкой двойной точности, 8 байт.

Тип char используется для описания символьных объектов. Типы *short*, *long*, *int* предназначены для описания объектов, значения которых выражаются целыми числами. Типы *float* и *double* предназначены для объектов, значения которых выражаются действительными (вещественными) числами.

В программе должно быть дано объявление каждого объекта с указанием его имени и типа. Описание объекта должно предшествовать его использованию в программе.

Пример объявления объектов

```
int n; /* Переменная n целого типа. */
float x1; /* Переменная x1 типа с плавающей точкой. */
char a; /* Переменная а символьного типа. */
```

§1.4. Простые объекты

К простым объектам языка Си относятся константы и переменные.

Константа — это ограниченная последовательность символов алфавита языка (лексема), представляющая собой изображение фиксированного (неизменяемого) объекта.

Константы бывают следующие: 1) *числовые*, 2) *символьные* и 3) *строковые*. Числовые константы делятся на целые и вещественые.

Целые константы

Виды целых констант показаны в табл. 2.1.

Таблица 1.1

Виды целых констант

Десятичные	Последовательность цифр $(0-9)$, которая начинаются с цифры отличной от нуля. Пример: 1, -29 , 385. Исключение здесь — число ноль 0
Восьмеричные	Последовательность цифр $(0-7)$, которая всегда начинается с нуля. Пример: 00 , 071 , -052 , -03
Шестнадцатиричные	Последовательность шестнадцатеричных цифр $(0-9 \text{ и A}-F)$, которой предшествует присутствует 0х. Пример: 0х0, 0х1, -0 X2AF, 0X17

В зависимости от значения целой константы компилятор присваивает ей тот или иной тип (*int*, long, unsigned int).

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

Пример

50000U — константа типа unsigned int

Константе 50000U выделяются 2 байта вместо четырех, как было бы при отсутствии суффикса. В этом случае, т.е. 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.

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, причем знаковый разряд будет использоваться для представления разряда кода (а не знака).

Пример

OLU — целая константа типа *unsigned long* длиной 4 байта 24242424UL — константа типа *unsigned long*

Вещественные константы

Константа с плавающей точкой (вещественная константа) всегда представляется числом с плавающей точкой двойной точности, т. е. как имеющая тип *double*, и состоит из следующих частей [2]:

- целой части последовательности цифр;
- десятичной точки;
- дробной части последовательности цифр;
- \bullet символа экспоненты e или E;
- экспоненты в виде целой константы (может быть со знаком).
- Любая часть (но не обе сразу) из нижеследующих пар может быть опущена:
 - целая или дробная часть;
- ullet десятичная точка или символ e (E) и экспонента в виде целой константы.

Примеры

```
345.
3.14159
2.1E5
.123E3
4037e-5
```

По умолчанию компилятор присваевает вещественному числу тип *double*.

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

Примеры:

- 3.14159F константа типа *float*, занимающая 4 байта;
- 3.14L константа типа *loung double*, занимающая 10 байт.

Символьные константы

Символьная константа — это один символ или обратная косая черта и символ, заключенные в апострофы (одинарные кавычки), например: 'z', ' \n', ' \t' и так далее. Обратная косая черта (слэш) и символ служат для обозначения управляющих символов, не имеющих графического представления, например, '\n' — переход на новую строку, '\t' — табуляция. Все символьные константы имеют тип char и занимают в памяти по 1 байту. Значением символьной константы является числовое значение её внутреннего кода.

Строковые константы

Строковая константа— это последовательность символов, заключенная в кавычки, например: "Это строковая константа". Кавычки не входят в строку, а лишь ограничивают её. Технически, строковая константа представляет собой массив символов и по этому признаку может быть отнесена к разряду сложных объектов языка Си. Однако, строковую константу удобнее рассмотреть вместе с другими константами.

В конце каждой строковой константы компилятор помещает символ '\0', чтобы программе было возможно определить конец строки. Такое представление означает, что размер строковой константы не ограничен каким-либо пределом, но для определения длины строковой константы её нужно полностью просмотреть.

Поскольку строковая константа состоит из символов, то она имеет тип char. Количество ячеек памяти, необходимое для хранения строковой константы на единицу больше количества символов в ней. Следует отчетливо понимать, что символьная константа и строка из одного символа не одно и то же: 'x' не есть "x". Первое — это символ, использованный для числового представления буквы x, а второе — строковая константа, содержащая символ x и '\0'. Если в программе строковые константы записаны одна за другой через разделители, то при выполнении программы они будут «склеены».

Переменные

Переменная — лексема, представляющая собой изображение изменяемого объекта.

С технической точки зрения, переменная — это область памяти, в которую могут помещаться различные числа (двоичные коды). Любая переменная до её использования в программе должна быть описана, т. е. для нее должены быть указаны тип и имя (идентификатор).

Пример

Предпочтительно использовать именно такой способ описания, чтобы при необходимости можно было модифицировать имя переменной. Кроме того, в этом случае каждую переменную удобно снабдить комментарием, поясняющим ее смысл.

Пример

Общий случай объявления переменных

```
тип_переменных имя_переменной_1, имя_переменной_2, . . . имя_переменной_n;
```

При объявлении переменных им можно задавать начальные значения — производить инициализацию.

Пример

```
тип_переменной имя_переменной = значение;
```

Примеры

```
int i=0, k, n, m=1;
float x=314.159E-2, y;
char a='a';
```

§1.5. Операции

Над объектами в языке Си могут выполняться различные операции [2]:

- 1) арифметические;
- 2) логические;
- 3) адресные;
- 4) операции отношения;
- 5) операции присваивания.

Результат выполнения операции — всегда число.

Операции могут быть *двухместными* (бинарными) или *одноместными* (унарными). Двухместные операции выполняются над двумя объектами, одноместные — над одним.

Арифметические операции

Основные двухместные операции, расположенные в порядке уменьшения приоритета:

- 1) умножение «*»;
- 2) деление «/»;
- 3) сложение «+»;
- 4) вычитание и арифметическое отрицание «-»;
- 5) целочисленное деление (вычисление остатка от деления) «%».

Самый высокий приоритет у операции «умножение», самый низкий у операции «целочисленное деление».

Основные одноместные операции:

- 1) приращение на единицу «++»;
- 2) уменьшение на единицу «--».

Результат вычисления выражения, содержащего операции «++» или «--», зависит от того, где расположен знак операции (до объекта или после него). Если операция расположена до переменной, то сначала происходит изменение значения переменной на 1, а потом выполняется какая-то операция; если — после переменной, то сначала выполняется операция, а потом значение переменной изменяется на 1.

Примеры:

- $a^{*}++b$ если a=2 и b=3, то результат вычислений равен 8, а b=4;
- a*b++ если a=2 и b=3, то результат вычислений равен 6, а b=4.

Логические операции

Логических операций в языке Си три:

- 1) «&&» логическое «И» (коньюнкция);
- 2) «||» логическое «ИЛИ» (дизъюнкция);
- 3) «!» логическое «НЕ» (отрицание).

Логические операции могут выполняться над любыми объектами. Результат логической операции: единица, если выражение истинно; ноль, если выражение ложно. Вообще, все значения, отличные от нуля, интерпретируются как истинные. Логические операции имеют низкий приоритет, и поэтому в выражениях с такими операциями скобки используются редко.

Адресные операции

Адресные операции:

- 1) определение адреса «&»;
- 2) обращение по адресу «*».

Адресные операции являются унарными.

Операции отношения

Операции отношения:

- 1) равно « == »;
- 2) не равно « != »;
- 3) меньше « < »;
- 4) больше « > »;
- 5) меньше или равно « <= »;
- 6) больше или равно « >= ».

Операции используются при организации условий и ветвлений. Все эти операции вырабатывают результат типа *int*. Если отношение между операндами истинно, то значение этого условия — единица, если ложно — ноль.

Операция присваивания

Операция присваивания выполняется следующим образом:

- 1) вычисляется выражение в правой части;
- 2) тип результата преобразуется к типу объекта в левой части;
- 3) результат записывается по адресу, где находится объект.

Пример

```
объект = <выражение>;
```

§1.6. Ввод и вывод информации

Основной задачей программирования является обработка информации, поэтому любой язык программирования должен иметь средства для ввода и вывода данных. В языке Си нет операторов ввода-вывода; ввод и вывод информации осуществляется через функции стандартной библиотеки. Прототипы данных функций находятся в файле stdio.h. Чаще всего вывод осуществляется через функцию *printf*, а ввод — *scanf*.

Функция printf — функция форматированного вывода. 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, т. е. влиять на их представление на экране дисплея. Возможность форматирования условно отмечена в самом имени функции с помощью литеры f в конце её названия (print formatted).

Общая форма записи функции printf()

```
printf("строка_форматов",объект_1,объект_2,...,объект_n);
```

Строка форматов состоит из следующих элементов:

- 1) управляющих символов;
- 2) текста, который выводится на экран;
- 3) форматов, предназначенных для вывода значений переменных различных типов.

Объекты могут отсутствовать. Управляющие символы не выводятся на экран, а управляют расположением выводимых символов. Отличительной чертой управляющего символа является наличие слэша перед ним.

Основные управляющие символы:

- 1) '\n' новая строка;
- 2) '\t' горизонтальная табуляция;
- 3) '\v' вертикальная табуляция;
- 4) '\b' возврат на символ;
- 5) '\r' возврат на начало строки;

6) '\a' — звуковой сигнал.

Форматы нужны для того, чтобы указывать вид, в котором информация будет выведена на экран. Отличительной чертой формата является наличие символа процент «%» перед ним.

Основные форматы:

- 1) %d целый формат со знаком;
- 2) %и целый формат без знака;
- 3) %f вещественный формат (числа с плавающей точкой типа float);
- 4) %lf— вещественный формат (числа с плавающей точкой типа double);
- 5) %e вещественный формат в экспоненциальной форме (числа с плавающей точкой типа *float* в экспоненциальной форме);
 - 6) %c символьный формат;
 - 7) % строковый формат;
 - 8) %*p* адресный формат.

Пример

```
printf("\n Здравствуй, язык Си!");
```

Результат работы программы

```
Здравствуй, язык Си!
```

Пример

```
a=5;
printf("\n Значение переменной a=%d.",a);
```

Результат работы программы

```
Значение переменной а=5.
```

Пример

```
x=2.78;
printf("\n Значение переменной x=%f",x);
```

Результат работы программы

```
Значение переменной х=2.780000
```

При указании формата можно явным образом указать общее количество знакомест и количество знакомест, занимаемых дробной частью.

Пример

```
y=3;
printf("\n Значение переменной y=%10.7f",х);
```

Результат работы программы

```
Значение переменной у=3.000000
```

В программе 10 — общее количество позиций под значение переменной; 7 — количество позиций после десятичной точки.

Функция форматированного ввода данных с клавиатуры scanf выполняет чтение кодов, вводимых с клавиатуры, преобразует их во внутренний формат и передаёт программе. При этом программист может повлиять на правила интерпретации входных кодов с помощью спецификаций форматной строки.

Общая форма записи функции scanf

```
scanf ("строка_форматов", адрес_объекта_1, адрес_объекта_2, ..., адрес_объекта_n);
```

Строка форматов аналогична функции *printf*. Адрес объекта генерируется следующим образом: *&имя_объекта*. Строка форматов и список аргументов для функции обязательны.

Пример программы

```
scanf("%d", &m);
/* Ввести целое число и присвоить */
/* его значение переменной m. */

scanf("%lf", &x1);
/* Ввести значение переменной x1, */
/* имеющей тип double. */
```

§1.7. Операторы

Условный onepamop if

Общая форма записи

```
if(< выражение>)
<оператор 1>;
```

```
[else
<оператор 2>;]
```

Если выражение истинно, то выполняется < *оператор* 1>, если выражение ложно, то выполняется < *оператор* 2> (при наличии опции else).

Оператор *if* может быть вложенным.

Пример

```
if (key == 1)
  printf("\n Выбран первый пункт");
else
  if (key == 2)
    printf("\n Выбран второй пункт");
else
    printf("\n Первый и второй пункты не
    выбраны");
```

Возможно использование оператора if без опции else. При использовании обеих форм оператора if опция else связывается с последним оператором if.

Пример

```
if (key!= 1)
if (key == 2)
printf( "\n Выбран второй пункт");
else
printf("\n Первый и второй пункты не
выбраны");
```

Если < *оператор1>* или < *оператор2>* должны состоять из нескольких операторов, то необходимо использовать составной оператор (блок).

Пример

```
if (key == 1)
{
    n=n+1;
    m=l+r;
}
else
{
    m=m-1;
```

```
n=l-r;
}
```

Onepamop ветвления switch

Оператор if позволяет осуществить выбор только между двумя вариантами. Для того, чтобы производить выбор одного из нескольких вариантов используется оператор switch.

Общая форма записи

Оператор выполняется следующим образом:

- 1) вычисляется выражение в скобках оператора *switch*;
- 2) полученное значение сравнивается с метками (константными выражениеми) в опциях *case*;
- 3) сравнение производится до тех пор, пока не будет найдена метка, соответствующая данному значению, после этого выполнится оператор соответствующей ветви;
- 4) если соответствующая метка не найдена, то выполнится оператор в опции *default*.

Альтернатива *default* может отсутствовать, тогда не будет произведено никаких действий.

Опция *break* осуществляет выход из оператора *switch* и переход к следующему за ним оператору. При отсутствии опции *break* будут выполняться все операторы, начиная с помеченного данной меткой и кончая оператором в опции *default*.

Константые выражения (выражения, операнды которого константы) должны быть целого типа (включая char).

Пример. Разработать программу, определяющую день недели по его введенному номеру. Программа должна реагировать на неверно введенный номер дня недели.

Операторы цикла

В языке Си реализованы три вида операторов цикла:

- 1) while цикл с предусловием;
- 2) *do...while* цикл с постусловием;
- 3) for цикл с заданным числом повторений (цикл с предусловием).

Цикл while

Общая форма записи

```
while(<выражение>)
<оператор> ;
```

Если выражение истинно (т. е. не равно нулю), то выполняется оператор или группа операторов, входящих в цикл *while*; затем выражение проверяется снова. Последовательность действий, состоящая из проверки и выполнения оператора, повторяется до тех пор, пока выражение не станет ложным (т. е. равным нулю). После этого происходит выход из цикла, и далее выполняется оператор, стоящий после оператора цикла. При построении цикла *while*, в него необходимо включить конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным. Иначе выполнение цикла никогда не завершится.

Цикл *while* — цикл с предусловием, поэтому вполне возможно, что тело цикла не будет выполнено ни разу.

Пример

```
k=5;
n=10;
while(k<n)
{
    printf("k=%d n=%d\n", k, n);
    k+=2;
    k=k+2;
    n++;
}</pre>
```

Цикл do...while

Общая форма записи

```
do <oператор>;
while(<выражение>);
```

Цикл do...while — это цикл с постусловием, в котором истинность выражения проверяется после выполнения всех операторов, включенных в цикл. Тело цикла выполняется до тех пор, пока выражение не станет ложным, т. е. тело цикла выполнится хотя бы один раз. Использовать цикл лучше всего в тех случаях, когда должна быть выполнена хотя бы одна итерация.

Пример

```
printf("\n Введите число 10");

do scanf("%d",&number);
while(number !=10);
```

Пример

```
do
{
    printf(" Введите n>0");
    scanf("%d", &n);
}
while (n<0);
```

Цикл for

Общая форма записи

```
for( <инициализация>; <проверка условия>; <коррекция> ) <оператор>;
```

Цикл for — цикл с фиксированным числом повторений. Для организации такого цикла должны рассматриваться три операции:

- инициализация счетчика;
- сравнение его величины с некоторым граничным значением;
- изменение значения счетчика при каждом прохождении тела цикла.

Эти три операции записываются в скобках и разделяются точкой с запятой. Первое выражение служит для инициализации счетчика. Инициализация осуществляется только один раз — когда цикл for начинает выполняться. Второе выражение — для проверки условия перед каждым возможным выполнением тела цикла. Когда выражение становится ложным (равным нулю), цикл завершится. Третье выражение вычисляется в конце каждого выполнения тела цикла. Счетчик может как увеличиваться, так и уменьшаться.

Пример

```
main()
{
   int num;
   for( num=1; num<=5; num++ )
      printf(" % 5d % 5d \n", num, num*num);
}</pre>
```

В качестве третьего выражения может использоваться любое правильно составленное выражение, изменяющее значение проверяемого условия.

Пример

```
for(x=1;y<=35;y=5*x++ +10)
printf("%10d %10d \n", x, y);
```

Можно опустить одно или несколько выражений, но нельзя опускать точку с запятой. Параметры, находящиеся в заголовке цикла, можно изменить при выполнении операций в теле цикла.

Пример

В цикле *for* часто используется операция «запятая» для разделения нескольких выражений. Это позволяет включить в спецификацию цикла несколько инициализирующих или корректирующих выражений. Выражения, к которым применяется операция «запятая», будут вычисляться слева направо.

Пример

```
for(i=0, j=1; i<n; i++, j=i);
```

В Си допускаются вложенные циклы, т. е. когда один цикл находится нутри другого.

Пример

```
for( i=0; i<n; i++)
{
   for( j=0; j<n; j++)
   {
      . . .
}
}</pre>
```

Рекомендации по выбору цикла

При выборе цикла необходимо оценить следующие факторы. Нужен ли вам цикл с предусловием или цикл с постусловием (чаще используется цикл с предусловием). Если в цикле необходима инициализация и коррекция, то цикл for более предпочтителен. Этот цикл чаще всего используется при подсчете числа прохождений тела цикла с обновлением индекса, если в этом нет необходимости, то можно использовать while.

Операторы break и continue

В теле любого цикла можно использовать оператор *break*, который позволяет выйти из цикла, не завершая его. Оператор *continue* позволяет пропустить часть операторов тела цикла и начать новую итерацию.

Пример

Пример

При вложенных циклах действия операторов *break* и *continue* распространяется только на самую внутреннюю структуру, в которой они содержатся. Использование этих операторов возможно, но нежелательно, т.к. они ухудшают читаемость программы, увеличивают вероятность ошибок, затрудняют модификацию.

Пример

```
main()
{
  int i,j;
  float k;
```

```
printf(" Введите j");
scanf("%d", &j );
for( i = -5; i <= 5; i++)
{
    if( i==0 )
        continue;
    printf("\n %d/%d =%f ", j,i, k=j/i);
}</pre>
```

Оператор безусловного перехода дото

Общая форма записи

```
goto метка;
. . .
метка : <оператор> ;
```

Выполнение оператора *goto* вызывает передачу управления в программе оператору, помеченному меткой. Для отделения метки от оператора используется двоеточие. Помеченный оператор может появиться в программе как до оператора *goto*, так и после. Имена меток образуются по тем же правилам, что и имена переменных.

Пример

```
if ( size > 12)
    goto a;
else
    goto b;
a: cost = cost*3;
flag=1;
b: s= const*flag;
```

Использование *goto* в программе крайне нежелательно, так как он усложняет логику программы. Язык Си реализован таким образом, что можно программировать без оператора *goto*.

§1.8. Функции

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной задачи. Функция является подпрограммой, которая может содержаться в основной программе, а может быть

создана отдельно. Каждая функция выполняет в программе определенные действия. Использование функции позволяет, например, удобно организовывать вычисления с минимальными затратами сил и средств. После создания функции можно забыть, как она устроена, нужно лишь помнить, что она умеет делать.

Определение функции

Каждая функция в языке Си должна быть определена, т. е. должны быть указаны:

- тип функции;
- имя функции;
- информация о формальных параметрах;
- тело функции.

Существует два формата определения функции — современный стиль определения функции и старый стиль определения функции [5].

Современный стиль определения функции

```
тип_функции имя_функции (объявление формальных параметров через запятую)
{
    /* тело функции */
    ...
    return(<выражение>);
}
```

Старый стиль определения функции

```
тип_функции имя_функции (имена формальных параметров через запятую)
объявление формальных параметров;
{
    /* тело функции */
    ...
    return (<выражение>);
}
```

Пример

```
float function(float x, float z)
{
```

```
float y;
y=x+z;
return(y);
}
```

Различают *системные* (в составе систем программирования) и *собственные* функции.

Разбиение программ на функции дает следующие преимущества:

- Функцию можно вызывать из разных мест программы, что позволяет избежать повторного программирования.
 - Одну и ту же функцию можно использовать в разных программах.
- Функции повышают уровень модульности программы и облегчают её проектирование.
- Использование функций облегчает чтение и понимание программы, ускоряет поиск и исправление ошибок.

Функцию можно представить в виде некоторого «черного ящика», у которого несколько входов и один выход (рис. 1.1). В «черный ящик» поступает информация, обрабатывается и выдаётся результат. Для программиста неважно, что происходит внутри функции, ему необходимо только знать входы и выход. Использование подобного подхода позволяет на стадии проектирования сконцентрировать внимание на общей структуре программы.

В языке Си нет требования, чтобы определение функции обязательно предшествовало её вызову. Функции могут определяться как до функции *main*, так и после её. Нельзя определять одну функцию внутри другой.

Возврат значения в вызвавшую функцию осуществляется с помощью оператора *return*.

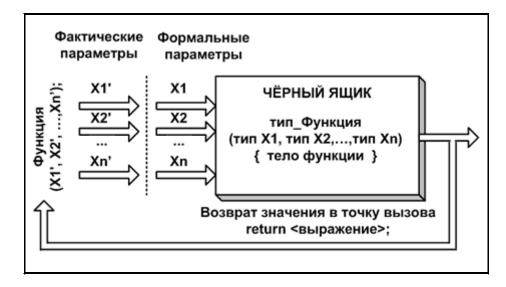


Рис. 1.1. Схема работы функции

Пример функции вычисления факториала

```
#include <stdio.h>
/* Определение функции factorial()*/
double factorial(double i)
   double j,k;
    k=1;
    for (j=2; j< i+1; j=j+1)
        k=k*j;
    return k;
/* Возврат в программу */
/*вычисленного значения */
}
/* Главная функция*/
main()
{
   double i;
   printf("\n Введите целое число\n");
   scanf("%lf",&i);
   printf("\n%lf!=%lf\n",i,
    factorial(i));
 }
```

Результат работы программы

```
Введите целое число
7
7.000000!=5040.000000
```

Вызов функций

Общий вид вызова функции

```
имя_функции(<список фактических параметров>);
```

Фактический параметр — это величина, которая присваивается формальному параметру при вызове функции. Таким образом, формальный параметр — это переменная в вызываемой функции, а фактический параметр — это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический параметр может быть константой, переменной или выражением. Фактический параметр сначала вычисляется, а затем его значение передается в

функцию. Если в функцию требуется передать несколько параметров, то они записываются через запятую [5].

Возврат значения в точку вызова

Функция может передать в вызывающую программу только одно значение, для этого используется оператор *return*.

```
return(<выражение>);
```

Действие оператора следующее: значение выражения, заключенного в скобки, вычисляется и передается в вызывающую функцию. Возвращаемое значение может использоваться в вызывающей программе как часть некоторого выражения.

Оператор *return* оказывает и другое действие. Он завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Оператор return не обязательно должен находиться в конце тела функции.

Пример использования return в середине функции

```
void function()
{
    ...
    return;
    ...
}
```

В примере оператор *return* завершает выполнение функции *function* и передает управление в вызывающую функцию. Никакое значение при этом не передается.

Вызов функции зависит от того, возвращает она значения или нет. Если *функция возвращает значения*, то её вызов производится из математических и логических выражений.

Пример функции определения максимума из двух чисел: $a = b^* \max(x,y)$;

```
int max(int a, int b)
{
   if (a>b)
     return (a);
   else
     return (b);
}
```

 Φ ункции могут и не возвращать значения, а просто выполнять некоторые вычисления. В таком случае указывается пустой тип данных void, и отсутствует оператор return.

Пример функции, не возвращающей значения: spravka(a);

```
void spravka (float a)
{
    ...
}
```

§1.9. Прототипы функций

В практике программирования бывают случаи, когда тело функции располагается в программе ниже функции её вызывающей или функция вообще компилируетсятся отдельно. В этом случае до использования функции должен быть указан прототи функции (или объявление функции), содержащий: а) тип функции, б) имя функции и в) информацию о параметрах.

Прототип необходим для того, чтобы кампилятор смог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров. *Объявление функции* имеет тот же вид, что и определение функции, однако тело функции отсутствует, и имена формальных параметров также могут отсутствовать.

Пример объявления функции модуля числа

```
int abs(int);
int abs(int i);
```

Если прототип не задан, то он будет построен по умолчанию на основе первой ссылки на функцию. Такой прототип не всегда может быть согласован с последующим определением или вызовом функции.

Рекомендуется всегда указывать прототип. Это позволяет кампилятору выдавать диагностичесие сообщения при неправильном использовании функции, либо корректировать несоответствие аргументов при выполнении программы.

При программировании на языке Си широко используются библиотечные функции. Эти функции были предварительно разработаны и записаны в состав системы программирования. Прототипы библиотечных функций находятся в специальных заголовочных файлах с расширением h (head), которые необходимо подключать с помощью директивы #include.

Рассмотрим пример программы генерации таблицы чисел 2^n .

Пример программы

```
#include <stdio.h>
 int power(int base; int index);
 /* int power(int, int); - второй вариант объявления
функции power().*/
main()
 {
    int i;
    for (i=0; i \le 10; i++)
       printf("%d, ", power(2,i));
 }
 int power(int base; int index)
    int i, p;
    p=1;
    for( i=0; i <= 10; i++ )
        p = p * base;
    return (p);
```

Результат работы программы

```
1, 2, 4, 8, 16, 32, 64, 128, 252, 504,
```

Схема программы представлена на рис. 1.2.

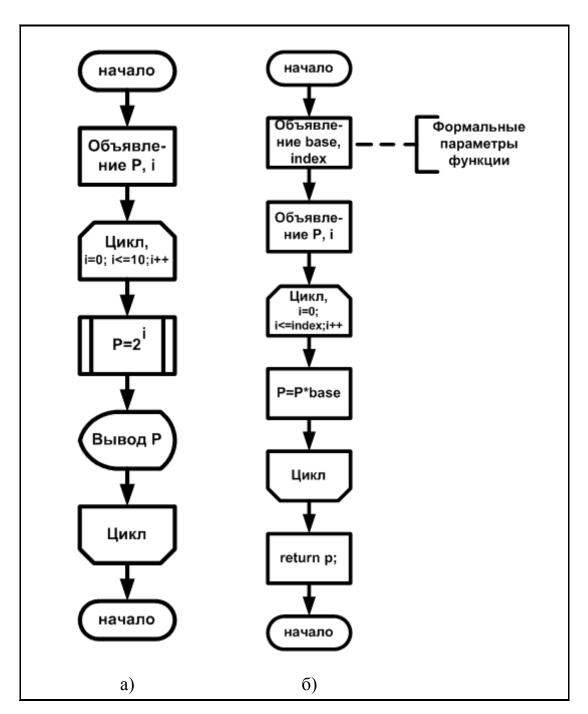


Рис. 1.2. Схема программмы: а — функция *main*; б — функция возведения в степень *power*

§1.10. Препроцессор

Препроцессор — это специальная программа, являющаяся частью компилятора языка Си и предназначенная для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения.

Работа препроцессора осуществляется с помощью специальных директив (команд). Они отмечаются знаком «решетка» — «#». Основными директивами препроцессора являются #include и #define.

Директива #include позволяет включать в текст программы указанный файл. Если файл находится в текущем каталоге, его можно записать в кавычках —

#include "func.c".

Можно также задать путь к файлу. Если имя файла записано в угловых скобках «<>», такой файл находится в системном каталоге, путь к которому задан в системе программирования. На нулевом этапе компиляции вместо директивы компилятор помещает файл.

Пример

```
#include <math.h>
#include "fact.c"
```

Директива #define позволяет вводить в текст программы макроопределения.

Общая форма записи

```
#define что_менять на_что_менять
```

Замена будет произведена на нулевом этапе компиляции. Символы «что менять» будут изменены на символы «на что менять».

Пример

```
#include <stdio.h>
#define pi 3.1459265
main()
{
    double x,y;

    printf("\n введите угол в радианах");
    scanf("%lf", &x);

    y=(180*x)/pi;

    printf("\n синус угла %lf в градусах %lf равен",
    y, sin(x));
}
```

§1.11. Математические функции

Математические функции стандартной библиотеки хранятся в головном файле <math.h>. Аргументы функции имеют тип double — тип с плавающей точкой двойной точности. Все математические функции также возвращают значения типа double. Углы в тригонометрических функциях задаются в радианах.

Основные математические функции стандартной библиотеки [4] приведены в табл. 2.2.

Таблица 1.2

Основные математические функции

Функция	Краткое описание
abs(x)	нахождение абсолютного значения выражения типа <i>int</i>
acos(x)	вычисление арккосинуса
asin(x)	вычисление арксинуса
atan(x)	вычисление арктангенса х
atan2(y, x)	вычисление арктангенса у/х
cabs(z)	нахождение абсолютного значения комплексного числа
ceil(x)	нахождение наименьшего целого, большего или равного x
cos(x)	вычисление косинуса
cosh(x)	вычисление гиперболического косинуса
exp(x)	вычисление функции экспоненты
fabs(x)	нахождение абсолютного значения х
floor(x)	нахождение наибольшего целого, меньшего или равного x
fmod(x, y)	нахождение остатка от деления x/y
hypot(x, y)	вычисление гипотенузы по двум заданным катетам x и y
labs(x)	нахождение абсолютного значения типа long
log(x)	вычисление натурального логарифма

Функция	Краткое описание
log10(x)	вычисление логарифма по основанию 10
matherr(x)	управление реакцией на ошибки при выполнении функций математической библиотеки
pow(x, y)	вычисление х в степени у
sin(x)	вычисление синуса
sinh(x)	вычисление гиперболического синуса
sqrt(x)	нахождение квадратного корня
tan(x)	вычисление тангенса
tanh(x)	вычисление гиперболического тангенса

§1.12. Специальные операции

В языке Си помимо основных операций — арифметических, логических, операций отношений — существуют ещё две специальные операции:

- 1) операция вычисление размера объекта sizeof;
- 2) операция «запятая» «, ».

Операция *sizeof* предназначена для определения объема оперативной памяти в байтах, необходимой для размещения объекта. Объектами могут быть типы данных, переменные и константы.

Общая форма записи

```
sizeof(объект)
```

Пример программы

```
#include <stdio.h>
#define pi 3.14159625
main()
{
   int x;
   printf("\n Размер памяти под целое число %d байт.",
   sizeof(int));
   printf("\n Размер памяти под тип double %d байт.",
```

```
sizeof(double));
printf("\n Размер памяти под переменную %d байт.",
sizeof(x));
printf("\n Размер памяти под константу pi %d байт.",
sizeof(pi));
}
```

Операция «запятая» (« , ») предназначена для связывания между собой выражений. Список, разделенный запятой, трактуется как единое выражение и вычисляется слева направо.

Пример

```
main()
{
   int x=3, y;
   y=3, 4*x;
   printf ("\n Значение y=%d.", y);
}
```

Результат работы программы

```
Значение у=12.
```

Пример

```
main()
{
   int i, b;
   for (i=0, b=1; i <= 5; i++)
   {
     b=b+i;
     printf("\n Значение b=%d.",b);
   }
}</pre>
```

Результат работы программы

```
Значение b=1.
Значение b=2.
Значение b=3.
```

```
Значение b=4.
Значение b=5.
Значение b=6.
```

§1.13. Глобальные и локальные объекты

Глобальными называются объекты, объявление которых дано вне функции. Они доступны (видимы) во всём файле, в котором они объявлены, а также во всех внешних файлах (модулях).

Локальными называются объекты, объявление которых дано внутри функции. Эти объекты доступны только внутри той функции, в которой они объявлены.

В Си существует понятие времени жизни объекта, которое бывает глобальным или локальным. Объект с глобальным временем жизни характеризуется тем, что в течение всего периода выполнения программы с ним связана определённая ячейка оперативной памяти и какое-то значение. Объекту с локальным временем жизни выделяется новая ячейка оперативной памяти при каждом вхождении в блок, в котором объявлен этот объект. Когда выполнение блока завершается, память, выделенная под локальный объект, освобождается, и объект теряет своё значение.

Пример программы

```
#include <stdio.h>
void autofunc(void)
{
   int k=1;
   printf("\n k = %u", k);
   k=k+1;
}

main()
{
   int i;
   for(i = 0; i<=5; i++)
      autofunc();
}</pre>
```

Результат работы программы

```
k = 1
k = 1
k = 1
k = 1
k = 1
k = 1
k = 1
```

Если в примере объявить переменную k как глобальную, результат работы программы будет иным.

Пример

```
#include <stdio.h>
int k=1;
void autofunc(void)
{
    printf("\n k = %u", k);
    k=k+1;
}

main()
{
    int i;

for (i = 0; i<=5; i++)
        autofunc();
}</pre>
```

Результат работы программы

```
k = 1
k = 2
k = 3
k = 4
k = 5
```

```
k = 6
```

Замечание: глобальные переменные нужны для того, чтобы организовывать обмен информацией между функциями.

Пример взаимодействия функций без использования глобальных переменных

Пример взаимодействия функций с использованием глобальных переменных

```
#include <stdio.h>
int x1, x2;
int func(void)
{
    ...
    y=x1+x2;
    ...
    return(y);
}
main()
{
    ...
    x1=...; /* изменение значений */
    x2=...; /* глобальных */
    /* переменных */
```

```
...
c=func(); /* вызов функции */
}
```

Программа имеет недостаток: функция *func* не имеет формальных параметров, обмен информацией между функциями организован через глобальные переменные x1, x2. Существует опасность, что переменные x1, x2 будут случайно изменены в вызываемой функции.

§1.14. Модификация объектов

Модификация или видоизменение объектов в языке Си применяется для того, чтобы расширить или, наоборот, сузить диапазон значений или область действия объекта. Инструкции, которые применяются для модификации, называются модификаторами.

Модификатор *unsigned*. Предназначен для того, чтобы объявлять переменные типов *short*, *int*, *long* беззнаковыми. Если переменную сделать беззнаковой, то при этом расширяется числовой диапазон абсолютного значения переменной. Это происходит из-за того, что один бит, который использовался под знак, используется под число.

Пример

```
-32768 <= int i <= 32768
0 <= unsigned int i <= 65535
```

Пример

```
unsigned int i;
unsigned long j;
```

Явная модификация типа данных применяется, когда необходимо явным образом изменить тип переменной, например, в том случае, когда результат вычисления выходит за границы ранее присвоенного переменной типа.

Пример

```
main()
{
   int i,j;
   long k;
   i= 30000;
   j= 20000;
   k= i+j; /*OMMBKA!!!*/
```

```
printf("\n%d+%d=%ld", i , j, k);
}
```

Несмотря на то, что k объявлена как long, результат вычисления выражения i+j

получается типа int, поскольку i и j объявлены как int. В то же время значение выражения выходит за границу типа int. Для того, чтобы не было ошибки, необходимо модифицировать переменные следующим образом:

```
k=(long)i+(long)j;
```

Таким образом, если необходимо явным образом изменить тип данных, который используется в выражениях, то перед объектами в круглых скобках нужно указать тот тип, который необходимо получить.

Модификатор *extern* предназначен для использования в данном программном модуле объекта, который объявлен в другом отдельном модуле.

Пример

```
extern_тип объект_1, объект_2, ... ,объект_n;
```

Модификация расположения объектов в оперативной памяти

Модификатор *static*. При выполнении программы объекты могут быть расположены либо по фиксированным адресам оперативной памяти, либо по произвольным адресам по мере того, как эти объекты появляются.

Если объект расположен по некоторому фиксированному адресу, то он называется *статическим* (типичный пример — глобальные переменные).

Объект, который располагается в произвольном месте оперативной памяти, называется *динамическим* (пример — все локальные объекты). Если необходимо динамический объект сделать статическим, то используется модификатор *static*.

Переменные, объявленные с модификатором *static* сохраняют свои значения при входе и выходе из функции, однако не являются глобальными.

Пример

```
#include <stdio.h>
void stat(void)
{
   static int k=1;
   printf ("\t k=%d",k);
   k++;
}
```

```
main()
{ int i;

for( i=0; i<5; i++)
    stat();
}</pre>
```

Результат работы программы

```
k=1 k=2 k=3 k=4 k=5
```

Переменная k в функции stat зафиксирована в оперативной памяти. Инициализация k проводится только один раз — при первом вызове функции переменной k присваивается значение 1. При втором обращении k функции k инициализация переменной k не будет производиться и на экран выведется 2. Значение k сохраняется в оперативной памяти, однако переменная не доступна из функции k

Модификатор *register*. Модификатор предназначен для того, чтобы поместить переменную в один из регистров общего назначения процессора. Благодаря этому повышается скорость работы с данными. Это необходимо для создания управляющих программ, требующих высокую скорость обработки данных.

Пример

```
register int i;
```

Объявление переменных с модификатором *register* приводит к помещению переменной в регистр, если регистр свободен.

Модификатор *const* предназначен для объявления объектов как неизменяемых. Объекту, который объявлен с данным модификатором, можно присвоить только одно значение.

Пример

```
const double pi = 3.14159265;
```

Если программист попытается изменить значение pi, то компьютер выдаст ошибку.

§1.15. Указатели

Указатель — переменная, содержащая адрес объекта. Указатель не несет никакой информации о самом объекте, а содержит сведения о том, где размещен

объект. Указатели широко используются при программировании на языке Си. Программы с указателями — короткие и очень эффективные.

Указатели применяются:

- 1) для доступа к ячейкам оперативной памяти и создания новых объектов в ходе выполнения программы;
 - 2) для доступа к сложным элементам данных;
 - 3) для выполнения различных операций с элементами массива;
 - 4) и т. д.

Понятие «указатель» можно пояснить, используя упрощенную схему организации памяти ЭВМ (рис. 1.3). Как правило, память ЭВМ можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками. Указатель — это тоже переменная, которая размещается в памяти. Обычно указатели занимают 2 или 4 байта — в зависимости от модели памяти. На рис. 1.3 переменная с имеет тип char; указатель p содержит адрес c. Взаимосвязь переменных p и c показана стрелкой.

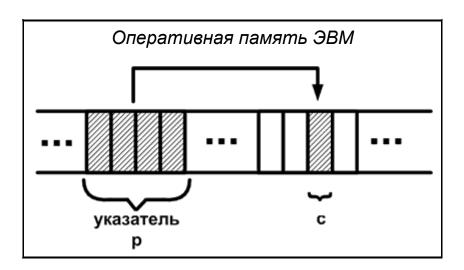


Рис. 1.3. Графическое представление указателя Указатель, как и любая переменная, должен быть объявлен.

Общая форма объявления указателя

Пример объявления указателя

```
int *p;
char *p;
```

Тип указателя — это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

- 1) операция «*» (звездочка) позволяет сгенерировать значение объекта по его адресу;
 - 2) операция «&» (амперсант) позволяет определить адрес объекта.

Операция «&» выдает адрес объекта, так что запись

$$p=\&c$$

присваивает переменной p адрес ячейки, в которой размещается значение переменной c.

Операция «*», применённая к указателю, выдает значение объекта, на который данный указатель ссылается, например:

$$k=*p;$$
 —

переменной k присваивается значение, размещенное по адресу, содержащемуся в указателе p.

Пример

Результат работы программы

```
Значение переменной а равно 134.
Адрес переменной а равен 0012FED4.
Данные по адресу указателя b равны 134.
Значение указателя b равно 0012FED4.
Адрес расположения указателя b равен 0012FEC8.
```

§1.16. Модели памяти

При разработке программ *для персональных компьютеров*, *оснащенных микропроцессорами серии x86*, размер указателя (число байтов, требуемых для размещения адреса памяти) зависит от модели памяти, задаваемой в компиляторе языка Си.

<u>Системы программирования для шестнадцатибитной среды DOS</u> предоставляют шесть стандартных моделей памяти [4]:

- 1) крошечную (*tiny*);
- 2) маленькую (*small*);
- 3) среднюю (*medium*);
- 4) компактную (сотраст);
- 5) большую (*large*);
- 6) огромная (*huge*).

Всю память, которая требуется для выполнения программы на языке Си, можно разделить на четыре части:

- 1) память под программный код;
- 2) память под статические объекты;
- 3) память под динамические объекты;
- 4) память под стек, в который заносятся параметры функций.

Каждая модель имеет определенные объемы этих частей и выбирается в зависимости от сложности задачи, размера программы и объема данных.

Крошечная модель. Под код программы, статические и динамические объекты, а также под стек отводится в общей в общей сложности 64 Кбайт памяти. Такая модель налагает на программу серьезные ограничения и используется только в тех случаях, когда особенно ощущается дефицит памяти. Переменные типа указатель в такой модели памяти занимают два байта (близкие указатели).

Маленькая модель. Под код программы отводится сегмент памяти размером 64 Кбайт. Статические, динамические объекты и стек занимают в общей сложности также 64 Кбайт памяти. Такая модель памяти принимается по умолчанию и вполне подходит для многих маленьких и средних задач. Переменные типа указатель занимают в такой модели два байта (близкие указатели).

Средняя модель. Код программы размещается в сегменте памяти размером 1 Мбайт. Статические, динамические объекты и стек занимают 64 Кбайт памяти. Такую модель рекомендуется использовать для программирования больших задач, имеющих малый объем данных. Для адресации в коде программы используются далекие указатели, занимающие 4 байта (все вызовы функций и возвраты из функций — далекие); для адресации данных используются близкие указатели, занимающие два байта.

Компактная модель. Под код программы отводится 64 Кбайт. Под статические, динамические объекты и стек отводится 1 Мбайт, но размер области памяти под статические объекты ограничивается величиной 64 Кбайт, под стек —

также 64 Кбайт. Такая модель применяется для программирования малых и средних задач с большим количеством данных. Для адресации в коде программы используются близкие (двухбайтовые) указатели; для адресации данных используются далекие (четырехбайтовые) указатели.

Большая модель. Под код программы отводится 1 Мбайт памяти. Под статические, динамические объекты и стек также отводится 1 Мбайт, однако статические объекты и стек занимают по 64 Кбайт памяти. Данная модель используется только для создания очень больших программных продуктов, в ней все указатели — далекие.

Огромная модель аналогична большой модели, но статические объекты могут занимать объем более 64 Кбайт. В модели используются далекие указатели.

<u>Приложения для шестнадцатибитной среды Windows</u> можно компилировать в расчете на четыре стандартные модели памяти [6]:

- 1) маленькую (*small*);
- 2) среднюю (*medium*);
- 3) компактную (сотраст);
- 4) большую (*large*).

Отличительной особенностью всех приложений Windows для шестнадцатибитной среды является то, что сегмент не содержит реальный адрес памяти. Вместо этого сегмент содержит только индекс (селектор), указывающий на строку в таблице (таблице дескрипторов), где этот адрес хранится. Для шестнадцатибитной же среды DOS процессор аппаратно суммирует значение сегментного регистра с указанным смещением, чтобы получить линейный адрес в оперативной памяти.

<u>Работа с памятью в тридцатидвухбитной среде Windows.</u> В тридцатидвухразрядных программах всегда используется *сплошная* (непрерывная) память [6]. Управление этой памятью осуществляют интегрированная среда программирования и операционная система.

§1.17. Массивы

Большинством объектов языка Си, с которыми мы имели дело, были переменные. Каждая переменная при объявлении получала тип и имя, с которым связывалась определенная ячейка памяти. Однако расположение значений переменных по адресам памяти никак не упорядочивалось. При решении многих задач, особенно с большим количеством однотипных данных, использование переменных с различными именами, а значит не упорядоченных по адресам памяти, затрудняет или делает вообще невозможным програмирование. В подобных случаях в языке Си используют объекты, называемые массивами.

Массив — это упорядоченная последовательность величин, обозначаемая одним именем. Упорядоченность заключается в том, что элементы массива располагаются в последовательных ячейках памяти. Можно провести аналогию между ячейками памяти, отведенными под массив, и таким же количеством контейнеров, скрепленных между собой (рис. 1.4). На всю конструкцию

контейнеров повешен ярлык с именем массива *data*. Контейнеры в такой конструкции пронумерованы начиная с нуля и представляют набор восьмибитных ячеек оперативной памяти, в которых хранятся элементы массива. Номер контейнера — значение индекса элемента массива. Чтобы получить доступ к нужному контейнеру, нужно указать имя массива и его индекс, который пишется в квадратных скобках.

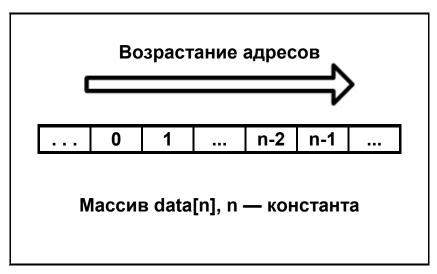


Рис. 1.4

Пример

```
data[2]=32;

/*Второму элементу массива */
/*с именем data присваивается */
/*значение 32.*/
```

Элементы массива могут употребляться в программе так же, как и простые переменные.

При объявлении массива нужно обязательно указать общее количество элементов, чтобы ЭВМ могла выделить память под весь массив.

Общая форма объявления массива

```
тип имя[размер массива];
```

Пример

```
float data[245];
```

Здесь массив содержит 245 элементов типа *float*: *data*[0], *data*[1], *data*[2], ..., *data*[244].

Связь массивов и указателей

Имя массива фактически является константой-указателем на начальный адрес данных — на адрес расположения элемента массива с нулевым индексом.

Графическое представление массива в памяти ЭВМ представлено на рис. 1.5, где data — адрес начала массива; sizeof(data) — размер массива data в байтах; sizeof(float) — размер памяти под один элемент массива в байтах; p1 и p2 — указатели для работы с массивом.

Начальный адрес массива определяется компилятором в момент его объявления, и такой адрес никогда не может быть изменен. Адрес массива можно узнать, если вывести на экран значение константы с именем массива или вывести адрес нулевого элемента массива. Это значение можно присвоить указателю, имеющему другое имя, а затем, наращивая значение этого указателя, обращаться по выбору к любому элементу массива. Следовательно, в ряде случаев операции с массивами можно свести к операциям с указателями.

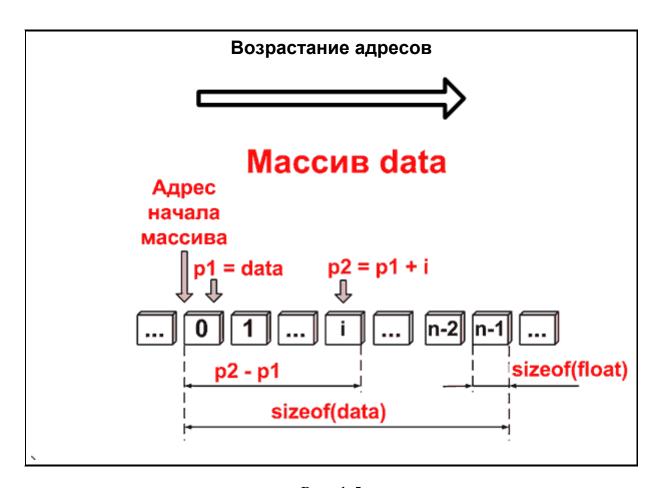


Рис. 1.5

Форма обращения к элементам массива с помощью указателей следующая:

- $\bullet \, data[0] == *p1;$
- data[1] == *(p1 + 1);
- data[2] == *(p1 + 2);

```
• data[i] == *(p1 + i);
• data[n-1] == *(p1 + n − 1).
Операции над указателями p1 и p2:
• p1 == p2 — проверка на равенство;
• p1!= p2 — проверка на неравенство;
• p1 < p2 — сравнение указателей на меньше;</li>
• p1 <= p2 — сравнение указателей на меньше или равно;</li>
• p1 > p2 — сравнение указателей на больше;
• p1 >= p2 — сравнение указателей на больше или равно;
• p1 >= p2 — сравнение указателей на больше или равно;
• p1 >= p1 — вычисление количества элементов между p2 и p1;
• p1 + i — наращивание указателя на i элементов;
• p1 − i — уменьшение указателя на i элементов.
```

При работе с указателями и массивами следует внимательно следить за тем, чтобы адреса, хранимые в указателях, не выходили за рамки адресов массивов.

Пример. Связь между указателями и именами массивов

```
#include <stdio.h>
main()
{
   int i;
   float data[5], *p;
   printf("\n Начальный адрес массива data - %u .",
   data);
   printf("\n Адрес элемента массива data[0] - %u .",
   &data[0]);
   printf("\n Адрес третьего элемента массива - %u .",
   &data[3]);
   p=data; /* p= & data [0]; */
   for (i=0; i<=4; i=i+1)
      printf("\n Адрес %d-го элемента массива data
      равен %u .", i+1, p+i);
      printf("\n Значение %d-го элемента массива
      data равно %f .", i+1, &data[i]);
   }
```

Инициализация массивов

Существует два способа инициализации массива:

- 1) указание начальных значений при объявлении массива;
- 2) организация цикла последовательного ввода значений элементов массива.

Общая форма инициализации массива при объявлении

```
тип имя[n]={ значение_1, значение_2, значение_3, ..., значение_n};
```

Начальные значения элементов массива заключаются в фигурные скобки. Если программист не указал в квадратных скобках размер массива, то компилятор сам задаст размер по количеству приведенных значений. Если начальные значения в фигурных скобках не заданы, то все элементы массива будут нулевыми.

Пример инициализации массива

```
# include <stdio.h>
main()
{
   int data[5]={5, 4, 3, 2, 1};
   float a[]={3, 4, 2, 7, 18, 90};
   char p[]={'f', 't', 'c', 't', '\0'};
   char *q;

   q=p;

   printf("\n Размер массива data %u байт.",
   sizeof(data));
   printf("\n Размер массива a%u байт.", sizeof(a));
   printf("\n Размер массива p%u байт.", sizeof(p));
   printf("\n Адрес массива p равен %p.", p);
   printf("\n Адрес в указателе q равен %p.", q);
}
```

Цикл последовательного ввода значений элементов массива удобно организовать с помощью оператора for.

Пример

```
#include <stdio.h>
    main()
{
    int data[5], *p, j;
    p=data;

    for (j=0; j<=4; j++)
    {
        printf("\n Введите элемент массива");
        printf("data[%d] = ",j);
        scanf("%d", p+j);
    }
...</pre>
```

§1.18. Передача массива в функцию

Обработку массивов удобно организовывать с помощью специальных функций. Если в функцию передается массив, то на самом деле внутрь функции попадает только адрес массива. Фактически в функцию передаются: 1) адрес массива, 2) размер массива. Исключение составляют функции обработки строк, в которые передаются только адреса.

Обычно параметры функций в языке Си передаются как копии. Если внутри функции произойдет изменение значения параметра, то это никак не отразится на его первоначальном (оригинальном) значении. С массивами дело обстоит иначе. Если в функцию передается адрес массива (адрес нулевого элемента), то все операции, выполняемые в функции над массивом, производятся над оригиналом; поэтому исходный массив может быть случайно поврежден.

Рассмотрим пример программы, в которой функция *modify* увеличивает на 1 значение каждого элемента массива *data*.

```
#include <stdio.h>
void modify(int * a, int size)
{
   int i;

   for (i=0; i< size; i++)
       *(a+i) =*(a+i) + 1;
       /* a[i] = a[i]+1;*/
}</pre>
```

```
main()
{
  int data[] = {-12, 14, -10, 16, 22};
  int i;

  printf("\n Исходный массив: ");

  for (i=0; i<5; i++)
     printf("\n data[%d] =%d", i,data[i]);

  modify(data, 5);

  printf("\n Модифицированный массив: ");

  for(i=0; i<5; i++)
     printf("\n data[%d]=%d", i, data[i]);
}</pre>
```

§1.19. Многомерные массивы

В языке Си массивы бывают не только одномерные, но и многомерные. Отличие состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном — несколькими.

Общая форма объявления многомерного массива

```
тип имя_массива [индекс_1]
[индекс_2]
.....
[индекс_n];
```

Элементы многомерного массива располагаются в последовательных ячейках оперативной памяти по возрастанию адресов. Между элементами нет разрывов. В памяти ЭВМ элементы располагаются построчно таким образом, что быстрее всего меняется последний индекс.

Пример расположения в памяти ЭВМ двухмерного массива *int d*[3][4]

```
d[0][0] d[0][1] d[0][2] d[0][3]
d[1][0] d[1][1] d[1][2] d[1][3]
d[2][0] d[2][1] d[2][2] d[2][3]
```

Пример расположения в памяти ЭВМ трехмерного массива *int* d[2][2][2]

```
d[0][0][0] d[0][0][1]
d[0][1][0] d[0][1][1]
d[1][0][0] d[1][0][1]
d[1][1][0] d[1][1][1]
```

Применение двухмерных массивов. Двумерные массивы, как правило, отождествляются с матрицами, поэтому типовыми задачами, в которых применяются двухмерные массивы, являются задачи с матрицами: транспонирование матриц, вычисление определителя, определение средних значений матрицы и так далее.

Пример. Разработать программу вычисляющую сумму элементов по строкам и столбцам квадратной матрицы 3х3.

```
#include <stdio.h>
#define i 3
#define j 3
main()
  int data[i][j],s;
   int k,l;
   printf("\n Введите массив data \n");
   for (k=0; k<i; k++)
       printf("\n");
       for (1=0; 1<j; 1++)
           printf("data[%d][%d] = ", k, 1);
           scanf("%d", &data[k][1]);
       }
   }
   for (k=0; k<i; k++)
       s = 0;
       for (1=0;1<j;1++)
           s=s+data[k][l];
       printf("\nСумма элементов %d-й строки равна %d",
       k+1, s);
   }
```

```
for (l=0; l<j; l++)
{
    s=0;
    for (k=0; k<i; k++)
        s=s+data[k][l];

    printf("\nСумма %d столбца равна %d.", l+1, s);
}

getchar();
getchar();
}
```

§1.20. Динамическое распределение памяти

При программировании на языке Си может возникнуть ситуация, когда формально правильно написанная программа содержит некоторую серьёзную ошибку, которая может настолько повредить работе компьютера, что потребуется его перезагрузка. Поскольку программа написана формально верно, то компилятор ошибку не распознаёт. Рассмотрим пример подобной программы.

```
#include <stdio.h>
main()
{
   int *x,*w,y,z;
   y=-15;
   *x=16;
   w=&y;

   printf("\n Размер указателя х составляет %d байт.",
    sizeof(x));
   printf("\n Значение указателя х равно %u байт.", х);
   printf("\n Значение по этому адресу равно %d.",*x);
   printf("\n Адрес переменной у равен %u.", &y);
   printf("\n Адрес переменной z равен %u.", &z);
   printf("\n Значение *w равно %d.", *w);
}
```

Ошибка, содержащаяся в программе, одна из наиболее распространенных. По отдельности каждая строка программы правильна. Проблема заключается в не инициализированном указателе x. Объявление int *x приведёт к тому, что под переменную x выделется блок оперативной памяти. В нашем случае не выделен участок оперативной памяти для размещения числа 16. Получается, что 16 будет записано в неизвестную область памяти: поскольку переменная x не

инициализирована, то в ней хранится неизвестное число, и запись *x=16 приводит к тому, что по неизвестному адресу расположится число 16. Это может привести к следующему: 1) стирание части кода операционной системы, 2) стирание части кода драйвера или другой программы, 3) стирание кода самой выполняемой программы. Компилятор такую ошибку распознать не сможет.

Исправить ситуацию можно, если использовать функцию динамического распределения памяти *malloc*, прототип которой хранится в головном файле alloc.h. Функция *malloc* выделит в оперативной памяти два байта, в которых можно разместить целое число 16, и адрес этих двух байтов гарантированно не совпадает ни с одним из адресов, используемых операционной системой.

Таким образом, в исходную программу необходимо добавить две строки: 1) директиву препроцессора #include < alloc.h> u 2) непосредственно перед оператором *x=16; строку $x=(int^*)malloc(sizeof(int))$;.

```
#include <stdio.h>
#include <alloc.h>
main()
 {
    int *x, *w, y, z;
    y = -15;
    x=(int*) malloc(sizeof(int));
    *x=16;
    w=&y;
   printf("\n Размер указателя х составляет %d байт.",
    sizeof(x));
   printf("\n Значение указателя х равно %u байт.", х);
   printf("\n Значение по этому адресу равно %d.", *x);
   printf("\n Адрес переменной у равен %u.", &y);
   printf("\n Адрес переменной z равен %u.", &z);
   printf("\n Значение *w равно %d.", *w);
```

Описание функции *malloc*. Функция *malloc* динамически распределяет блок памями размером *size* байт. Возвращаемое значение — указатель на выделенную область памяти.

```
#include <alloc.h>
void *malloc(size);
unsigned size;
```

§1.21. Динамическое распределение памяти под массивы

Динамическое распределение памяти под массивы необходимо использовать в том случае, когда размер массива заранее не известен. Это означает, что размер массива будет определяться в ходе выполнения программы.

Пример 1. Рассмотрим работу с массивом без динамического распределения оперативной памяти.

```
main ()
{
    float b[100];
    int n, i;

    printf("Введите размер массива \n, n<=100");
    scanf("%u", &n);

    for(i=0; i<=n-1; i++)
    {
        /*Инициализация массива.*/
        printf("\nВведите элемент массива b[%u]=", i+1);
        scanf("%f", b+i);
    }
    . . .
}</pre>
```

Недостаток программы заключается в том, что мы резервируем больше оперативной памяти, чем будем использовать. Память в данном случае расходуется неэкономно.

Пример 2. Работа с массивом с использованием динамического распределения оперативной памяти.

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

main()
{
   float *b;
   int n;

   printf ("\nВедите размер массива \n , n= ");
   scanf("%u", &n);

b=(float*)malloc(n*sizeof(float));
```

```
for(i=0;i<=n-1;i++)
{
    printf("\nВведите элемент массива b[%u]=", i+1);
    scanf("%f", &b[i]);
}
...</pre>
```

В программе объявляется указатель с именем b, который содержит адрес переменной типа float. Оператор программы вводит размер массива n. С помощью функции malloc выделяется память под массив из n чисел типа float и инициализируется указатель b. Память компьютера при этом расходуется экономно.

Графическое представление выделения памяти под массив представлено на рис. 1.6.



Рис. 1.6

Динамическое размещение двухмерных массивов. Допустим нам необходимо разместить в оперативной памяти матрицу, которая содержит вещественные числа. Размерность матрицы n^*m , где n — количество строк; m — количество столбцов. Для размещения матрицы необходимо зарезервировать, с помощью функции malloc, блок памяти размером sizeof(float)*n*m байт. Для работы с матрицей определим следующее правило для передвижения по блоку оперативной памяти:

$$*(b + i*n + j),$$

где i — номер строки $(0 \le i \&\& i \le n); j$ — номер столбца $(0 \le j \&\& j \le m)$. Таким обрахзом будет осуществлятся доступ к элементу массива с индексами i и j.

Графическое представление размещения двухмерного массива в оперативной памяти представлено на рис. 1.7.

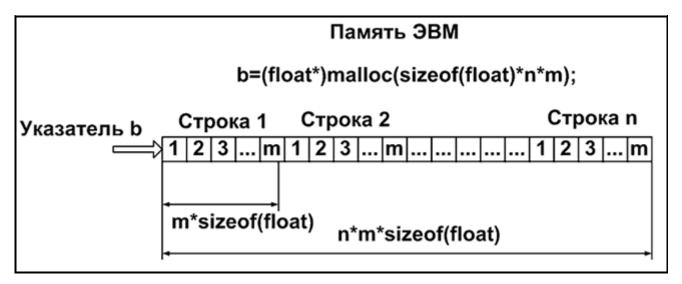


Рис. 1.7

Программа

```
#include <stdio.h>
#include <alloc.h>
main()
    float *b;
    int n, m, i, j;
    printf("\n Введите количество строк: n= ");
    scanf("%u", &n);
    printf("\n Введите количество столбцов: m= ");
    scanf("%u", &m);
    b=(float*)malloc( sizeof(float)*m*n );
    for (i=0; i \le n-1; i++)
    for (j=0; j \le m-1; i++)
        printf ("\nВведите элемент b[%u][%u] = ",
        i+1, j+1);
        scanf ("%f", b+i*n+j);
    }
    for (i=0, i \le n-1, i++)
    for (j=0, j \le m-1, j++)
    {
                                 /*обработка массива*/
        * (b+i*n+j) = ...
    }
```

§1.22. Массивы указателей

Свободным называется двухмерный массив (матрица), размер строк которого может быть различным. Преимущество использования свободного массива заключается в том, что не требуется отводить память ЭВМ с запасом для размещения строк максимальной возможной длины. Фактически, свободный массив представляет собой одномерный массив указателей на одномерные массивы данных.

Рассмотрим вначале как можно разместить в оперативной памяти матрицу со строками одинаковой длины. Чтобы создать в памяти ЭВМ двухмерный массив необходимо: 1) выделить блок оперативной памяти под массив указателей, 2) выделить блоки оперативной памяти под строки матрицы — одномерные массивы и 3) записать адреса строк в массив указателей. Графическое представление такой структуры хранения информации представлено на рис. 1.8. Обращение к элементу массива будет осуществлятся следующим образом:

b[i][j],

где i — номер строки; j — номер столбца.

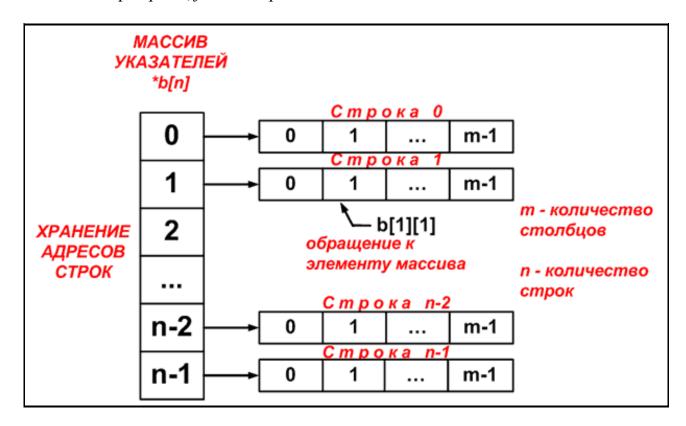


Рис. 1.8

Фрагмент программы реализации двухмерного массива со строками одинаковой длины показан ниже.

```
#include <alloc.h>
#include <stdio.h>
main()
   float **b;
   int n, m, i, j;
   printf ("\n Введите количество строк: n=");
   scanf ("%d", &n);
   printf ("\n Введите количество столбцов: m=");
   scanf ("%d", &m);
   b=(float**)malloc( n*sizeof(float*) );
   /*Выделяется блок оперативной памяти */
   /*для хранения адресов строк матрицы.*/
   /* Используется двойная косвенная адресация. */
   for(i=0; i<=n-1; i++)
      b[i]=(float*)malloc( n*sizeof(float) );
      /*В цикле динамически выделяются блоки*/
      /*оперативной памяти под строки. Адреса */
      /*строк записываются в массив указателей b.*/
   for (i=0; i \le n-1; i++)
   for (j=0; j <= m-1; j++)
      printf ("\n введите элемент b[%d][%d]=",
      i+1, j+1);
      scanf ("%f", &b[i][j]);
   }
   for (i=0; i \le n-1; i++)
   for (j=0; j <= m-1; j++)
       b[i][j]= /* обработка массива */
    }
```

Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив m, в котором будут храниться

размеры строк. Фрагмент программы, в которой реализуется динамическое размещение такой матрицы приведен ниже [12].

```
#include <stdio.h>
#include <alloc.h>
main()
   float **b;
   int i, j, n;
   int *m;
   printf("\n Введите количество строк: n= ");
   scanf("%u", &n);
   b=(float **)malloc( n*sizeof(float*) );
   m=(int *)malloc( n*sizeof(int) );
   for (i=0; i \le n-1; i++)
      printf("\n Введите длину строки: m[%u]=", i+1);
      scanf("%u", m+i)
   }
   for(i=0; i<=n-1; i++)
      b[i]=(float*)malloc( m[i]*sizeof(float) );
   for (i=0; i \le n-1; i++)
   for (j=0; j \le m[i]-1; j++)
       printf("\n Введите элемент b[%u][%u]= ",
       i+1, i+1);
       scanf("%f", &b[i][j]);
   }
   for (i=0; i \le n-1; i++)
   for (j=0; j \le m[i]-1; j++)
     b[i][j] = ...
   }
```

Графическое представление двухмерного массива со строками разной длины представлено на рис. 1.9.

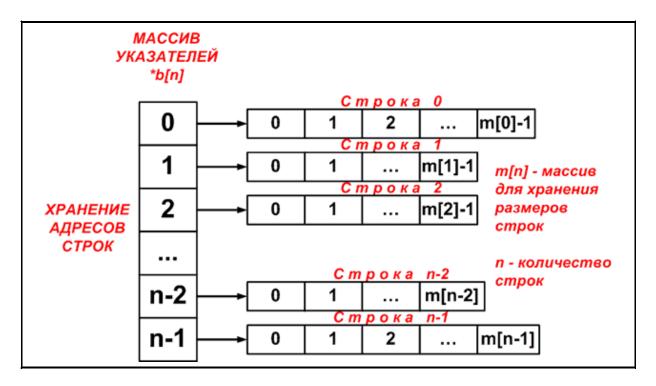


Рис. 1.9

§1.23. Структуры

Структура — это объединение одного или нескольких объектов, возможно различного типа, под одним именем, которое является типом структуры. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.

Структуры позволяют трактовать группу связанных между собой объектов не как множество отдельных элементов, а как единое целое.

Пример — строка платёжной ведомости, которая содержит следующие сведения о работнике: полное имя, адрес, зарплату и так далее.

Таким образом, структура — сложный тип данных, составленный из простых типов.

Общая форма объявления структуры

После закрывающей фигурной скобки « } » в объявлении структуры обязательно ставится точка с запятой.

Пример объявления структуры

```
struct date
{
   int day;
   int month;
   int year;
};
```

Элементы структуры располагаются в памяти ЭВМ в том же порядке, в котором они объявляются.

При объявлении структур, их разрешается вкладывать одну в другую.

Пример

```
struct persone
{
   char fam[20];
   char im[20];
   char ot[20];
   struct date bd;
};
```

Здесь одним из элементов структуры persone является структура типа date с именем структурной переменной bd (birthday).

Инициализация структуры может осуществляться двумя способами:

- присвоение значений элементам структуры в процессе объявления переменной, относящейся к типу структуры;
- \bullet присвоение начальных значений элементам структуры с использованием функций *printf* и *scanf*.

В первом случае инициализация осуществляется по следующей форме:

```
struct тип_структуры имя_переменной= \{ значение_элемента_1,  значение_элемента_2,  . . . ,  значение_элемента_n\}; .
```

Второй случай не отличается от способа инициализации объектов языка Си уже известных типов.

Для обращения к элементу структуры нужно указать не только имя самого элемента, но и имя переменной. Они разделяются точкой:

```
имя переменной.имя элемента.
```

Имя структурной переменной может быть указано при объявлении структуры. В этом случае оно размещается после закрывающейся фигурной скобки.

Пример

```
struct complex_type
{
    double real;
    doubl imag;
} number;
```

В этом примере элементами структуры будут переменные *number.real* и *number.imag*.

Пример 1. Объявление и инициализация структуры

```
/* Объявление структуры computer,*/
/* состоящей из двух элементов: model и memory.*/
struct computer
{
   char model[30];
   int memory;
};

/* Объявление и инициализация переменной*/
/* Elecom типа computer.*/
struct computer Elecom = {"IBM PC Pentium 4", 1024};
```

Пример 2. Объявление и инициализация структуры

```
#include <stdio.h>
main()
{
    /*Объявление структуры типа data. */
    struct data
    {
       int day;
       char month[10];
```

```
int year;
};
/*Объявление структуры типа person; */
/*одним из элементов структуры person*/
/*является структурная переменная bd типа data.*/
struct person
     char fam[20];
     char im[20];
     char ot[20];
     struct data bd;
};
/*Объявление структурной переменной ind1*/
/*типа person.*/
struct person ind1;
printf ("\n Укажите через Enter фамилию, имя,
         отчество,");
printf ("\n День, месяц и год рождения гражданина
         ind1\n");
/* Ввод сведений о гражданине ind1. */
scanf ("%s%s%s%d%s%d", indl.fam, indl.im, indl.ot,
&indl.bd.day, &indl.bd.month. &indl.bd.year);
/* Вывод сведений о гражданине ind1. */
printf ("\n\n Сведения о гражданине ind1:\n\n");
printf ("Фамилия, имя, отчество: \t%s %s %s\t \n",
         ind1.fam, ind1.bd.month, ind1.bd.year);
```

§1.24. Объединения

Объединениями называют сложный тип данных, позволяющий размещать в одном и том же месте оперативной памяти данные различных типов. Естественно, что в данный момент времени в данном месте памяти может быть размещено значение только одного включенного в объединение типа.

Размер оперативной памяти, требуемый для хранения объединения, определяется размером памяти, необходимым для размещения данных того типа, который требует максимального количества байт.

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется одна и та же область памяти. Когда используется элемент меньшей длины, чем наиболее длинный элемент объединения, то этот элемент использует только часть отведенной памяти. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Общая форма объявления объединения

```
union имя_объединения
{
    тип имя_объекта_1;
    тип имя_объекта_2;
    . . .
    тип имя_объекта_n;
} имя_переменной;
```

Видно, что объединение подобно структуре, однако в определённый момент времени может использоваться только один из элементов объединения. Объединения применяются для следующих целей:

- 1) для инициализации используемого объекта памяти, если в каждый момент времени только один из многих объектов является активным;
- 2) для интерпретации основного представления одного типа, как если бы этому объекту был присвоен другой тип.

Значение текущего элемента объединения теряется, когда другому элементу объединения присваивается значение.

Пример

```
#include <stdio.h>
union data_record
{
    char s[4];
    int i;
    char c;
} data;

main()
{
    printf("\nРазмер памяти под объединение data_record
    %d байт\n", sizeof(data));
    printf("\nАдреса переменных: \ns \t %p\n%i \t %p \n
```

```
%c \t %p \n", data.s,&data.i,&data.c);
}
```

Результат работы программы

```
Размер памяти под объединение data_record 4 байт
Адреса переменных:
s 0552
i 0552
c 0552
```

§1.25. Битовые поля

Используя структуры, можно упаковать целочисленные компоненты ещё более плотно, чем это было сделано с использованием массива. При упаковке целых объектов, как правило, допускается применять только беззнаковый тип *unsigned int*. Это обусловлено тем, что место, отводимое под объект, используется полностью, включая знаковый разряд.

В Си под объект типа *unsigned int*, как правило, отводится два байта памяти, т. е. 16 разрядов. Эти 16 разрядов (бит) можно разбить на битовые поля, каждое из которых выделяется для определенной переменной типа *unsigned int*. Пример размещения битового поля в оперативной памяти показан на рис. 1.10.



Рис. 1.10

Здесь под переменные i и s выделено по два бита памяти, под переменные p и k по три бита, под переменную j шесть бит. При работе с битовыми полями нужно внимательно следить за тем, чтобы значение переменной не потребовало памяти больше, чем под неё выделено.

Пример. Разработать программу, осуществляющую упаковку целых переменных i, j, k, p, s в битовые поля и вводящую на печать содержимое памяти, отведенной под эти переменные.

Программа

```
#include <stdio.h>
struct field type
   unsigned i: 2;
  unsigned j: 6;
  unsigned k: 3;
  unsigned p: 2;
  unsigned s: 2;
} my field;
main()
{
  my field.i = 3;
  my field.j = 20;
  my field.k = 2;
  my_field.p = 1;
  my field.s = 0;
  printf("\n my_field = %u \n", my_field);
```

В результате выполнения программы в ячейке памяти будет размещен код, показанный на рис. 1.11.



Рис. 1.11. Содержимое участка памяти, отведенного под структуру *struct field type*

На экран будет выведено следующее число: $1 \cdot 2^{11} + 1 \cdot 2^{9} + 1 \cdot 2^{6} + 1 \cdot 2^{4} + 1 \cdot 2^{1} + 1 \cdot 2^{0} = 2643$.

§1.26. Указатели и структуры

Выбор элементов структуры или объединения. Выражение выбора элемента позволяет получить доступ к элементу структуры или объединения. Выражение имеет значение и тип выбранного элемента.

Возможны три способа доступа к полям [4]:

- 1) <выражение>.<идентификатор>, где <выражение> значение типа struct или union; <идентификатор> поле структуры или объединения;
- 2) <выражение> -> <идентификатор>, где <выражение> указатель на структуру или объединение; <идентификатор> поле структуры или объединения;
- 3) (<*выражение>).<идентификатор>, где <выражение> указатель на структуру или объединение; <идентификатор> поле структуры или объединения.

Работа со структурами с помощью указателей. Доступ к элементам структуры или объединения можно осуществить с помощью указателей. Для этого необходимо инициализировать указатель адресом структуры или объединения [14].

Пример

```
/* Объявление структуры типа book.*/
struct book
{
   char title [15];
   char author [15];
   float value;
};
/* Объявление массива структур типа book, */
/* состоящего из 100 элементов. */
/* Имя массива libry. */
struct book libry [100];
/* Объявление указателя на структуру типа book.*/
struct book *p;
/* Инициализация указателя р адресом*/
/* 0-го элемента массива структур типа book.*/
p=&libry[0];
```

Для организации работы с массивом можно использовать указатель p или имя массива:

- 1) (*p).value, что равнозначно записи libry[0].value;
- 2) $p \rightarrow value$, что равнозначно записи *libry*[0].*value*.

Инициализация структур без указателя

```
gets(libry[i].title); /* Ввод названия книги.*/
gets(libry[i].author); /* Ввод имени автора.*/
scanf("%f", &libry[i].value); /* Ввод цены книги.*/
libry[i].value=123; /* Ввод цены книги.*/
```

Инициализация массива структур при помощи указателя р

```
gets((*p).title); /* Ввод названия книги.*/
gets((*p).author); /* Ввод имени автора.*/
(*p).value=123; /* Ввод цены книги.*/
gets(p -> title); /* Ввод названия книги.*/
gets( p -> author); /* Ввод имени автора.*/
p -> value=150; /* Ввод цены книги.*/
```

Указатель p содержит адрес начала массива — адрес нулевого элемента массива структур типа book. Используя p, можно осуществить доступ к каждому полю структуры типа book. Чтобы осуществить доступ к другим структурам, необходимо изменить адрес, хранящийся в p. Сдвиг указателя на i элементов массива осуществляется следующим образом:

$$p = p + i$$
;

Графическое представление массива структур в памяти ЭВМ показано на рис. 1.12.

Динамическое распределение памяти под массив структур. Динамически распределять память под массив структур необходимо в том случае, когда заранее не известен размер массива.

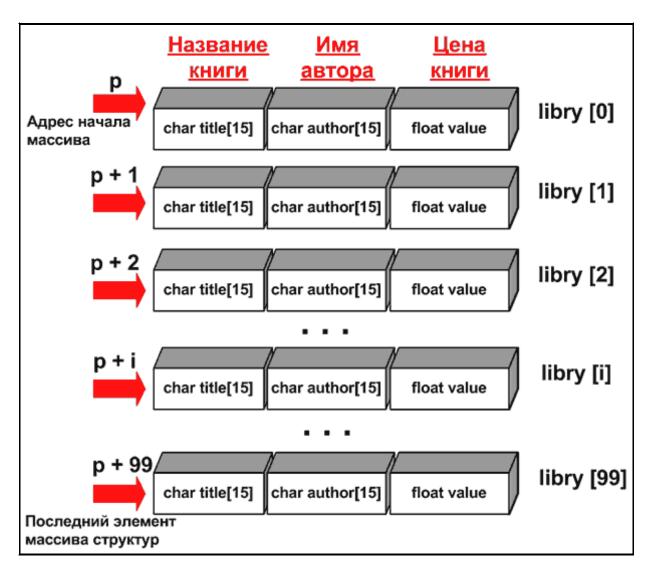


Рис. 1.12. Массив структур

Пример

```
scanf("%u", &n);

p=(struct book *)malloc(n*sizeof( struct book));

/*Переменная р содержит адрес */
/*нулевой структуры типа book.*/
```

§1.27. Классификация функций ввода-вывода

Функции ввода-вывода позволяют:

- 1) читать данные из файлов и устройств ввода информации;
- 2) записывать данные в файлы и выводить их на устройства вывода информации.

Существуют три класса функций ввода-вывода:

- **1. Функции ввода-вывода верхнего уровня.** Для данных функций характерен потоковый ввод-вывод информации.
- **2.** Функции ввода-вывода для консольного терминала и порта. Ввод и вывод осуществляется путем непосредственного обращения к устройствам персонального компьютера.
 - 3. Функции ввода-вывода нижнего уровня.

Для функций ввода-вывода верхнего уровня характерным является то, что обмен информации производится через буфер. *Буфер* — это некоторая область оперативной памяти, в которую подгружаются данные и откуда считываются данные.

При считывании из файла информация сначала загружается в буфер, а затем записывается в переменные и массивы, определенные в программе (рис. 1.13).

Буферизация используется для того, чтобы снизить количество обращений к медленно работающим внешним устройствам.

Для отражения особенностей организации ввода-вывода информации вводится специфическое понятие «поток».

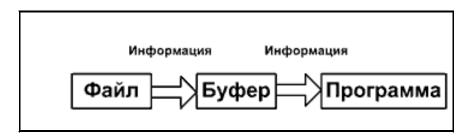


Рис. 1.13. Схема передачи информации функций ввода-вывода верхнего уровня

Поток — это абстрактное понятие, относящееся к любому переносу данных от источника данных к получателю данных. Ввод информации от источника называется приемом данных. Вывод информации на приемник называется втавкой или помещением данных.

Потоков в языке Си пять. Когда программа будет начинать выполняться, автоматически открываются следующие потоки:

- стандартный ввод (stdin);
- стандартный вывод (*stdout*);
- вывод для сообщений об ошибках (stderr);
- стандартное устройство печати (stdprn);
- стандартный последовательный порт (stdaux).

Функции ввода-вывода для консольного терминала и порта используют специфические особенности ПК — наличие видеоадаптора.

Для функций ввода-вывода нижнего уровня характерно следующее:

- 1) используются средства ввода-вывода операционной системы;
- 2) форматирование данных не осуществляется;
- 3) отсутствует буферизации данных.

§1.28. Функции ввода-вывода высокого уровня

В языке Си ввод и вывод информации возложен на функции ввода-вывода. Прототипы функций ввода-вывода высокого уровня содержатся в файле stdio.h.

До сих пор мы использовали только две функции из стандартной библиотеки функций ввода-вывода:

- 1) функцию *printf* функцию форматированного вывода;
- 2) функцию *scanf* функцию форматированного ввода.

Функция printf

```
#include <stdio.h>
int printf(format_string [, argument...]);
char *format_string;
```

Функция возвращает число напечатанных символов. Информация выводится в стандартный поток *stdout*.

Пример

```
printf ("\n %s %s %s\n", "Я", "учусь", "в ЮУрГУ");

char *a="Я учусь в ЮУрГУ";

printf(a);
```

Пример

```
#include <stdio.h>

main()
{
    char name1[40], name2[11];
    int count;

    printf("Введите два имени\n");
    count = scanf("%s%4s", name1, name2);
    printf("\n Введено %d имени %s и %s \n",
    count, name1, name2);
}
```

Результат работы

```
Введите два имени
Caшa <Enter>
Александр <Enter>
Введено 2 имени Саша и Алек
```

Функция scanf()

```
#include <stdio.h>
int scanf(format_string [, argument...]);
char *format_string;
```

Функции *scanf* работает со строками через формат %s. Данные читаются из стандартного потока *stdin*; чтение строки символов производится до тех пор, пока не встретится пустой символ «», т. е. с помощью *scanf* нельзя ввести строку с пробелом. С помощью функции *scanf* можно читать указанное число символов (например %10s — мы вводим 10 символов). Функция возвращает целое число, равное количеству введеных символов, если ввод прошел успешно. Функция возвращает значение EOF (end of file), если была попытка прочитать конец файла.

Продолжаем рассматривать функции ввода и вывода из стандартной библиотеки Си.

Рассмотрим функции, позволяющие осуществлять текстовый ввод и вывод. Текстовый ввод-вывод имеет дело с потоком литер или с текстовым потоком.

Текстовый поток — это последовательность литер, разбитая на строки, каждая из которых состоит из нуля или более литер (или не содержит символов) и завершается литерой «новая строка». Поэтому для работы с текстовым потоком достаточно иметь лишь функции ввода или вывода одной литеры. В библиотеке stdio.h эти функции носят имена getchar и putchar.

Функция getchar читает символ из входного потока stdin.

```
int getchar();
```

За одно обращение к функции *getchar* читается одна литера ввода из текстового потока, код которой и выдается в качестве результата.

Пример

```
c=getchar();
```

После выполнения операции присвоения переменная c будет содержать код введенной литеры.

Функция *putchar* записывает символ с кодом c в выходной поток *stdout*. Обращение к *putchar* приводит к печати одной литеры. Например, *putchar*(c) выводит на экран одиночный символ с кодом c.

```
int putchar(c);
int c;
```

Пример. Разработать программу, копирующую текстовый файл с клавиатуры на экран. Схема программы приведена на рис. 1.14.

Программа

```
#include <stdio.h>

main()
{
   int a;
   a=getchar();

   while(a!=EOF)
   {
      putchar(a);
      a=getchar();
   }
}
```

Работа программы

а	<enter></enter>
a	<enter></enter>
абв	<enter></enter>
абв	<enter></enter>
1	<enter></enter>
1	<enter></enter>
^ Z	<ctrl-z></ctrl-z>

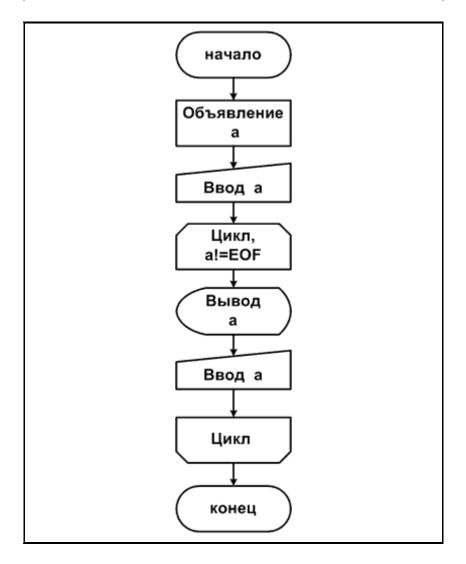


Рис. 1.14

При нажатии клавиш 'a' и <Enter> во входном потоке будут находиться символы 'a' и '\n', поэтому программа выведет 'a' и переведёт курсор на следующую строку. Если оператор нажмет <Ctrl-z>, то программа закончит работу, поскольку <Ctrl-z> трактуется как конец файла. Признаком конца файла является константа EOF (end of file), определенная в файле stdio.h. Константа EOF содержит код, возвращаемый как признак конца файла. (В файле stdio.h имеется строка #define EOF -1 .)

В рассматриваемом примере переменная *а* имеет тип *int*, так как функция *getchar* возвращает код символа типа *int*. При написании программы появилась проблема: как, используя функцию *getchar*, отличить конец ввода от обычных читаемых данных? Было найдено следующее решение: необходимо, чтобы функция *getchar* по исчерпании входного потока читала такое значение, которое нельзя было спутать ни с одной реальной литерой. Это значение и есть *EOF*.

Символ *<Cntrl-z>* (шестнадцатеричный код 1A) рассматривается как индикатор конца файла. Он автоматически помещается текстовым редактором при создании файла в его конец.

Функция *gets* читает строку из стандартного входного потока *stdin* и помещает её по адресу, задаваемому параметром a.

```
#include <stdio.h>
char *gets(a);
char *a;
```

В строку включается все символы до первого встретившегося символа новой строки '\n', не включая его. Строка заканчивается нулевым символом '\0'. Возвращаемое значение: значение параметра (адрес хранящийся в a) — если строка считана удачно; NULL — если произошла ошибка или достигнут конец файла.

Функция puts

```
#include <stdio.h>
int puts(a);
char *a;
```

Функция *puts* записывает строку, адрес которой определяется значением параметра a, в стандартный выводной поток *stdout*, добавляет в завершение символ новой строки '\n'. Выводятся все символы из строки, пока не встретися завершающий строку нулевой символ '\0'.

Возвращаемые значения: '\n' — символ новой строки, если ввод прошел удачно; EOF — если произошла ошибка.

§1.29. Работа с файлами данных

Для программиста открытый файл представляется как последовательность данных — считываемых данных или записываемых данных. При открытии файла с ним связывается поток. Выводимая информация записывается в поток; считываемая информация берется из потока. Когда поток открывается для вводавывода, он связывается со структурой типа *FILE*, которая определена в stdio.h. Структура содержит разнообразную информацию о файле.

При открытии файла с помощью функции fopen возвращается указатель на структуру типа FILE, который можно использовать для последующих операций с файлом.

Функция *fopen* открывает файл, имя которого задается аргументом *name*, и связывает с ним поток для выполнения операций ввода-вывода данных. Параметр *type* — это указатель на строку символов, определяющих способ доступа к файлу. Возвращаемое значение функции — указатель на структуру типа *FILE*.

Значения аргумента *type* следующие:

- "r" открыть файл для чтения (файл должен существовать);
- "w" открыть пустой файл для записи; если файл существует, то его содержимое теряется;
- "а" открыть файл для записи в конец (для добавления); файл создается, если он не существует;
 - "r+" открыть файл для чтения и записи (файл должен существовать);
- "w+" открыть пустой файл для чтения и записи; если файл существует, то его содержимое теряется;
- "а+" открыть файл для чтения и дополнения, если файл не существует, то он создаётся.

Возвращаемое значение функции fopen:

- 1) указатель на открытый поток;
- 2) значение *NULL*, если обнаружена ошибка.

Пример. Открытие существующего файла для чтения

```
main()
{
   FILE *f; /* f - указатель на открытый поток */
   char *a, *b; /* a и b - указатели на строки */
   a= "a.dat";
   b="r";
```

```
f=fopen(a, b);

/* или более простой вариант: */

/* f= fopen("a.dat", "r"); */
```

Пример

Функция *fclose* закрывает поток или потоки, связанные с открытыми при помощи функции *fopen* файлами. Закрываемый поток определяется аргументом f.

```
#include <stdio.h>
int fclose(f);
FILE *f;
```

Возвращаемое значение:

- 1) значение 0, если поток успешно закрыт;
- 2) ЕОГ, если произошла ошибка.

Пример

```
fclose(f);
```

Функция *feof* определяет достигнут ли конец файла. Если конец достигнут, то операция чтения будет возвращать значение EOF до тех пор, пока поток не будет закрыт.

```
#include <stdio.h>
int feof(f);
FILE *f;
```

Возвращаемое значение:

- 1) 0 если конец файла не найден;
- 2) ненулевое значение, если достигнут конец файла.

Функции ввода-вывода, работающие с файлами

1. Функция чтения символа из файла fgetc. Функция fgetc читает один символ из вводного потока f.

```
#include <stdio.h>
int fgetc(f);
FILE *f;
```

Возвращаемое значение:

- 1) код прочитанного символа;
- 2) ЕОГ, если конец файла или ошибка.
- **2. Функция записи символа в файл fputc.** Функция **fputc** записывает одиночный символ, имеющий код c, в поток f. Это аналог функции **putchar** для работы с файлами.

```
#include <stdio.h>
int fputs(c,f);
int c;
FILE *f;
```

Возвращаемое значение:

- 1) код записанного символа;
- 2) ЕОГ, если встретился конец файла или произошла ошибка.

Пример. Посимвольное копирование файла

```
fclose(out);
fclose(in);
}
else
printf("Не могу открыть
out.dat \n");
}
else
printf("Не могу открыть in.dat \n");
}
```

Функции *fscanf* и *fprintf*. Действия данных функций похоже на действия функций *scanf* и *printf*, однако эти функции работают с файлами данных, и первый аргумент функций — это указатель на файл.

3. Функция fscanf читает данные из указанного потока f, выполняя форматные преобразования, и полученные значения записывает в переменные, адреса которых задаются параметрами **argument**.

```
#include <stdio.h>
int fscanf(f, format_string [,argument...]);
FILE *f;
char *format_string;
```

4. Функция fprintf выполняет формальные преобразования данных и печатает последовательность символов и значений в выводной поток f.

```
#include <stdio.h>
int fprintf(f, format_string[, argument...]);
FILE *f;
char *format_string;
```

Пример. Дано два файла S.dat и S1.dat. Необходимо считать число типа *int* из файла S.dat и записать его в файл S1.dat.

```
main()
{
    FILE *S, *S1;
    /* S — указатель на файл S.dat */
    /* S1 — указатель на файл S1.dat*/
    int age;

S=fopen("S.dat", "r");
    S1=fopen("S1.dat", "a");
```

```
fscanf(S, "%d", &age);

fclose(S);

fprintf(S1,"Пете — %d лет \n", age);

fclose(S1);
}
```

Функции *fgets* и *fputs* предназначены для ввода-вывода строк, они являются аналогами функций *gets* и *puts* для работы с файлами.

5. Функция чтения строки символов из файла fgets. Функция fgets читает строку из входного потока f и помещает её в строку, адрес которой задается значением параметра s. Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки '\n', который включается в строку, или пока не наступит конец потока или не будет прочитано n-1 символов. Результат помещается в s и заканчивается нулевым символом '\0'. Функция возвращает адрес строки.

```
#include <stdio.h>
char *fgets(s, n, f);
char *s;
int n;
FILE *f;
/* f - указатель на файл */
```

6. Функция записи строки символов в файл fputs. Функция fputs копирует строку s в поток f с текущей позиции. Завершающий нулевой символ не копируется.

```
#include <stdio.h>
int fputs(s, f);
char *s;
FILE *f;
```

§1.30. Функции обработки строк

Основные функции стандартной библиотеки string.h [4]:

• char *strcat(s, ct) — присоединяет ct к s; возвращает s;

- char *strncat(s, ct, n) присоединяет не более n литер ct к s, завершая s литерой '\0'; возвращает s;
- char *strchr(cs, c) отыскивает в строке cs первый символ, код которого равен значению c, и возвращает значение указателя на данный символ; если символа не оказалось, возвращает значение NULL;
- char *strchr(cs, c) отыскивает в строке cs последний символ, код которого равен значению c, и возвращает значение указателя на даннный символ; если символа не оказалось, возвращает значение NULL;
- char *strcpy(s, ct) копирует строку ct в строку s, включая '\0'; возвращает s;
- *char* **strncpy*(s, ct, n) копирует не более n литер строки ct в строку s; возвращает s; дополняет результат литерами '\0', если литер в ct больше n;
- *int strcmp*(cs, ct) сравнивает cs с ct; возвращает значечние, меньшее нуля, если cs меньше ct; значение, равное нулю, если строка cs эквивалентна строке ct; и значение, большее 0, если cs больше ct;
- *int strncmp*(cs, ct, n) сравнивает не более n литер cs и ct; возвращает значение, меньшее 0, если cs меньше ct; 0, если строка cs эквивалентна строке ct; и значение, большее 0, если cs больше ct;
- $int\ strlen(s)$ выдает число символов в строке s без учета нулевого символа конца строки;
- char *strlwr(s) переводит всю строку s в нижний регистр (в строчные буквы);
- char *strset(s, c) заполняет всю строку s в верхний регистр (в прописные буквы);
 - char *strdup(s) вызывает функцию malloc и отводит место под копию s;
- char *strset(s, c) заполняет всю строку символами, код которых равен значению c;
- *char* **strnset*(s, c, n) заменяет первые n символов строки s на символы, код которых равен значению c;
- char *strpbrk(s, t) просматривает строку s до тех пор, пока не встретится символ, содержащийся в t;
- $int\ strspn(s,\ t)$ возвращает длину начального сегмента строки s, который состоит исключительно из символов, содержащихся в строке t.

§1.31. Работа со строками

В программе строки могут определяться следующим образом [14]:

- 1) как строковые константы;
- 2) как массивы символов;
- 3) через указатель на символьный тип;
- 4) как массивы строк.

Программист должен понимать, что строка — это массив символов, и под неё необходимо выделять участок оперативной памяти.

Строковая константа

Любая последовательность символов, заключенная в двойные кавычки, рассматривается как строковая константа. Последовательность символов будет размещена в оперативной памяти ЭВМ, включая нулевой байт. Под строку выделяются последовательно идущие ячейки оперативной памяти. Для кодировки одного символа достаточно одного байта.

Если необходимо включить в строку символ двойных кавычек, ему должен предшествовать символ обратной дробной черты:

```
"Повсюду стали слышны речи:\n"
```

"\"Пора добраться до картечи!\"\n"

"И вот на поле грозной сечи\n"

"Ночная пала тень.\n".

Строковые константы размещаются в *статической* памяти. Сама последовательность символов в двойных кавычких трактуется как адрес строки. Это аналогично использованию имени массива, служащего указателем на расположение массива.

Массив символов

При определении массива символьных строк необходимо сообщить компилятору требуемый размер памяти.

```
char m[82];
```

Компилятор также может самостоятельно определить размер массива символов.

Как и для других массивов, m2 и m3 являются указателями на первые элементы массивов.

```
m2 == &m2[0]
*m2 == '\Gamma'
*(m2 + 1) == 'o'
m3 == &m3[0]
*(m3 + 2) == 'x'
```

Для создания строки можно использовать указатель.

```
char *m4 = "Не пылит дорога, не дрожат листы...";
```

Это почти то же самое, что и

```
char\ m5[] = "Подожди немного, отдохнешь и ты."; ...
```

В обоих случаях *m*4 и *m*5 являются указателями на строки, и размеры массивов определяются компилятором. Однако между *m*4 и *m*5 есть и некоторая разница.

Массив или указатель. Во втором случае объявление массива вызывает создание в памяти массива из 33 элементов (по одному на каждый символ плюс один на завершающий символ '\0'). Каждый элемент инициализируется соответствующим символом. В дальнейшем компилятор будет рассматривать имя m5 как синоним адреса первого элемента массива, т. е. &m5[0]. Следует отметить, что m5 является константой-указателем. Нельзя изменить m5, так как это означало бы изменение положения (адреса) массива в памяти. Можно использовать операции, подобные m5+1, для идентификации следующего элемента массива, однако не разрешается выражение ++m5. Оператор увеличения можно использовать с именами переменных, но не констант [14].

В случае с указателем m4 в памяти создается 36 элементов для запоминания строки. Но, кроме того, выделяется еще одна ячейка памяти для nеременной m4, являющейся указателем. Сначала эта переменная указывает на начало строки, но её значение может изменяться. Поэтому допустимо использовать операцию приращения на единицу: ++m4 будет указывать на второй символ строки — 'e' [14].

Различия между массивами и указателями. Рассмотрим различия в использовании описаний следующих двух видов [14]:

```
char\ heart[] = "Подожди немного, "; и char\ *reason = "отдохнешь и ты."; .
```

Основное отличие состоит в том, что указатель *heart* является константой, в то время как указатель reason — переменной. Посмотрим, что на самом деле дает эта разница.

Во-первых, и в том и в другом случае можно использовать операцию сложения с указателем.

```
for (i = 0; i < 17; i++ )
    putchar( *(heart + i) );

putchar('\n')

for (i = 0; i < 15; i<sup>+ +</sup> )
    putchar( *(reason + i) );

putchar('\n');
```

Результат работы

```
Подожди немного, отдохнешь и ты.
```

В случае с указателем можно использовать операцию приращения на единицу.

```
while( *reason != '\0') /*Останов в конце строки.*/
{
   putchar( *(reason++) );
}
   /*Печать символа*/
   /*и перемещение указателя.*/
```

Результат работы

```
отдохнешь и ты.
```

Явное задание размера памяти. При объявлении массива можно указать: $char\ ml[36] =$ "В полдневный жар в долине Дагестана";

вместо

 $char\ ml[\] =$ "В полдневный жар в долине Дагестана"; .

Число элементов по крайней мере на один нулевой символ больше, чем длина строки. Как и в других статических или внешних массивах, любые неиспользованные элементы автоматически инициализируются нулем (который в символьном виде является нулевым байтом '\0', а не символом цифры нуль).

Массивы символьных строк

Обычно бывает удобно иметь массив символьных строк. В этом случае можно использовать индекс для доступа к нескольким разным строкам [14].

```
char *poet[4] = {"Погиб поэт!", " - невольник чести - ", "Пал,", " оклеветанный молвой..."};
```

Можно сказать, что *poet* является массивом, состоящим из четырех указателей на символьные строки. Каждая строка символов представляет собой символьный массив, потому имеется четыре указателя на массивы. Первым указателем является poet[0], и он ссылается на первую строку. Второй указатель poet[1] ссылается на вторую строку. Каждый указатель, в частности, ссылается на первый символ своей строки:

*
$$poet[0] == '\Pi',$$

* $poet[1] == ",$
* $poet[2] == '\Pi'$

и т.д.

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны скобочной записи

$$\{\{...\}, \{...\}, ..., \{...\}\};$$

где многоточия подразумевают строки. В первую очередь можно отметить, что первая последовательность, заключенная в двойные кавычки, соответствует первым парным скобкам и используется для инициализации первого указателя символьной строки. Следующая последовательность в двойных кавычках инициализирует второй указатель и так далее. Запятая разделяет соседние последовательности.

Кроме того, мы можно явно задавать размер строк символов, используя описание, подобное такому:

Разница заключается в том, что второй индекс задает «прямоугольный» массив, в котором все «ряды» (строки) имеют одинаковую длину. Описание

$$char *poet[4];$$

однако, определяет «рваный» массив, где длина каждого «ряда» определяется той строкой, которая этот «ряд» инициализировала. Рваный массив не тратит память напрасно (рис. 1.15).

<i>«РВАНЫЙ» МАССИВ</i> Погиб поэт!\0																						
_		н		В	0	Л	ь	Н	И	к		띡	е	С	Т	N		-	\0			
П	а	Л	,	\0																		
0	к	Л	е	В	е	Т	а	Н	Н	Ы	Й		М	0	Л	В	0	й				\0
ПРЯМОУГОЛЬНЫЙ МАССИВ																						
П	0	г	И	б		П	0	Э	Т	!	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
_		Н	е	В	0	Л	Ь	Н	И	к		Ч	Ф	С	Т	И		-	\0	\0	\0	\0
П	а	Л	,	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
0	к	Л	е	В	е	Т	а	Н	Н	ы	й		М	0	Л	В	0	й				\0

Рис. 1.15

Указатели и строки

Большинство операций языка Си, имеющих дело со строками, работает с указателями. Для размещения в оперативной памяти строки символов необходимо предпринять следующие действия:

- 1) выделить блок оперативной памяти под массив;
- 2) осуществить ввод строки по адресу массива, используя специальную функцию ввода информации.

Пример

```
char *name;
scanf("%s", name);
```

Данная программа содержит ошибку, поскольку ЭВМ запишет строку по неизвестному адресу, так как *name не инициализирован. Возможно «зависание» компьютера.

Пример

```
char *name;
name = (char*)malloc(10);
scanf("%s", name);
```

Данная программа корректна. Однако, выделенной памяти может не хватить. Тогда часть символов будет записана в область, не предназначенную для этого. Доработанный вариант программы, считывающей только 9 символов, приведен ниже.

```
char *name;
name = (char*)malloc(10);
scanf("%9s", name);
```

Как только выделена память для массива, можно считывать строку. Для ввода часто используют функции *scanf* и *gets*.

Функция *gets* получает строку от стандартного устройства ввода системы. Функция читает символы до тех пор, пока не встретится символ новой строки \n' , который создается при нажатии клавиши <Enter>. Функция берет все символы до (но не включая) символа новой строки, присоединяет к ним нулевой символ \n' 0' и передает строку вызывающей программе [4].

```
/*Получение имени №1.*/
main()
{
    char name[81]; /*Выделение памяти под строку.*/
    printf("Привет, как вас зовут?\п");
    gets (name);
    /*Размещение имени по адресу name.*/

    printf("Хорошее имя, %s.\n", name);
}
```

Функция примет любое имя (включая пробелы) длиной до 80 символов.

```
/*Получение имени №2.*/
main()
{
    char name[80];
    char *ptr;

    printf("Привет, как вас зовут?\n");
    ptr = gets(name);
    printf("%s? Ax! %s!\n", name, ptr);
}
```

Получился диалог

```
Привет, как вас зовут?
Владимир <Enter>
Владимир? Ax! Владимир!
```

Функция *scanf*. Основное различие между *scanf* и *gets* заключается в том, как они определяют, что достигли конца строки; *scanf* предназначена скорее для получения слова, а не строки. Функция *gets* принимает все символы до тех пор, пока не встретит первый символ новой строки.

Существует два варианта использования функции scanf [4]:

- 1. Если применять формат %s, строка вводится до (но не включая) следующего пустого символа (пробел, табуляция или новая строка).
- 2. Если определять размер строки как %10s, то функция scanf считает не более 10 символов или же считает до любого пришедшего первого пустого символа.

Функция scanf возвращает целое значение, равное числу считанных символов, если ввод прошел успешно, или символ EOF, если была попытка прочитать конец файла.

Программа

```
/* Использование функции scanf().*/
main()
{
    char name1[40], name2[11];
    int count;

    printf("Введите, пожалуйста, 2 имени:\n");
    count = scanf("%s%6s", name1, name2);

    printf("Я считал %d имени %s и %s.\n",
    count, name1, name2);
}
```

Два примера работы программы

```
Введите, пожалуйста, два имени:
Наталья Анна <Enter>
Я считал 2 имени Наталья и Анна.
Введите, пожалуйста, 2 имени.
Наталья Кристина <Enter>
Я считал 2 имени Наталья и Кристи.
```

Во втором примере были считаны только первые 6 символов от *Кристины*, так как использовался формат %6s.

Если с клавиатуры вводится только текст, лучше применять функцию *gets*. Она проще в использовании, быстрее и более компактна. Функция *scanf* предназначена в основном для ввода смеси типов данных в некоторой стандартной форме. Например, если каждая вводимая строка содержит наименование инструмента, количество его на складе и стоимость каждого инструмента, можно использовать функцию *scanf*.

В следующем примере размер массива символов запрашивается у оператора.

Пример

```
int n;
char *p;
```

```
printf("\nСколько букв в Вашем имени?");
scanf ("%u", &n);
p=(char*)malloc( n+1 );
printf("\nВведите Ваше имя: ");
scanf("%s", p);
```

Вывод строк

Для вывода строк наиболее часто используют функции puts и printf.

Функция *puts* выводит на экран строку с заданным адресом, добавляя в завершение символ новой строки. Выводятся все символы из строки, пока не встретится завершающий нуль-символ. У функции только один аргумент, являющийся указателем строки.

Программма

```
#include <stdio.h>
#define DEF "Я строка #define."
main()
{
    char strl[]="Массив инициализирован мной.";
    char *str2="Указатель инициализирован мной.";
    puts("Я аргумент функции puts.");
    puts(DEF);
    puts(strl);
    puts(str2);
    puts(&str1[4]);
    puts(str2 + 4);
}
```

Результат работы программы

```
Я аргумент функции puts.
Я строка #define.
Массив инициализирован мной.
Указатель инициализирован мной.
ив инициализирован мной,
атель инициализирован мной.
```

Функция *printf*. Мы рассмотрели подробно данную функцию. Подобно *puts*, она использует указатель строки в качестве аргумента. Функция *printf* менее удобна, чем *puts*, но более гибкая.

Разница между *puts* и *printf* заключается в том, что *printf* не переводит автоматически курсор на новую строку. Так,

 $printf("%s\n", string);$

дает то же самое, что и

puts(string); .

Первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, *printf* позволяет легко объединять строки для печати их в одной строке. Например:

printf("Хорошо, %s, %s.\n", *name*, MSG);

объединяет слово "Хорошо" с именем name и с символьной строкой MSG в одну строку.

Примеры обработки строк

Программа «обрезания» строки [14]

```
#include <stdio.h>
#include <string.h>
main()
{
   char message[]="Строка символов";
   puts (message);
   delmessage(message, 10);
                     /*вызов функции delmessage*/
   puts (message);
}
/*delmessage() — функция укорачивает строку*/
/*message, оставляя лишь первые size символов.*/
void delmessage(str, size)
char *str;
int size;
   if (strlen(str)>size)
        *(str+size)='\0';
```

Результат работы программы

```
Строка символов
Строка сим
|<---->|
10 символов
```

Содержимое памяти ЭВМ

```
Строка сим'\0'олов'\0'
```

Программа объединения двух строк [14]

Результат работы программы

```
Назовите ваш любимый цветок
Пион <Enter>
Пионы хорошо пахнут
```

Замечание: массив flower должен содержать достаточно места, чтобы поместить addition.

Программа сравнения двух строк [14]

```
#include <stdio.h>
#include <string.h>
#define reply "Грант"
```

```
main()
{
    char name[40];

    puts("Кто похоронен в могиле Гранта?");

    gets(name);

    while(strcmp(name, reply)!=0)
    {
        puts("Неверный ответ.\n");
        gets(name);
    }

    puts("Правильно!");
}
```

Программа копирования строк [14]

```
#include < stdio.h)
#define WORDS "Проверьте, пожалуйста, вашу последнюю
запись."

main()
{
    static char *orig = WORDS;
    static char copy[40];

    puts(orig);
    puts(copy);

    strcpy(copy, orig);

    puts(orig);
    puts(orig);
    puts(orig);
    puts(copy);
}</pre>
```

Результат работы программы

```
Проверьте, пожалуйста, вашу последнюю запись. Проверьте, пожалуйста, вашу последнюю запись. Проверьте, пожалуйста, вашу последнюю запись.
```

§1.32. Логический тип данных в языке Си

Логическая переменная — переменная, которая принимает только два значения:

- 1) истина (1 или *TRUE*);
- 2) ложь (0 или *FALSE*).

На первый взгляд логического типа данных в языке Си нет. К этому выводу можно прийти, если рассматривать типы данных:

- 1) *char* символьный;
- 2) *int* целый;
- 3) *short* короткий целый;
- 4) *long* длинный целый;
- 5) *float* число с плавающей точкой;
- 6) double число с плавающей точкой двойной точности.

Логического типа данных мы здесь не видим. Посмотрим, как обстоит дело с этим типом данных в других языках программирования.

Visual Basic — язык разработки приложений под Windows:

- 1) *integer* целое;
- 2) *long integer* длинное целое;
- 3) single число с плавающей точкой;
- 4) double число с плавающей точкой двойной точности;
- 5) *currency* денежный тип;
- 6) *byte* целое положительное число (0... 255);
- 7) boolean <u>логический тип данных</u> (значение TRUE или FALSE).

Типы данных языка Фортран (Fortran):

- 1) *INTEGER* целое;
- 2) REAL число сплавающей точкой;
- 3) REAL*8 число с плавающей точкой двойной точности;
- 4) СОМРЬЕХ комплексное число;
- 5) СОМРLЕХ*16 комплексное число двойной точности;
- 6) LOGICAL <u>логический тип данных</u>.

На мысль о том, что логический тип данных в Си все же есть, может натолкнуть наличие логические операции в языке. Логические операции не могут существовать без логических переменных.

Логические операции:

- 1) логическая функция «И» «&&»;
- 2) логическая функция «ИЛИ» «||»;
- 3) логическая функция «НЕ» «!».

Операнды логических операций могут иметь:

- 1) целый тип;
- 2) вещественный тип;
- 3) тип указатель.

Результат логических операций:

1) ложь (0) — типа *int*;

2) истина (1) — типа *int*;

Таким образом, логический тип данных в языке Си существует и имеет mun int.

Логическая операция «И» вырабатывает значение 1, если оба оперенда имеют ненулевое значение (табл. 1.3).

Таблица 1.3

Логическое «И» (&&) Y=X1&&X2;

X1 = <ненулевое значение>	X2 = 0	Y = 0
X1 = 0	X2=<ненулевое значение>	Y = 0
X1 = 0	X2 = 0	Y = 0
X1 = <ненулевое значение $>$	X2 = <ненулевое значение>	Y=1

Логическая операция «ИЛИ» вырабатывает значение 1 в том случае, если какой-либо из операндов имеет ненулевое значение (табл. 1.4).

Таблица 1.4

Логическое «ИЛИ» Y=X1||X2;

X1 = <ненулевое значениие $>$	X2 = 0	Y=1
X1 = 0	X2 = <ненулевое значениие>	Y=1
X1 = <ненулевое значениие $>$	X2 = <ненулевое значениие>	Y=1
X1 = 0	X2 = 0	Y = 0

Логическая операция «НЕ» осуществляет логическое отрицание значения операнда (табл. 1.32.3).

Таблица 1.6

Логическое «HE» Y=!X;

X = <ненулевое значение $>$	Y = 0
X = 0	Y=1

Программа

```
#include <stdio.h>
main()
{
   printf("!2=%u", !2);
}
```

Результат работы программы

```
! 2=0
```

Логические операции «И», «ИЛИ» и «НЕ» позволяют создавать функции алгебры логики

$$Y = f(X1, X2, ..., Xn),$$

в которой X1, X2,, Xn — логические переменные, принимающие только два значения — «истина» и «ложь».

Пример. Представим, что нам необходимо сконструировать устройство для запуска двигателя лифта. Для этого необходимо написать логическую функцию

$$Y = f(X1, X2, X3, X4),$$

которая бы определяла работу двигателя.

Переменные логической функции в нашем случае будут следующие:

- 1) *X*1 вызов лифта извне;
- 2) *X*2 вызов из кабины лифта;
- 3) ХЗ датчик закрытия дверей;
- 4) X4 датчик перегрузки (если кабина лифта перегружена, то он стоит на месте).

Двигатель включиться в том случае (Y=1), если произошел вызов извне или из кабины лифта, дверь закрыта и нет перегрузки. Тогда логическая функция устройства управления запуском лифта будет следующая:

$$Y = (X1 \&\&X3 \&\&(!X4)) || (X2 \&\&X3 \&\&(!X4));$$
.

§1.33. Программная реализация стека

Стинения стеком называют область памяти ЭВМ, предназначенную для сохранения данных и организованную по принципу «первым вошел — последним вышел» [11]. Графическое представление стека показано на рис. 1.16.

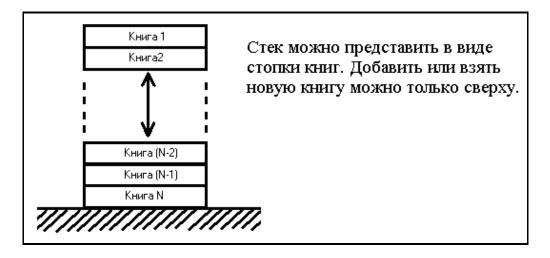


Рис. 1.16

В языке Си стек можно представить в виде следующей структуры:

```
struct stack
{
  int info;
  struct stack *next;
};.
```

Эта структура содержит переменную целого типа *info*, значение которой равно значению элемента, помещаемого в стек, и указатель на структуру типа *stack* под именем *next*, значение которого равно *текущему адресу начала стековой области памяти*.

В отношении стека определены три операции [11]:

- 1) push добавляет в стек новый элемент;
- 2) рор удаляет из стека элемент, помещенный туда последним;
- 3) peek взять элемент с вершины стека, не изменяя при этом весь стек.

Рассмотрим программу, в которой реализованы базовые операции работы со стеком *push*, *peek*, *pop*.

```
#include <stdio.h>
#include <alloc.h>
#define STACK struct stack

STACK
{
   int info;
   STACK *next;
};
void push(STACK **s,int item)
{
   STACK *new_item;
   new_item=( STACK *) malloc( sizeof(STACK) );
```

```
new item->info=item;
   new item->next=*s;
   *s=new item;
int pop(STACK **s,int *error)
 /*если операция рор выполнена успешно, */
 /* то error=0, в противном случае error=1*/
   STACK *old item=*s;
   int old info=0;
   if(*s)
   {
      old info=old item->info;
      *s=(*s) \rightarrow next;
      free(old item);
      *error=0;
    else *error=1;
    return (old info);}
int peek(STACK **s,int *error)
 /*если в стеке не менее одного элемента, */
 /* то error=0; если в стеке нет элементов, */
 /* To error=1*/
   if(*s)
      *error=0;
      return (*s) ->info;
   }
   else
      *error=1;
      return 0;
   }
}
main()
   STACK *st1=NULL;
   int error;
   push(&st1, 1200);
   printf("\nУказатель стека после ввода
   числа 1200 %u", st1);
   push(&st1, 13);
   printf("\nУказатель стека после ввода
```

```
числа 13 %u", st1);
push(&st1, 125);
printf("\пУказатель стека после ввода
числа 125 %u", st1);
printf("\n peek(st1)=%d", peek(&st1,&error));
printf("\n pop(st1)=%d", pop(&st1,&error));
printf("\n pop(st1)=%d", pop(&st1,&error));
printf("\n pop(st1)=%d", pop(&st1,&error));
printf("\n pop(st1)=%d", pop(&st1,&error));
printf("\n peek(st1)=%d", pop(&st1,&error));
}
```

Функция *push* в качестве результата своей работы возвращает указатель на новый элемент стека (изменяется значение указателя стека st1). Аргументами функции *push* являются: текущий адрес стека (значение указателя стека) st1 и значение элемента, размещаемого по этому адресу.

В функции push переменная *new_item* — указатель на новый элемент стека (рис. 1.17). Значение этого указателя определяется с использованием функции динамического распределения памяти *malloc*. Переменной *info* (элементу структуры типа *stack*), размещенной по этому адресу, присваивается значение элемента, помещаемого в стек. Указателю *next* (элементу структуры *stack*) присваивается значение s. Указателю s присваивается значение указателя на стек new_item (см. рис. 1.17).

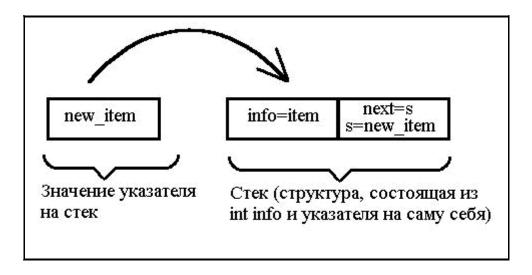


Рис. 1.17. Графическое представление работы программы

Анализ работы функции push приведен ниже. Шаг 1. С помощью строки

STACK *st1 = NULL;

объявляется указатель, который содержат адрес структуры типа stack. Указатель st1 инициализируется значением константы NULL (рис. 1.18).

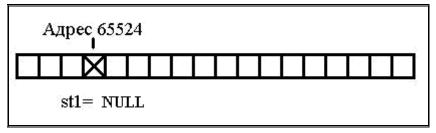


Рис. 1.18

Шаг 2. Вызывается функция

push(&st1, 1200).

В качестве аргумента используются: адрес указателя st1 и число 1200, которое помещается в стек.

Шаг 3. В теле функции *push* объявляется указатель *new_item* на структуру типа *stack*. Значение указателя *new_item* до выделения блока памяти составляет 6851 (рис. 1.19).

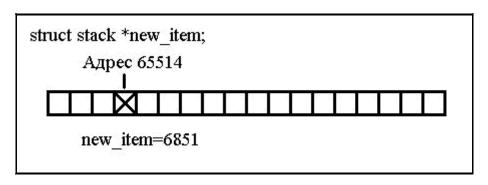


Рис. 1.19

Шаг 4. Строкой

new_item = (STACK *)malloc(sizeof(STACK));

выделяется блок памяти под структуру типа *stack*. Размер выделяемого блока памяти — 4 байта. Адрес блока памяти поместили в *new item* (рис. 1.20).

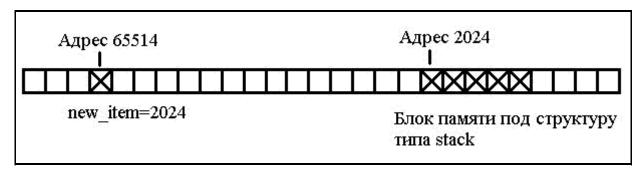


Рис. 1.20

Шаг 5. С помощью строки

$$new_item->info = item;$$

поместили в ячейку памяти *info* число 1200, так как *item* = 1200 (рис. 1.21).

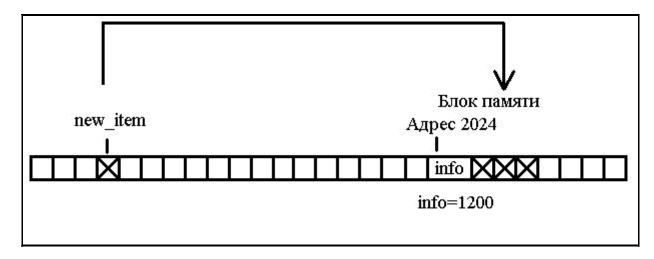


Рис. 1.21

Шаг 6. С помощью строки

$$new_item->next=*_S;$$

переменной *next* присваивается значение указателя st1, где st1 = NULL (рис. 1.22).

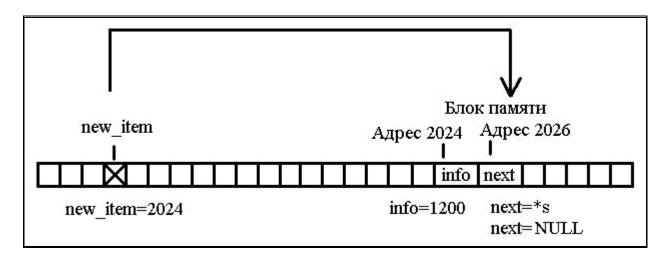


Рис. 1.22

Шаг 7. Строкой программы

$$*s = new_item;$$
 ,

где $new_item = 2024$, указателю st1 присваивается значение адреса стека (рис. 1.23).

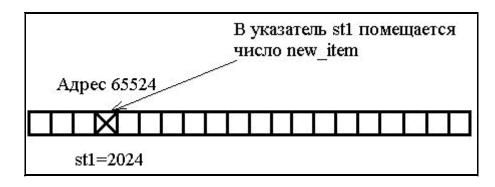


Рис. 1.23

Для иллюстрации работы программы построим таблицу (табл. 2.6).

Таблица 2.6

Изменение значений переменных в процессе работы програмы

Адрес st1	Старое значение <i>st</i> 1	Новое значение <i>st</i> 1	Адрес info	Адрес next	Значение info	Значение пехт
65524	NULL	2204	2204	2206	1200	NULL
65524	2204	2212	2212	2214	13	2204
65524	2212	2220	2220	2222	125	2212

Графическое представление работы программы представлено на рис. 1.24.

Функция *рор*. Параметрами функции *рор* являются: а) s — указатель на указатель на структуру типа stack, б) указатель еггог на переменную типа int. Функция возвращает значение типа int (значение переменной old_info). Объект old_item — указатель на старый элемент стека, которому присваивается значение указателя s.

Рассмотрим пошагово работу функции рор.

Шаг 1. Объявляется указатель на структуру типа stack (STACK *st1;).

Шаг 2. С помощью функции *push* в стек помещаются числа 1200, 13, 125.

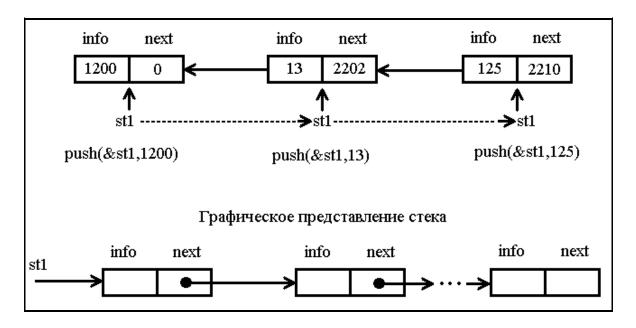


Рис. 1.24

Шаг 3. Из стека выталкиваются элементы. Вызывается функция

$$pop(\&st1, \&error)$$
.

В качестве аргументов функции используются: адрес указателя st1 и адрес переменной error.

Шаг 4. В теле функции *pop* объявляется указатель на структуру типа stack (имя указателя $old\ item$). Указателю $old\ item$ присваивается значение указателя s:

$$STACK * old_item = *s;$$
,

т. е. переменной *old_item* присваивается адрес верхнего элемента стека (рис. 2.25).

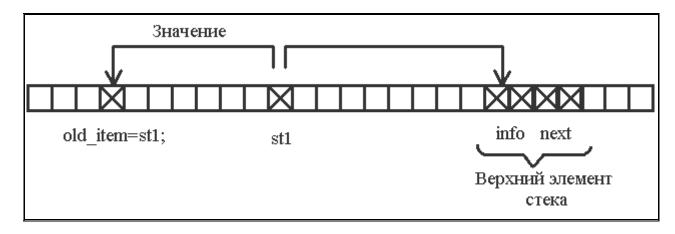


Рис. 1.25

Шаг 5. Если значение *s (т. е. st1) не равно 0, то выполняются следующие действия:

1. Строка

old
$$info = old item \rightarrow info$$
.

Переменной old info присваивается значение верхнего элемента стека.

2. Строка

$$*_S = (*_S) -> next.$$

Указатель st1 передвигается на следующий узел.

3. С помощью строки

освобождается блок памяти, на который указывает *old_item* (т.е. удаляется узел). На рис. 1.26 приведена иллюстрация работы функции *pop* на 5-м шаге.

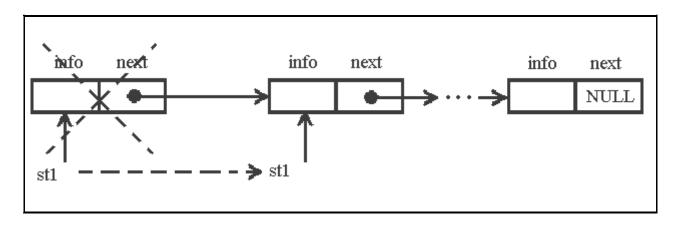


Рис. 1.26

- **Шаг 6.** Если значение указателя st1 равно 0, то переменной еггог присваивается значение 1. (Переменная еггог сигнализирует об ошибках при работе со стеком. Если error = 1, то произошла ошибка. Если error = 0, то ошибки нет.)
- **Шаг** 7. Функция возвращает значение переменной *old_info* (т. е. функция возвращает число, вытолкнутое из стека). Если операция *pop* выполнена успешно, то error = 0, в противном случае error = 1.

Функция *peek*. Параметрами данной функции являются: a) &st1 — адрес указателя st1 (т. е. адрес указателя, хранящего адрес стека), б) &error — адрес переменной error. Функция возвращает значение типа int.

Если значение указателя st1 не равно NULL, то переменной еггог присваивается значение 0, и функция возвращает значение верхнего элемента стека (значение переменной info).

Если значение указателя st1 равно NULL, то переменной еггог присваивается значение 1, и функция возвращает значение 0. (Если в стеке не менее одного элемента, то error = 0; если в стеке нет элементов, то error = 1.)

Результат работы программы

```
Указатель стека после ввода числа 1200 2204
Указатель стека после ввода числа 13 2212
Указатель стека после ввода числа 125 2220
рееk(st1)=125
рор(st1)=125
рор(st1)=13
рор(st1)=1200
рор(st1)=0
рееk(st1)=0
```

Резюме:

- 1. Каждый элемент стека содержит два поля: поле информации info и поле следующего адреса next.
 - 2. Поле адреса содержит указатель на следующий элемент стека.
- 3. Доступ к стеку осуществляется через указатель st1, который содержит адрес верхнего элемента стека.
- 4. Поле адреса последнего элемента должно содержать нулевое значение, как свидетельство того, что в стеке больше нет элементов.

Глава 2. Структуры данных

§2.1. Введение в структуры данных

Во второй главе мы рассмотрели *основные структуры данных* языка Си: переменные, массивы, структуры, объединения. Структуры данных называются основными, поскольку они часто используются в практике программирования, и из них можно образовывать более *сложные объекты*.

Переменные, массивы, структуры, объединения при объявлении получают имя и тип — под них выделяются ячейки оперативной памяти, в которые можно записывать некоторые значения. Таким образом, данные объекты имеют неизменяемую (статическую) структуру. Существует, однако, много задач, в которых требуются данные с более сложной структурой. Для них характерно, что в процессе вычислений изменяются не только значения объектов, но даже и сама структура хранения информации. Поэтому такие объекты стали называть динамическими информационными структурами. Их компоненты на некотором уровне детализации представляют собой объекты со статической структурой, т. е. они принадлежат к одному из основных типов данных. Эта глава посвящена проектированию объектов с динамической структурой, анализу и работе с ними [1, 3, 8, 11].

§2.2. Стек

Стеком называется упорядоченный набор элементов, в котором размещение новых и удаление существующих элементов происходит с одного конца, называемого *вершиной*. Иными словами, в стеке реализуется дисциплина обслуживания *LIFO* (last input — first output; первым вошел — последним вышел) [11].

Над стеком реализованы следующие операции [3, 8, 11]:

- 1) помещение элемента в стек push(s, i), где s стек, i помещаемый элемент;
 - 2) удаление элемента из стека i = pop(s);
- 3) определение верхнего элемента без его удаления i = stacktop(s), которая эквивалентна операциям i = pop(s) и push(s, i);
- 4) определение пустоты стека emty(s), которая возвращает *true*, если стек пустой и *false* в противном случае.

Существует несколько способов реализации стека:

- 1) с помощью одномерного массива конечного размера, распределенного статически;
 - 2) с помощью одномерного массива, распределенного динамически;
 - 3) в виде связанного списка.

Стек можно реализовать в виде следующей структуры:

```
#define NMAX 100
struct stack
{
  float elem[NMAX];
  integer top;
};
```

где NMAX — максимальное количество элементов в стеке; elem — массив из NMAX чисел типа float, предназначенный для хранения элементов стека; top — индекс элемента, находящегося на вершине стека.

Пример. Необходимо написать программу-калькулятор, вычисляющую значение выражения, записанного в постфиксной форме [7]. При записи выражения допускаются операции (*+), (*-), (*+), (*-), (*+), (*-),

$$(1-2)*(4+5)$$

запишется в следующем виде:

круглые скобки при этом не нужны.

Реализация оказывается весьма простой. Каждый операнд помещается в *стек*; когда поступает знак операции, нужное количество операндов вынимается из *стека*, к ним применяется операция и результат направляется обратно в *стек*. Так, в приведенном выше примере 1 и 2 помещаются в стек и затем заменяются их разностью, —1. После этого 4 и 5 вводятся в стек и затем заменяются своей суммой 9; далее числа —1 и 9 заменяются в стеке на их произведение, равное —9. Операция « = » печатает верхний элемент стека, не удаляя его (так что промежуточные вычисления могут быть проверены).

Сами операции помещения чисел в стек и их извлечения оформляются в виде функций. Кроме того, используется отдельная функция для выборки из ввода следующей операции или операнда. Требуемая структура программы приведена ниже [7].

```
while(поступает операция или операнд, а не конец файла)
if( число )
поместить его в стек
else if( операция )
вынуть операнды из стека
выполнить операцию
поместить результат в стек
else
ошибка
```

Операции «взять из стека» и «послать в стек» реализуем в виде функция. Добавим механизм обнаружения и нейтрализации ошибок [7].

```
#include<stdlib.h>
#include<stdio.h>
#include<ctype.h>
#include<math.h>
#include <string.h>
#define MAXOP 100
                        /*максимальный размер оператора*/
#define NUMBER
                      0 /*признак числа*/
#define IDENTIFIER
#define TRUE 1
#define FALSE 0
int Getop(char s[]);
void push(double val);
double pop(void);
void showTop(void);
void duplicate(void);
void swapItems(void);
void clearStack();
void dealWithName(char s[]);
int main(void)
   int type;
   double op2;
   char s[MAXOP];
   int flag = TRUE;
   while((type = Getop(s)) != EOF)
      switch(type)
      case NUMBER:
         push (atof(s));
         break;
      case IDENTIFIER:
         dealWithName(s);
         break;
      case '+':
         push(pop() + pop());
         break;
      case '*':
         push(pop() * pop());
         break;
```

```
case '-':
         op2 = pop();
         push(pop() - op2);
         break;
      case '/':
         op2 = pop();
         if(op2)
            push(pop() / op2);
         else
            printf("\nОшибка! Деление на ноль.");
         break;
      case '%':
         op2 = pop();
         if(op2)
            push(fmod(pop(), op2));
         else
            printf("\nОшибка! Деление на ноль.");
         break;
      case '?':
         showTop();
         break;
      case '#':
         duplicate();
         break;
      case '~':
         swapItems();
         break;
      case '!':
         clearStack();
      case '\n':
         printf("\n\t%.8g\n", pop());
         break;
      default:
         printf("\nОшибка! Неизвестная операция. %s.\n",
s);
         break;
      }
   return EXIT SUCCESS;
```

Поскольку операции «+» и «*» коммутативны, то порядок, в котором операнды берутся из стека, не важен. Однако, в случае операций «–» и «/» левый и правый операнды должен различаться. Так, в строке

```
push(pop() - pop());
```

порядок, в котором выполняются обращения к рор() не определен. Для гарантии правильного порядка первое значение из стека присваивается ор2.

```
#define MAXVAL 100
                     /*максимальная глубина стека*/
int sp = 0;
                     /* позиция следующего свободного */
                     /* элемента стека*/
double val[MAXVAL]; /* crek */
/* push: поместить f в стек */
void push(double f)
   if(sp < MAXVAL)</pre>
      val[sp++] = f;
   else
      printf("\nОшибка: стек полон, элемент %q не
помещается!\n", f);
/*рор: взять значение с вершины стека.*/
double pop(void)
   if(sp > 0)
      return val[--sp];
   else
      printf("\nОшибка! Стек пуст. \n");
      return 0.0;
   }
```

Стек и индекс стека которые должны быть доступны для *push* и *pop* определены вне этих функций. Но так как в *main* не используют ни стек, ни индекс, то они скрыты от *main*.

Займемся реализацией *getop* — функции, получающей следующий оператор или операнд. Требуется пропустить пробелы и табуляции; если следующая литера — не цифра и не десятичная точка, то выдать ее; в противном случае накопить цепочку цифр с десятичной точкой, если она есть, и выдать признак числа *NUNBER* в качестве результата.

```
int getch(void);
void unGetch(int);
```

```
/* Getop() получает следующий оператор или операнд */
int Getop(char s[])
   int i = 0;
   int c;
   int next;
   /*size t len;*/
   /* пропуск пробела */
   while ((s[0] = c = getch()) == ' ' | c == ' t') ;
   s[1] = ' \ 0';
   if(isalpha(c))
      i = 0;
      while (isalpha (s[i++] = c))
         c = getch();
      s[i - 1] = ' \setminus 0';
      if(c != EOF)
         unGetch(c);
      return IDENTIFIER;
   }
   /* Не число, но, возможно, содержит унарный минус */
   if(!isdigit(c) && c != '.' && c != '-')
      return c;
   if(c == '-')
      next = getch();
      if(!isdigit(next) && next != '.')
         return c;
     c = next;
   }
   else
      c = getch();
   while (isdigit (s[++i] = c))
      c = getch();
   if(c == '.')
                         /* накапливаем дробную часть */
      while (isdigit (s[++i] = c = getch()))
```

```
;
    s[i] = '\0';
    if(c != EOF)
        unGetch(c);
    return NUMBER;
}
s[i]='\0';
if (c!=EOF)
ungetch (c);
return NUMBER;
}
```

Как работают функции *getch* и *ungetch*? Во многих случаях программа не может определить прочла ли она все, что требуется, пока не прочтет лишнего. Поэтому *getch* читает очередную литеру из ввода, *ungetch* отправляет «лишнюю» литеру назад.

Поскольку функции совместно используют буфер и индекс, то они являются внешними по отношению к ним.

```
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;

/* Getch: получить символ. */
int getch(void)
{
   return (bufp > 0) ? buf[--bufp]: getchar();
}

/* unGetch: поместить символ назад в стек. */
void unGetch(int c)
{
   if(bufp >= BUFSIZE)
      printf("\nUnGetch: слишком много символов.\n");
   else
      buf[bufp++] = c;
}
```

Функции showTop, dublicate, swapItem, clearStack и dealWithName.

```
void showTop(void)
```

```
if(sp > 0)
      printf("Ha вершине стека лежит: %8g\n", val[sp-1]);
   else
      printf("Cτeκ πycτ!\n");
void duplicate(void)
   double temp = pop();
   push(temp);
   push(temp);
}
void swapItems(void)
   double item1 = pop();
   double item2 = pop();
   push(item1);
   push(item2);
void clearStack(void)
   sp = 0;
/*Функция работы с математическими функциями.*/
void dealWithName(char s[])
   double op2;
   if ( 0 == strcmp(s, "sin"))
      push(sin(pop()));
   else if (0 == strcmp(s, "cos"))
      push (cos (pop()));
   else if (0 == strcmp(s, "exp"))
      push (exp(pop()));
   else if(!strcmp(s, "pow"))
      op2 = pop();
      push(pow(pop(), op2));
```

```
}
else
printf("Функция %s не поддерживается0.\n", s);
}
```

§2.3. Однонаправленные связанные списки

Связанный список является простейшим типом данных динамической структуры. Компоненты связанного списка можно помещать и исключать произвольным образом [1, 3, 8, 11]. Графическое представление связанного списка показано на рис. 2.1.

Элемент списка называется узлом. Каждый узел содержит два поля: поле информации info и поле следующего адреса ptrn. Поле адреса хранит указатель на следующий элемент. Доступ к списку осуществляется через указатель lst, который содержит адрес первого элемента. Поле адреса последнего элемента хранит нулевое значение NULL. Пустой список инициализируется как lst = NULL.

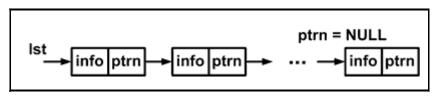


Рис. 2.1

Введем обозначения [11]: пусть p — указатель на элемент списка; тогда info(p) — ссылка на информационную часть элемента; ptrn(p) — ссылка на адрес следующего элемента; p = getnode() — выделение участка оперативной памяти под элемент списка; freenode(p) — освобождение участка оперативной памяти, отведенного под элемент списка.

Операции над списком

1. Размещение произвольного элемента в начало списка [11]

p=getnode();	Выделение места под узел. Запись в переменную р адреса узла.
info(p)=x;	Запись значения х в информационную часть узла.
ptrn(p)=lst;	Подключение узла к началу списка.
lst=p;	Установка указателя lst на новый узел.

2. Удаление первого элемента списка [11]

3. Размещение элемента после узла с адресом p [11]. Новый элемент вследствие однонаправленности списка может помещаться только после узла, адрес которого известен. Поэтому для вставки узла необходимо поступить следующим образом: новый элемент размещаем после старого, затем меняем указатели. Пусть операция insafter(p, x) означает операцию вставки элемента со значением x после p.

Oперация insafter(p, x)

g=getnode();	Выделяем память под узел.
info(g)=x;	Записываем в новый узел значение х.
ptrn(g)=ptrn(p);	Записываем в адресную часть нового узла указатель на элемент, следующий за р.
ptrn(p)=g;	Записываем в адресную часть узла р указатель g.

4. Удаление элемента после узла с адресом p [11]. В однонаправленном связанном списке можно удалить только элемент, следующий за данным узлом. Пусть default(p, x) — операция удаления элемента со значением x, следующего за узлом с адресом p.

Oперация default(p, x)

g=ptrn(p);	Установка у	указателя р на	адрес
	элемента, с	следующего за р	•

Примеры

1. Из списка lst удаляются все элементы, равные 4. Имеем два указателя: p — для просмотра списка; q — указатель на элемент перед p. Используем операции удаления: pop — из начала списка; default — из середины списка.

```
q=NULL; /*инициализация указателя*/
p=lst; /*текущий указатель на начало списка*/
while (p!=NULL)
{
   if (info(p) == 4)
      if (q==NULL)
          /*удалить первый элемент*/
          x=pop(lst);
          freenode(p);
          p=lst;
      else
          /*передвинуть р и удалить элемент, */
          /*следующим за node(q)*/
          p=ptrn(p);
          default(q, x);
   else
      /*продолжить просмотр, передвинуть р и q*/
      q=p;
      p=ptrn(p);
```

2. Список lst упорядочен по возрастанию. Вставить x, не нарушая упорядоченность. Используем операции push для добавления элемента к началу списка, insafter — внутрь списка.

```
q=NULL;
p=lst;
while((p!=NULL) && (x>info(p))
{
    q=p;
    p=ptrn(p);
}
/*pазмещаем x*/
if(q == NULL)
    push(lst, x); /*поместить в начало списка*/
else
    insafter(q,x);
```

§2.4. Однонаправленные циклические списки

Одним из недостатков линейных списков является то, что имея указатель p на элемент, мы не можем получить доступа к предшествующим элементам.

Циклический связанный список — это список, в котором последний узел связан с первым узлом [11]. Чтобы превратить линейный список в циклический необходимо в поле адреса последнего элемента записать не NULL, а адрес первого элемента списка lst (рис. 2.2).

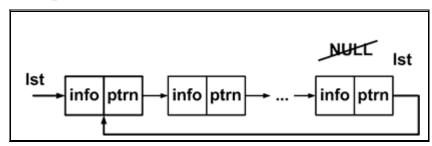


Рис. 2.2. Графическое представление циклического списка

В циклическом списке всегда существует возможность доступа к любому элементу, начиная с произвольно выбранного узла.

Простейшие операции над циклическим списком [11]:

- 1) размещение нового узла после элемента с известным адресом p;
- 2) удаление узла после элемента с известным адресом p.
- В виде циклического стека организуются стеки и очереди.

Недостатки циклического списка:

- 1) список нельзя просматривать в обратном направлении;
- 2) располагая только значением указателя на элемент, его невозможно удалить.

Для преодоления этих недостатков используются *двунаправленные связанные списки*.

§2.5. Двунаправленные связанные списки

Каждый элемент двунаправленного связанного списка имеет два указателя (рис. 2.3) [8, 11]:

- один указатель установлен на предшествующий элемент;
- другой указатель установлен на следующий элемент.

Двунаправленный список может быть линейным и циклическим (рис. 2.4).

Над линейными двунаправленными списками могут выполняться следующие операции [8]:

- \bullet получение доступа к k-му узлу для проверки или изменения содержимого его полей;
 - вставка нового узла сразу после или до k-го узла;
 - удаление *k*-го узла;
 - объединение нескольких списков в один;
 - разбиение списка на несколько списков;
 - создание копии списка;
 - определение количества узлов в списке;
 - сортировка узлов в порядке возрастания значений в определенных полях;
 - поиск узла с заданным значением в некотором поле.

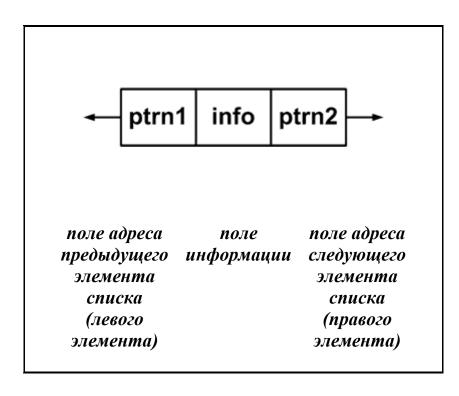


Рис. 2.3. Элемент двунаправленного списка

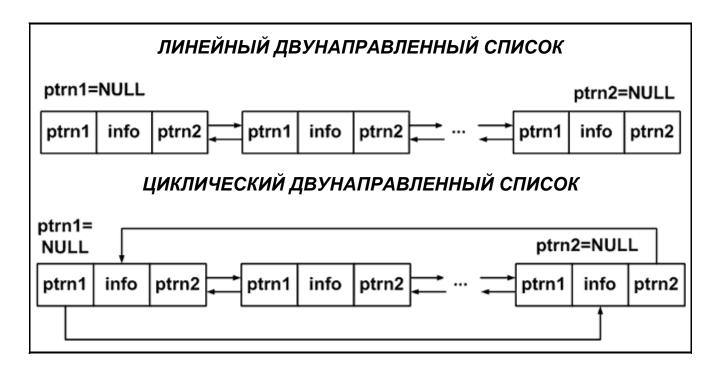


Рис. 2.4. Графическое представление двунаправленных связанных списков

Двунаправленные списки имеют следующее удобное свойство [11]: если p — указатель на элемент, то

```
left(right(p))=p=right(left(p)),
```

где $\operatorname{right}(p)$ — ссылка на адрес следующего элемента; $\operatorname{left}(p)$ — ссылка на адрес предыдущего элемента.

Операция, которую можно выполнить над двунаправленным списком, но нельзя над обычным — это удаление элемента с заданным адресом.

Пример удаления элемент с заданным адресом *ptr*

```
if( ptr == NULL )
    printf("Удаление запрещено.");
else
{
    x = info(ptr);
    g = left(ptr);
    r = right(ptr);
    right(g) = r;
    left(r) = g;
    freenode(ptr);
}
```

Реализация списков на языке Си

1. Узел однонаправленного списка

```
struct NODE
{
    < TMT> info;
    struct NODE *ptrn;
};
```

2. Узел двунаправленного списка

```
struct NODE
{
   struct NODE *ptrn1;
   <Tuπ> info;
   struct NODE *ptrn2;
};
```

3. Указатель на список

```
struct NODE *lst;
struct NODE *p;
```

4. Выделение блока оперативной памяти под узел

```
p=(struct NODE*)malloc( sizeof(struct NODE) );
```

5. Обращения к полям узла

Однонаправленный список

```
p->ptrn=...;
p->info=...;
```

Двунаправленный список

```
p->ptrn1=...;
p->info=...;
p->ptrn2=...;
```

6. Удаление узла

```
free(p);
```

7. Перемещение на следующий элемент списка

Однонаправленный список

```
p=p->ptrn;
```

Двунаправленный список

```
p=p->ptrn1; /*перемещение влево */
p=p->ptrn2; /*перемещение вправо*/
```

Примеры работы со списками

1. Определение количества узлов в линейном однонаправленном списке (рис. 2.5).

```
p=lst;
k=0;
while(p!=NULL)
{
    k++;
    p=p->ptrn;
}
printf("\nB списке %d узлов.",k);
```

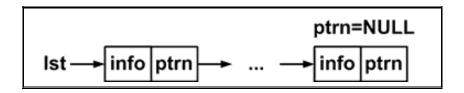


Рис. 2.5. Линейный однонаправленный список

2. Включение узла в двунаправленный связанный список после элемента с адресом р (рис. 2.6).

```
p2=(struct NODE*)malloc(sizeof(struct NODE));
```

```
p3=p->ptrn2;
p->ptrn2=p2;
p2->ptrn1=p;
p2->ptrn2=p3;
p3->ptrn1=p2;
p2->info=x;
```

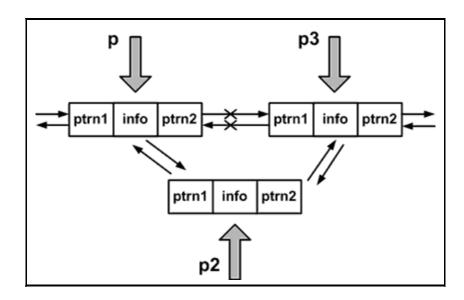


Рис. 2.6

3. Удаление узла с адресом р из двунаправленного связанного списка (рис. 2.7).

```
x=p->info;
q=p->ptrn1;
r=p->ptrn2;
q->ptrn2=r;
r->ptrn1=q;
free(p);
```

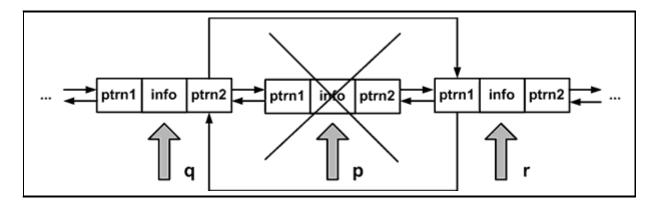


Рис. 2.7

4. Создание линейного двунаправленного списка, состоящего из п узлов.

```
lst=(struct NODE*)malloc(sizeof(struct NODE));
lst->ptrn1=NULL;
lst->info=...;
p1=lst;
for(i=1;i<=n-1;i++)
{
    p1->ptrn2=(struct NODE*)
        malloc(sizeof(struct NODE));
    p2=p1;
    p1=p1->ptrn2;
    p1->info=...;
    p1->ptrn1=p2;
}
p1->ptrn2=NULL;
```

Пример. Рассмотрим реализацию списка с двойными связями. Каждый узел содержит три поля: обратный указатель B, прямой указатель F и данные *info* (рис. 2.8). Рассматриваемый список является *циклическим*.

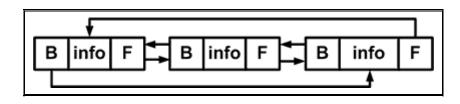


Рис. 2.8. Пример

Определим структуру, представляющую узел списка.

Узел списка

Разработаем программу, которая упорядочивает имена в поле *info* по алфавиту. Функция *main* в диалоге запрашивает вводимые имена и вызывает функцию *insert*, которая образует узел для очередного имени и включает его в соответствующее место списка.

Чтобы сохранить информацию о положении начального узла списка создадим особый узел, называемый *головой списка*. Прямому и обратному указателям головы присваивается адрес его начала, тем самым образуется пустой список.

Функция *insert* просматривает список, пока не найдет место для имени.

Пример создания связанного списка имен, упорядоченных по алфавиту

```
#include <stdio.h>
#define NAME_SIZE 30

struct NODE

{
    struct listnode*pred;
    struct listnode*succ;
    char data[NAME_SIZE];
};

main()

{
    char name [NAME_SIZE];
    struct NODE *root; /*Fonoba cnucka.*/
    struct NODE *np; /*Wcnonbsyetca npu npocmotpe.*/

/*Занять блок памяти для головы списка и иницилизировать
его так, чтобы он показывал сам на себя.*/
    root=malloc(sizeof(struct NODE));
```

```
root->pred= root->succ=root;
    root->data[0]='\0';
/*Создать связанный список имен.*/
    for (;;)
       printf ("имя:");
       gets (name);
       if (strcmp(name), "конец") == 0)
          break;
       if (insert(root, name) == 0)
            printf ("не хватает динамической памяти \n");
            exit(1);
/*Изобразить содержимое списка.*/
    for (np=root->succ; np!=np->succ)
       printf ("имя=%s \n", np->data);
   printf ("работа закончена \n");
 }
/*Зарезервировать память для нового узла списка;
скопировать в него имя и вставить новый узел в список в
алфавитном порядке. Возвратить либо указатель на новый
узел, либо NULL, если для создания узла не хватило
памяти.*/
 struct NODE *insert (node, name);
 struct NODE *node; /*Голова списка.*/
 char *name;
   NODE *np; /*Узел списка.*/
   NODE *newnode;
/*Узел, вставленный в список.*/
/*Просматривать список, пока не обнаружится узел, поле
info которого имеет значение, большее введенного имени
или равное emy.*/
    for (np=node->succe; (np!=node) &&
    strcmp (name, np->data)>0); np=np->succe);
```

```
/*Зарезервировать память для нового узла; поместить
введеное имя в его поле info и вставить новый узел перед
тем, на который показывает указатель пр.*/
    if( (newnode=malloc(sizeof(struct NODE)))!=0 )
       strncpy(newnode->data, name, NAME SIZE);
       newnode->succ=np;
       newnode->pred=np->pred;
/*Изменить прямой указатель в узле, предшествующем
вставленному (теперь он должен показывать на вставленный
узел) и изменить обратный указатель в узле, следующем за
вставленным. */
       (newnode->pred) ->succ=newnode;
       np->pred=newnode;
    }
 return (newnode);
 }
```

§2.6. Очереди

Очередью называется упорядоченный набор элементов, которые могут удаляться с *начала очереди* и помещаться в *конец очереди* (рис. 2.9). Очередь организована в отличие от стека по принципу *FIFO* (first input — first output; первым вошел — первым вышел) [1, 3, 8, 11].

Для очереди определены три простейшие операции:

- insert(q, x) помещает элемент x в конец очереди q, где q указатель на очередь;
 - x = remove(q) удаляет элемент x из очереди q;
- \bullet empty(q) возвращает true или false в зависимости от того, является ли очередь пустой или нет.

Реализация очереди на базе массива

Рассмотрим реализацию очереди на базе массива. Используем, например, массив и две переменные frnt и rear, которые запоминают позиции первого и последнего элементов. Изначально frnt = 1 и rear = 0. Очередь пуста, если rear < frnt. Число элементов в очереди rear - frnt + 1.

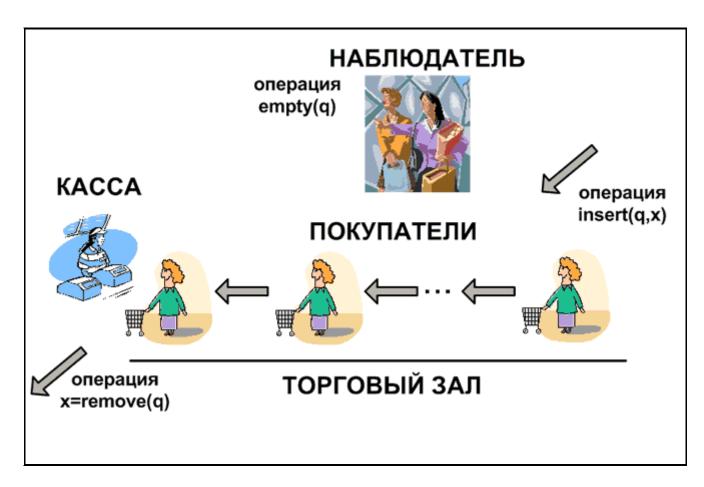


Рис. 2.9. Графическое представление очереди

Объявление пустой очереди

```
#define Maxg 5
float git[Maxg];
frnt=1;
rear=0;
```

Игнорируя возможность переполнения очереди, операцию insert(g, x) запишем следующим образом:

```
rear++; git[rear]=x; , a операцию x=\mathrm{remove}(g) так: x=git[frnt]; frnt++; .
```

На рис. 2.10 показано, что произойдет при таком представлении.

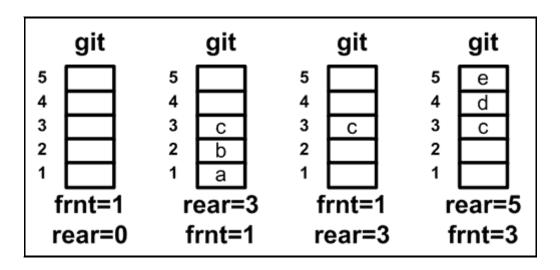


Рис. 2.10. Элементы в очереди

Изначально очередь пуста. Размер массива *git* будет 5. В результате выполнения операции *insert* и *remove* в очереди будут находиться три элемента. Больше поместить нельзя.

Одним из решений этой проблемы является модификация операции *remove* таким образом, чтобы при удалении элемента смещать очередь к началу.

Пример

```
x = git[0];
for(i=0; i<rear-1; i++)
{
    git[i] = git[i+1];
}
rear--;</pre>
```

Переменной frnt не требуется, так как первый элемент — начало очереди. Для пустой очереди rear = 0. Метод непроизводителен, так как требует перемещение оставшихся элементов.

Другой способ предлагает рассматривать массив в виде циклического связанного списка. **Недостаток** — трудно определить, когда очередь пуста, так как условие rear < frnt не выполняется.

Рассмотрим следующий пример (рис. 2.11). Одним из способов решения проблемы является соглашение о том, что frnt является индексом элемента, предшествующего первому элементу. В этом случае для пустой очереди frnt = rear.



Рис. 2.11. Пример

Инициализация очереди

```
#define Maxg 100
float git[Maxg];
frnt = Maxg;
rear = Maxg;
```

Операция *етрtу*

```
if(frnt == rear) empty=1;
else empty =0;
```

Операция remove

```
if (empty == 1)
  printf("Очередь пуста");
if (frnt == Maxg-1) frnt = 0;
else frnt = frnt + 1;
remove = git[frnt];
```

При реализации вставки необходимо контролировать ситуацию переполнения, при которой frnt = rear. Это же условие характеризует пустую очередь. Одно из решений проблемы — очередь растет до Maxg-1.

Операция insert

```
/*выделение места для элемента*/
if (rear == Maxg - 1) rear = 0;
else rear = rear + 1;
```

```
/*проверка на переполнение*/
if (rear == frnt)
   printf("Переполнение.");
git[rear] = x;
```

Функции, реализующие операции работы с очередью

```
int empty()
   if(frnt == rear) return 1;
   else return 0;
}
float remove()
   if(empty() == 1)
      printf("Очередь пуста.");
      return 0.0;
   }
   else
      if (frnt == (Maxq-1)) frnt=0;
      else frnt++;
      return git[frnt];
    }
}
void insert(float x)
   if (rear == (Maxq-1)) rear = 0;
   else rear++;
   if(rear == frnt)
      printf("Переполнение!");
   else git[rear]=x;
```

Реализация очереди на базе однонаправленного связанного списка

Однонаправленный связанный список можно рассматривать в качестве очереди поскольку (рис. 2.12): а) определены начало и конец списка, б) задан порядок расположения узлов.

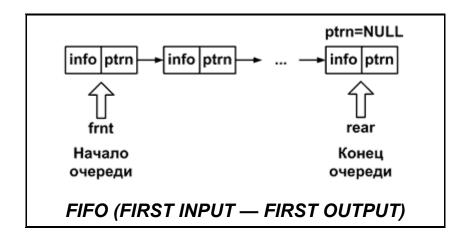


Рис. 2.12

1. Операция проверки пустоты очереди етру

```
if(frnt == NULL && rear == NULL)
printf("Очередь пуста.");
```

2. Операция удаления элемента из очереди remove (рис. 2.13)

```
x=frnt->info;
p=frnt;
frnt=frnt->ptrn;
free(p);
```

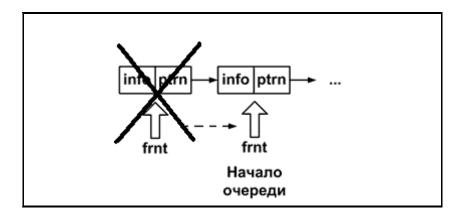


Рис. 2.13

3. Операция помещения элемента в очередь *insert* (рис. 2.14)

```
p=(struct NODE*)malloc(sizeof(struct NODE));
p->ptrn=NULL;
```

```
p->info=x;
rear->ptrn=p;
rear=p;
```

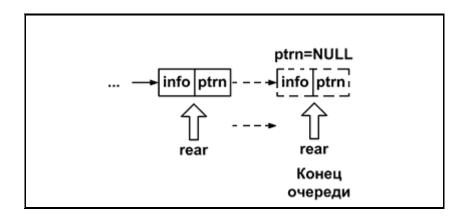


Рис. 2.14

Недостатки представления очереди и стека в виде связанного списка:

- Элемент списка занимает в оперативной памяти больше места, чем элемент массива.
- Требуется дополнительное время на обработку списка: необходимо выделять блоки оперативной памяти под узлы и изменять значения указателей.

§2.7. Бинарные деревья

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом [1, 3].

Общепринятый способ изображения бинарного дерева представлен на рис. 2.15: дерево состоит из 9 узлов; A — корень; левое поддерево имеет корень B; правое — C.

Узел y, который находится непосредственно под узлом x, называется $nomokom\ x$; если x находится $ha\ ypoвhe\ i$, то y — на уровне i+1. Наоборот, узел x называется $npedkom\ y$. Считается, что корень дерева расположен на уровне 1. Максимальный уровень какого-либо элемента дерева называется его color polor po

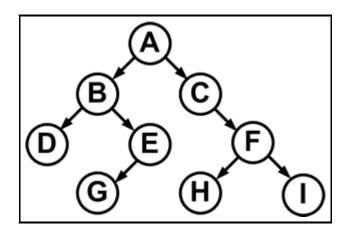


Рис. 2.15

Бинарное дерево — полезная структура данных в тех случаях, когда в каждой точке вычислительного процесса должно быть принято одно из двух возможных решений.

Узел дерева можно описать как структуру с полями.

Узел дерева

```
struct tnode
{
   char *word; /*указатель на поле info*/
   struct tnode *left; /*левый потомок*/
   struct tnode *right; /*правый потомок*/
};
struct tnode *root; /*указатель на узел*/
```

Имеется много задач, которые можно выполнять на дереве; распространенная задача — выполнение заданной операции p с каждым элементом дерева. Здесь p рассматривается как параметр более общей задачи посещения всех узлов или задачи обхода дерева.

Если рассматривать задачу как единый последовательный процесс, то отдельные узлы посещаются в определенном порядке и могут считаться расположенными линейно.

Пусть имеем дерево, где R — корень, A и B — левое и правое поддеревья (рис. 2.16). Существует три способа обхода дерева:

- 1. Обход дерева сверху вниз (в прямом порядке): R, A, B.
- 2. В симметричном порядке (слева направо): A, R, B.
- 3. В обратном порядке (снизу вверх): A, B, R.

Функции обхода дерева удобно выразить рекурсивно.

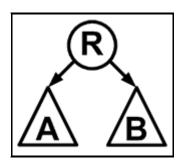


Рис. 2.16. Дерево

Пример. Пусть имеем дерево для арифметического следующего выражения (рис. 2.17):

$$(a + b/c)*(d - e*f).$$

Обходя дерево выписывая символы в узлах в том порядке, как они встречаются, получаем:

- 1. Сверху вниз: *+a/bc-d*ef префиксная запись выражения.
- 2. Слева направо: a+b/c*d-e*f инфиксная запись без скобок, которые определяют порядок действий.
 - 3. Снизу вверх: $abc/+def^*-^*$ постфиксная запись.

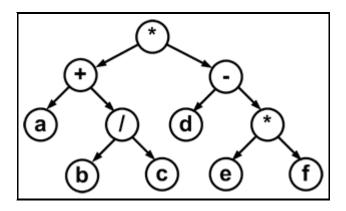


Рис. 2.17. Дерево

Эти три способа обхода легко сформулировать в виде функций.

Функция обхода дерева сверху вниз

Функция обхода дерева слева направо

```
void postorder (struct node *t)
{
   if( t!=NULL)
   {
     postorder (t->left);
     postorder (t->right);
     P(t);
   }
}
```

Функция обхода дерева снизу вверх

```
void inorder (struct node *t)
{
   if(t!=NULL)
   {
     inorder (t->left);
     P(t);
     inorder (t->right);
   }
}
```

Здесь P — операция, которую надо выполнить с узлом; t — указатель на дерево.

В заключение рассмотрим пример [7]. Необходимо написать программу, подсчитывающую частоту встречаемости для любых слов входного потока. Поскольку список слов заранее не известен, мы не можем предварительно упорядочить его и применить бинарный поиск. Было бы неразумно пользоваться и линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет. В этом случае программа работает слишком медленно.

Как можно организовать данные, чтобы эффективно справиться со списком произвольных слов?

Один из способов — постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Воспользуемся структурой данных, называемой *бинарным деревом*.

В дереве на каждое отдельное слово предусмотрим узел, который содержит:

- 1) указатель на текст слова;
- 2) счетчик числа встречаемости;
- 3) указатель на левого потомка;
- 4) указатель на правого потомка.

У каждого узла может быть один или два сына, или узел может вообще не иметь сыновей.

Узлы в дереве располагаются так, что по отношению к любому узлу левое поддерево содержит только те слова, которые лексикографически меньше, чем слово данного узла, а правое — слова, которое больше него. Вот как выглядит дерево, построенное для фразы «now is the time for all good men to come to the aid of their party» (настало время всем добрым людям помочь своей партии), после завершения процесса (рис. 2.18).

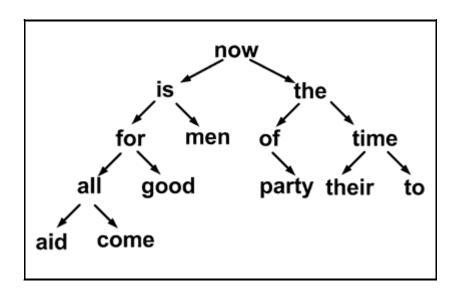


Рис. 2.18. Дерево

Чтобы определить, помещено ли в дерево новое слово, начинают с корня, сравнивая слева со словом из корневого узла. Если совпали, ответ положительный. Если новое слово меньше слова из дерева, то поиск продолжается в левом поддереве, если больше, то в правом. Если же в выбранном направлении поддерева нет, то этого слова в дереве нет, а пустующая ссылка — место, куда помещается новое слово. Описанный процесс — рекурсивен, поэтому применяем рекурсивные функции.0

Описание узла

```
struct tnode
{
   char *word; /*указатель на строку*/
   int count; /*число вхождений*/
   struct tnode *left; /*левый потомок*/
   struct tnode *right; /*правый потомок*/
};
```

Главная программа *main* читает слова с помощью *getword* и вставляет их в дерево посредством *addtree*.

Функция *addtree* рекурсивна. Первое слово функция *main* помещает в корень дерева. Каждое новое слово сравнивается со словом узла и погружается в левое или правое поддерево. При совпадении слова с каким-либо словом дерева к счетчику *count* добавляется 1, в противном случае создается новый узел.

Память для нового узла запрашивается программой *talloc*, которая возвращает указатель на свободное пространство; копирование нового слова осуществляет функция *strdup*. В программе для простоты опущен контроль ошибок.

Функция *treeprint* печатает дерево в лексикографическом порядке. Для каждого узла вначале печатается левое поддерево, затем узел, затем правое поддерево.

Функция talloc распределяет память.

Функция strdup копирует строку, указанную в аргументе, в место, полученное с помощью *malloc*.

Функция *getword* принимает следующее слово или литеру из ввода. Она обращается к *getch* и *ungetch*. После окончания набора букв-цифр оказывается, что *getword* взяла лишнюю литеру. Она возвращается функцией *ungetch* назад во входной поток.

В *getword* используются следующие функции: 1) *isspace* — пропуск пробельных литер; 2) *isalpha* — идентификация букв; 3) *isalnum* — идентификация букв-цифр.

Функции getch и ungetch совместно используют буфер и индекс, значения которых должны сохраняться между вызовами, поэтому они должны быть внешними.

Функция *ungetch* возвращает одну литеру.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXWORD 100
#define BUFSIZE 100
                               /* узел дерева: */
struct tnode {
                               /* указатель на строку */
 char *word;
                               /* число вхождений */
 int count;
 struct tnode *left;
                               /* левый потомок*/
 struct tnode *right;
                               /* правый потомок */
};
```

```
struct tnode *addtree(struct tnode *, char *);
struct tnode *talloc(void);
void treeprint(struct tnode *);
void ungetch(int);
int getword(char *, int);
int getch (void);
                    /* буфер для ungetch */
char buf[BUFSIZE];
int bufp = 0;
                          /* следующая свободная */
                           /* позиция в буфере */
                          /* ввод строки символов */
int main(void) {
 struct tnode *root;
char word[MAXWORD];
root = NULL;
while(getword(word, MAXWORD) != EOF)
  if(isalpha(word[0]))
    root = addtree(root, word);
treeprint(root);
exit(0);
/* Функция getword() получает следующее слово из входного
потока. */
int getword(char *word, int lim) {
int c, getch (void);
void ungetch(int);
char *w = word;
while(isspace(c = getch()))
if(c != EOF)
 *w++ = c;
if(!isalpha(c)) {
 *_W = ' \setminus 0';
 return c;
for (; --lim > 0; w++)
 if(!isalnum(*w = getch())) {
  ungetch (*w);
  break;
 *w = ' \setminus 0';
return word[0];
```

```
/* Функция addtree() добавляет узел к дереву.*/
struct tnode *addtree(struct tnode *p, char *w) {
 int cond;
 if(p == NULL) {
 p = talloc()
 p->word = strdup(w);
 p->count = 1;
 p->left = p->right = NULL;
 } else if ((cond = strcmp(w, p->word)) == 0)
 p->count++;
 else if(cond < 0)</pre>
 p->left = addtree(p->left, w);
 p->right = addtree(p->right, w);
 return p;
 /* Функция talloc() выделяет память под узел. */
struct tnode *talloc(void) {
 return(struct tnode *)malloc(sizeof(struct tnode));
 /* Функция treeprint() печатает дерево. */
void treeprint(struct tnode *p) {
 if(p != NULL) {
 treeprint(p->left);
 printf("%4d %s\n", p->count, p->word);
  treeprint(p->right);
}
int getch(void) {
 return (bufp > 0) ? buf[--bufp] : getchar();
void ungetch(int c) {
 if(bufp >= BUFSIZE)
 printf("ungetch: слишком много символов\n");
 else
 buf[bufp++] = c;
```

Глава 3. Сортировка и поиск

§3.1. Введение в поиск

Алгоритмы поиска занимают важное место в прикладных алгоритмах. Обычно данные хранятся в определенном образом упорядоченном наборе. Найти некоторую запись из этого набора — вот классическая задача программирования, вокруг которой сгенерировано множество идей [1, 3, 9, 10, 13].

Пусть мы имеем таблицу, состоящую из записей (табл. 3.1). Первое поле каждой записи содержит *ключ* (например, табельный номер); второе — фамилию и так далее. Ключом может любое поле записи.

Таблица 3.1

14	Иванов	
2	Андреев	
308	Сидоров	
•••		
1026	Петров	•••

Основная задача поиска — найти запись с заданным ключом.

Все алгоритмы поиска, в зависимости от того, упорядочена таблица или нет, разбиваются по две большие группы. Упорядоченность понимается как наличие хотя бы одного отсортированного поля — а именно, ключевого.

§3.2. Последовательный поиск

Наиболее примитивный, а значит, и наименее эффективный, способ поиска — это обычный последовательный просмотр записей таблицы [9, 11]. Метод применяется к таблице, организованной как массив. Предложим, что k — массив из n ключей; r — массив записей такой, что k(i) — ключ для записи r(i); key — аргумент поиска. Запишем в переменную search наименьшее целое число i, такое, что k(i) = key, если такое i существует, и -1 в противном случае.

Алгоритм последовательного поиска

```
for( search=-1, i=0; i<n; i++ )
   if(key == k[i])
   {
      search=i;
      break;
   }</pre>
```

Некоторым улучшением этой идеи является *метод транспозиции*: каждый запрос к записи сопровождается сменой мест этой и предшествующий записи; в итоге наиболее часто используемые записи постепенно перемещаются в начало таблицы; и при последующем обращении к ней запись будет находиться почти сразу.

На подобной идее основан и *метод перемещения в начало*: каждый запрос к записи сопровождается её перемещением в начало таблицы; в итоге в начале таблицы оказываются записи, используемые в последнее время.

§3.3. Поиск в упорядоченной таблице

Все реально применяемые методы поиска относятся к *отвертированным таблицам*. Для упорядоченной таблицы наиболее эффективными являются: 1) индексно-последовательный поиск и 2) бинарный поиск .

Индексно-последовательный поиск. В дополнение к отсортированной таблице заводится вспомогательная таблица, называемая индексной [9, 11]. Каждый элемент индексной таблицы состоит из ключа и указателя на запись в основной таблице, соответствующую этому ключу kindex. Элементы в индексной таблице, так же как элементы в основной таблице, должны быть отсортированы по этому ключу. Если индекс имеет размер, составляющий одну восьмую от размера основной таблицы, то каждая восьмая запись основной таблицы будет представлена в индексной таблице (рис. 3.1).

Алгоритм индексно-последовательного поиска прост. Предположим, что k — массив из n ключей; r — массив записей такой, что k(i) является ключом для записи r(i); key — аргумент поиска. Пусть — массив ключей в индексе; pindex — массив указателей в индексе на записи. Размер ochobhoù maблицы — n. Размер uhdekchoù maблицы — ind_size .

Алгоритм поиска

```
i=0;
while((i < ind_size) && (kindex[i] <= key))
    i++;

/*установить low на наименьшую возможную позицию
элемента в массиве*/

if(i == 0)
    low=0;
else
    low = pindex[i];

/*установить high на наибольшую возможную
позицию элемента в массиве*/</pre>
```

```
if (i==ind_size)
   high=n;
else
   high=pindex[i];

/*поиск в массиве от low до high*/

for (j=low, search=-1; j<high; j++)
   if (k[j]==key)
   {
      search=j;
      breake;
   }</pre>
```

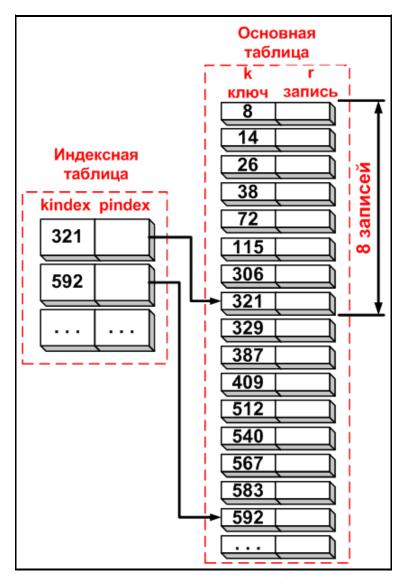


Рис. 3.1. Схема хранения информации при индексно-последовательном поиске

Достоинство алгоритма заключаются в том, что сокращаются время поиска, так как последовательный поиск первоначально ведется в индексной таблице, имеющей меньший размер, чем основная таблица; когда найден правильный индекс, второй последовательный поиск выполняется по небольшой части записей основной таблицы.

Бинарный поиск. Аргумент сравнивается с ключом среднего элемента в массиве [9, 11]. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой и правой частях массива. Алгоритм определяется рекурсивно, однако на практике применяется нерекурсивная версия ввиду её большой эффективности.

Алгоритм поиска

```
low=0;
high=n-1;
search=-1;
while(low<high)
{
    mid=(int)(low+high)/2;
    if (key==k[mid])
    {
        search=mid;
        break;
    }
    if (key<k[mid])
        high=mid-1;
    else
        low=mid+1;
}</pre>
```

Здесь key — аргумент поиска (число, которое записывается в ключевом поле искомой записи); search — переменная, которая хранит номер разыскиваемой записи; mid — номер записи, в которой осуществляется поиск необходимого значения ключевого поля; low и high — границы поиска; n — число ключей в массиве.

§3.4. Хеширование таблиц

Хеширование — один из способов увеличения эффективности поиска [9]. Основная идея хеширования — подобрать некий способ преобразования значения ключа сразу в адрес записи. Таким образом, поиск становится вообще ненужным.

Поясним это на примере. Пусть в файле прямого доступа лежит структура со следующими полями (табл. 3.2): 1) номер накладной, 2) грузоотправитель, 3) грузополучатель, 4) груз, 5) количество единиц груза. Номер накладной обычно

представляет собой довольно длинное целое число; в нашем примере это ключ. Если массив структур снабжен дополнительной таблицей индексов (табл. 3.3), в которой значения ключей отсортированы, например, по возрастанию, и для каждого значения ключа в таблице имеется номер соответствующей записи, то поиск нужной записи выполняется следующим образом: сначала проводим поиск в левой части таблицы индексов, а найдя заданный ключ, получаем номер записи — т. е. прямой доступ к ней.

Поля структуры

Таблица 3.2

Поля	Конкретные значения
Номер накладной	207008945
Грузоотправитель	Computer ltd
Грузополучатель	AO «Теледейта»
Груз	AS-400
Количество единиц груза	100

Дополнительная таблица индексов

Таблица 3.3

Ключ	Номер записи
207008282	23
401000511	112
947008945	921

Попробуем упростить процесс. Пусть мы имеем не более 1000 записей. Если бы значения ключа лежали в диапазоне 1 до 1000, то таблицу индексов можно построить иначе, а именно — заносить номер записи в ячейку таблицы, номер которой равен ключу (табл. 3.4). Ясно, что теперь для поиска записи с ключом, например, 282, достаточно было бы сразу обратиться к ячейке с номером 282 (прямой доступ), извлечь оттуда значение 23 и прочитать запись номер 23 в массиве (снова прямой доступ). Таким образом, поиск не нужен.

Хеширование есть способ перехода от длинных ключей к целым значениям, лежащим в диапазоне номеров записей. На практике хеширование проводят, подбирая некоторую функцию, отображающую все множество значений исходного ключа во множество индексов (адресов). Ясно, что диапазон значений

исходного ключа шире, чем диапазон индексов; поэтому хеширующая функция (*хеш-функция*) не может быть взаимно однозначной. Это порождает ряд проблем.

Таблица 3.4

Таблица

Номер ячейки	Номер записи
282	23
511	112
945	921

Вернемся к примеру. Если внимательно посмотреть на последнюю таблицу, то станет ясно, что в качестве *хеш-функции* мы взяли просто усечение номера накладной до последних трех цифр (это не худший метод хеширования). Однако, что делать, если имеется накладная с номером 207008282, а приходит документ с номером 010550282? Эта ситуация весьма типична и называется *коллизией хеширования*.

Имеется достаточное количество методов разрешения конфликтов. частности Возможно сформировать вторичный индекс. В онжом повторяющихся значений усеченного ключа использовать связный список. В основную запись можно было бы включить исходный ключ и проводить поиск по списку вторичных ключей до обнаружения нужного исходного. Известны и другие методы разрешения конфликтов.

§3.5. Введение в сортировку

Под *сортировкой* понимают процесс перестановки объектов данного множества в определенном порядке.

Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве. Следовательно, методы сортировки важны особенно при обработке данных [1, 3, 9, 10, 13].

Зависимость выбора алгоритма от структуры данных — явление довольно частое, и в случае сортировки она настолько сильна, что методы сортировки обычно разделяют на две категории: сортировка массивов (внутренняя сортировка) и сортировка файлов (внешняя сортировка). Массивы обычно располагаются в оперативной памяти, для которой характерен быстрый произвольный доступ; файлы хранятся в более медленной, но более вместительной внешней памяти, на дисках.

Введем некоторую терминологию [3]. Пусть даны элементы $a_1, a_2, ..., a_n$. Сортировка означает перестановку элементов в таком порядке $a_{k1}, a_{k2}, ..., a_{kn}$., что при заданной упорядочивающей функции f(x) справедливо отношение

$$f(a_{x1}) \le f(a_{x2}) \le ... \le f(a_{xn}).$$

Обычно упорядочивающая функция не вычисляется, а содержится в виде явной компоненты каждого элемента — поля. Её значение называется ключом элемента. Следовательно, для представления элемента a_i подходит структурный тип данных:

```
struct item {
  int key;
  onucaние других компонент;
}:
```

где key — ключ, служащий для идентификации элементов (выбор его типа как int произволен, можно использовать любой тип, для которого задано отношение порядка).

Сортировка массивов. Основное требование к методам сортировки массивов — экономное использование памяти. Это значит, что перестановки элементов нужно выполнять на том же месте оперативной памяти, где они находятся, и что методы, которые пересылают элементы из массива A в массив B, не представляют интереса. Таким образом, выбирая метод сортировки, руководствуясь критерием экономии памяти, классификацию алгоритмов, проводят в соответствии с их эффективностью, т.е. быстродействием. Удобная мера эффективности получается при подсчете числа необходимых сравнений ключей C и пересылок элементов C и пересыло

Рассмотрим вначале простые методы, требующие порядка n^2 сравнений элементов. Методы используются при небольших n.

Методы сортировки массивов можно разбить на три класса [3]:

- 1) сортировка включениями;
- 2) сортировка выбором;
- 3) сортировка обменом.

Сравним эти методы. Используем массив a, описанный следующим образом:

 $int\ nambers[n];$.

§3.6. Сортировка с помощью прямого включения

Элементы массива условно разделяются на готовую последовательность $a_1, a_2, ..., a_{i-1}$ и входную последовательность $a_i, a_2, ..., a_{n-1}$ [1, 3, 9, 10, 13]. На каждом шаге i-й элемент помещается на подходящее место в готовую последовательность (рис. 3.2).

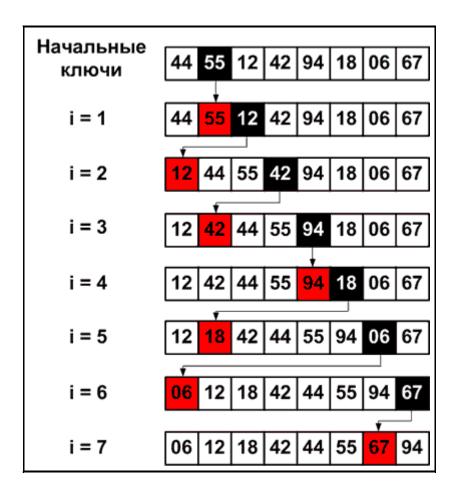


Рис. 3.2. Пример сортировки

Алгоритм сортировки

```
for( i=1; i<n; i++ )
{
    x=a[i];
    включение х на соответствующее место среди a<sub>0</sub>, ..., a<sub>i</sub>
}
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать x, т. е. x сравнивается с очередным элементом a_j , а затем либо x вставляется на свободное место, либо a_j сдвигается (передаётся) вправо и процесс «уходит» влево. Процесс просеивания закончится при выполнении одного из двух следующих условий:

- 1. Найден элемент a_j с ключом, меньшим, чем ключ у x.
- 2. Достигнут левый конец готовой последовательности.

Функция сортировки с помощью метода прямого включения

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i=1; i < array_size; i++)
    {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
       numbers[j] = numbers[j-1];
       j = j - 1;
       }
       numbers[j] = index;
    }
}
```

Анализ алгоритма. Число сравнений ключей C_i при i-м просеивании составляет самое большое i-1, самое меньшее 1. Если предположить, что все перестановки из n ключей равновероятны, то среднее число сравнений — i/2. Число пересылок M_i равно C_i +2. Поэтому общее число сравнений и пересылок таковы [3]:

$$C_{min} = n - 1$$
; $M_{min} = 3(n - 1)$;
 $C_{cp} = (n^2 + n - 2)/4$; $M_{cp} = (n^2 + 9n - 10)/4$;
 $C_{max} = (n^2 + n - 4)/4$; $M_{min} = (n^2 + 3n - 4)/2$.

Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие же оценки — когда элементы первоначально расположены в обратном порядке.

Резюме: сортировка методом прямого включения — не очень подходящий метод для ЭВМ, так как включение элемента с последующим сдвигом на одну позицию целой группы элементов неэффективно.

§3.7. Сортировка с помощью прямого выбора

Метод сортировки основан на следующих правилах [1, 3, 9, 10, 13]:

- 1. Выбирается элемент с наименьшим ключом.
- 2. Он меняется местами с первым элементом a_0 .
- 3. Затем эти операции повторяются с оставшимися n–1 элементами, n–2 элементами и так далее до тех пор, пока не останется один, самый большой элемент.

На рис. 3.3 приведен процесс сортировки этим методом.

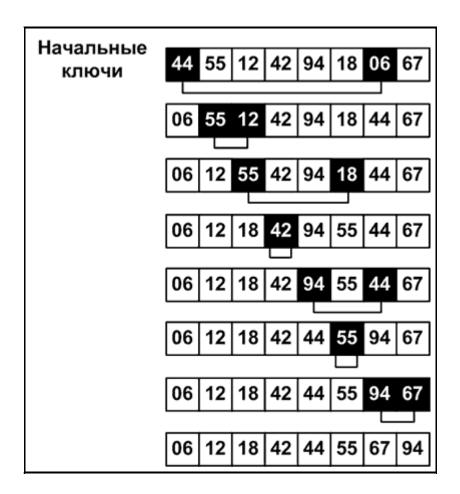


Рис. 3.3. Пример сортировки

Алгоритм формулируется следующим образом

```
for(i=0; i<n-1; i++)
{

присвоить k индекс наименьшего
элемента из a[i]..a[n-1];
поменять местами a[i] и a[k];
}
```

Сортировка прямым выбором в некотором смысле противоположена сортировке прямыми включениями. При прямом включении на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готовой последовательности для нахождения места включения. При прямом выборе для поиска одного элемента с наименьшим ключом просматриваются все элементы входной последовательности и найденный элемент помещается как очередной элемент в конец готовой последовательности.

Функция сортировки прямым выбором

```
void selectionSort(int numbers[], int array_size)
{
  int i, j;
  int min, temp;

  for( i = 0; i < array_size-1; i++ )
  {
    min = i;
    for( j = i+1; j < array_size; j++ )
    {
       if (numbers[j] < numbers[min])
         min = j;
    }
    temp = numbers[i];
    numbers[i] = numbers[min];
    numbers[min] = temp;
}
</pre>
```

Анализ алгоритма [3]. Число сравнений ключей C не зависит от порядка ключей:

$$C = 1/(2n^2 - 2n).$$

Число перестановок минимально

$$M_{min} = 3(n-1)$$

в случае изначально упорядоченных ключей и максимально

$$M_{max} = n^2/4 + 3(n-1),$$

если первоначально ключи располагаются в обратном порядке.

Среднее число пересылок

$$M_{cp} \approx n(\ln n + g),$$

где g = 0,577216... — константа Эйлера.

Резюме: как правило, алгоритм с прямым выбором предпочтительнее алгоритму прямого включения; однако, если ключи в начале упорядочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

§3.8. Сортировка с помощью прямого обмена

Алгоритм основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будет отсортирован весь массив. Как и в методе прямого выбора совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в банке с водой, причем вес

каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу (рис. 3.4). Такой метод известен под именем «пузырьковая сортировка» [1, 3, 9, 10, 13].

Функция сортировки прямым обменом

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for( i = 0; i < array_size; i++ )
  {
    for( j = (array_size - 1); j > i; j-- )
        {
        if( numbers[j-1] > numbers[j] )
            {
            temp = numbers[j-1];
            numbers[j-1] = numbers[j];
            numbers[j] = temp;
        }
    }
  }
}
```

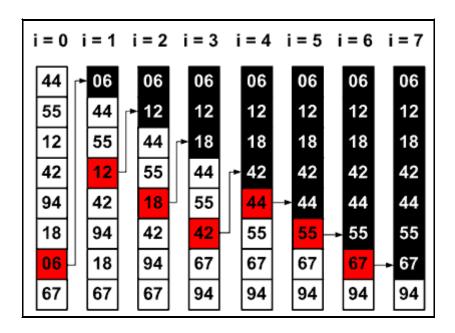


Рис. 3.4. Пример сортировки

Анализ алгоритма [3]. Число сравнений в алгоритме прямого обмена $C = (n^2 - n)/2$,

а минимальное, среднее и максимальное число перемещений элементов равно соответственно

$$M_{min} = 0,$$

 $M_{cp} = 3(n^2 - n)/2,$
 $M_{max} = 3(n^2 - n)/4.$

Резюме: «обменная сортировка» представляет собой нечто среднее между сортировками с помощью включений и с помощью выбора; фактически в пузырьковой сортировке нет ничего ценного, кроме привлекательного названия.

Далее мы рассмотрим улучшенные методы сортировки.

§3.9. Сортировка включениями с убывающим приращением

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки с помощью отомкап включения. Сам метод сортировки объясняется демонстрируется на стандартном примере (рис. 3.5). Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на 4 позиции. Такой процесс называется четвертной сортировкой. В нашем примере восемь элементов каждая группа состоит ровно из двух элементов. После первого прохода элементы перегруппировываются — теперь каждый элемент группы отстоит от другого на две позиции — и вновь сортируются. Это называется двойной сортировкой. Наконец, на третьем проходе идет обычная сортировка [1, 3, 9, 10, 13].

Сначала может показаться, что необходимость нескольких проходов сортировки, в каждом из которых участвуют все элементы, потребует большего количества машинных ресурсов, чем обычная сортировка. Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Ясно, что такой метод в результате дает упорядоченный массив, и, конечно же, сразу видно, что каждый проход от предыдущих только выигрывает (так как каждая i-сортировка объединяет две группы, уже отсортированные 2i-сортировкой). Очевидно так же, что расстояния в группах можно уменьшать поразному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу.

Функция сортировки Шелла

```
void shellSort(int numbers[], int array_size)
{
  int i, j, increment, temp;
  increment = 3;
  while (increment > 0)
  {
```

```
for (i=0; i < array_size; i++)
{
    j = i;
    temp = numbers[i];
    while((j>=increment)&&(numbers[j-increment]>temp))
    {
        numbers[j] = numbers[j - increment];
        j = j - increment;
    }
    numbers[j] = temp;
}
if (increment/2 != 0)
    increment = increment/2;
else if (increment == 1)
    increment = 0;
else
    increment = 1;
}
```

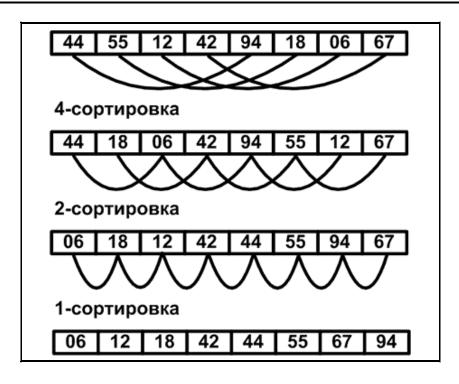


Рис. 3.5. Пример сортировки Шелла

Приводимая программа не ориентирована на некую определенную последовательность расстояний. Все t расстояний обозначаются соответственно $h_1, h_2, ..., h_t$, для них выполняются условия:

$$h_t = 1;$$

$$h_{i+1} < h_i.$$

Каждая h-сортировка программируется как сортировка с помощью прямого включения.

Анализ алгоритма. При анализе алгоритма возникают очень сложные математические задачи, многие из которых ещё не решены [3, 9]. В частности, не известно, какие расстояния дают наилучшие результаты. Однако выявлен удивительный факт, что они не должны быть множителями друг другу. Дональд Кнут рекомендует такую последовательность [9]:

где

$$h_{k-1} = 2h_k + 1;$$

 $h_t = 1 \text{ u}$
 $t = \lceil \log_2 n \rceil - 1.$

В этом случае затраты на сортировку n элементов пропорциональны $n^{1,2}$.

§3.10. Сортировка с помощью дерева

Метод сортировки с помощью прямого выбора основан на повторяющихся поисках наименьшего ключа среди n элементов, затем среди n-1 элементов и так далее. Улучшить сортировку можно в том случае, если получать от каждого прохода больше информации, чем просто идентификация единственного элемента [1, 3, 9, 10, 13]. Например, с помощью n/2 сравнений можно определить наименьший ключ из каждой пары элементов; при помощи следующих n/4 сравнений можно выбрать наименьший из каждой пары уже выбранных наименьших ключей; наконец, при помощи n-1 сравнения можно построить дерево выбора и определить корень как наименьший ключ (рис. 3.6).

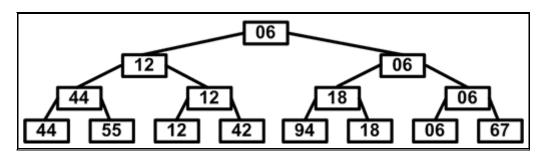


Рис. 3.6. Повторяющиеся наборы среди двух ключей

На втором шаге спускаемся по пути, указанному наименьшим ключом и исключаем его, последовательно заменяя на «дыру», либо на элемент, находящийся на противоположной ветви промежуточного узла (рис. 3.7).

Элемент, оказавшийся в корне дерева, вновь имеет наименьший ключ и может быть исключен. После n шагов дерево становится пустым, и процесс сортировки заканчивается. Каждый из n шагов требует $\log_2 n$ сравнений. Поэтому вся сортировка требует $n \cdot \log_2 n$ операций, не считая n шагов на построение дерева. Это значительное улучшение по сравнению с прямым методом выбора, который

требует n^2 шагов, но и даже по сравнению с сортировкой Шелла, которая требует $n^{1,2}$ шага.

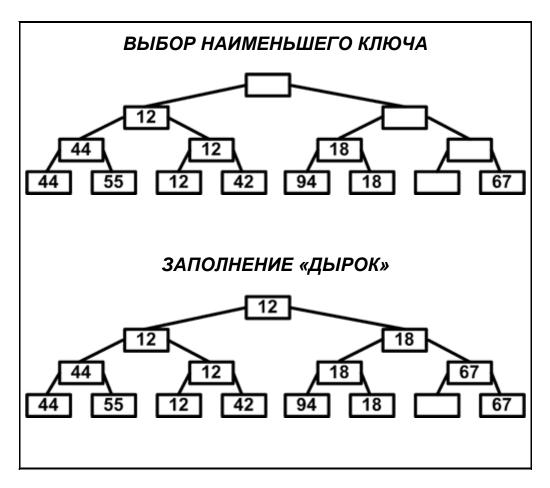


Рис. 3.7

Наша очередная задача — найти способы эффективной организации информации, обрабатываемой на каждом шаге.

Во-первых, желательно избавиться от необходимости в «дырах»; во-вторых, нужно найти способ представить дерево из n элементов в виде массива.

§3.11. Пирамидальная сортировка

Метод пирамидальной сортировки, изобретенный Д. Уилльямсом, является улучшением традиционных сортировок с помощью дерева [1, 3, 9, 10, 13].

Пирамидой называется двоичное дерево такое, что

$$a[i] \le a[2*i];$$

 $a[i] \le a[2*i+1].$

Отсюда следует, что a[1] — минимальный элемент пирамиды.

Сначала расположим исходный массив в пирамиде, а затем пересортируем элементы. Это и есть общая идея пирамидной сортировки (рис. 3.8).

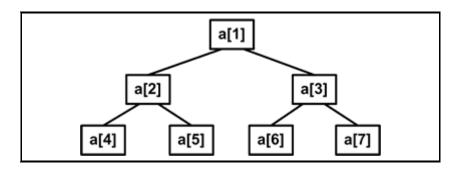


Рис. 3.8

Выполнение алгоритма разбивается на два этапа.

І этап. Построение пирамиды. Определяем правую часть дерева, начиная с n/2+1 (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираете путь через меньший элемент.

Массив для сортировки:

Расположим элементы в виде исходной пирамиды; номер элемента правой части (6/2 + 1) = 4. Просеиваемый элемент имеет номер 3 (рис. 3.9).

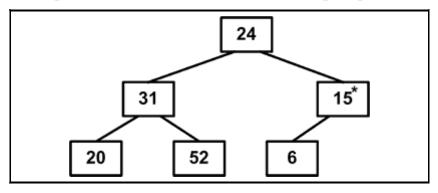


Рис. 3.9

Результат просеивания элемента 15 через пирамиду показан ниже (рис. 3.10).

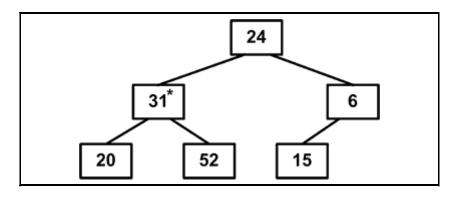


Рис. 3.10

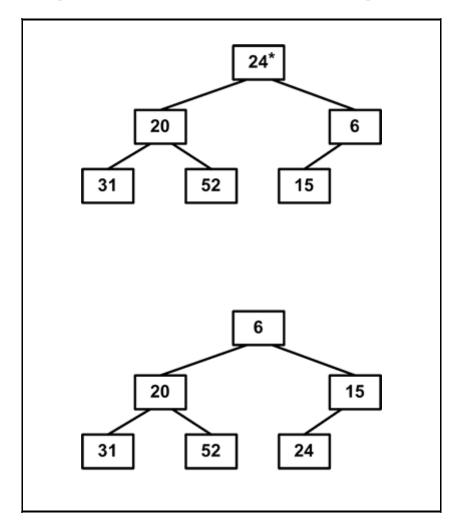


Рис. 3.11

Разумеется, полученный массив еще не упорядочен. Однако, процедура просеивания является основой для пирамидальной сортировки. В итоге просеивания наименьший элемент оказывается на вершине пирамиды. Оформим функцию просеивания элемента с индексом left сквозь часть пирамиды left+1,...,right.

П этап. Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем первый элемент массива (верхний элемент пирамиды, он наименьший) и текущий местами. Текущий элемент (он теперь первый) просеиваем сквозь n–1 элементную пирамиду. Затем берем предпоследний элемент и т.д.

1. Исходной является построенная ранее пирамида. Меняем местами элементы 6 и 24: элемент 6 встал на место (рис. 3.12).

Просеиваем элемент 24 сквозь пирамиду, не трогая элемента 6. В итоге просеивания на вершине окажется 15 — наименьший элемент из оставшейся части массива.

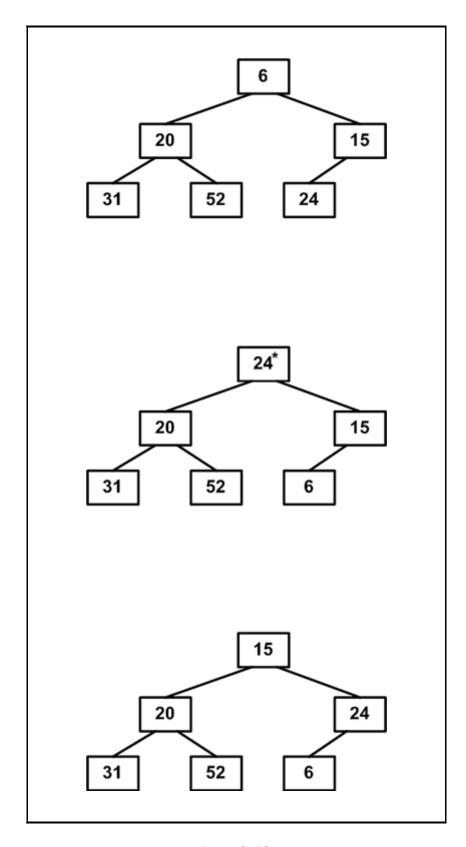


Рис. 3.12

2. Теперь уже два элемента на месте. Продолжим процесс (рис. 3.13).

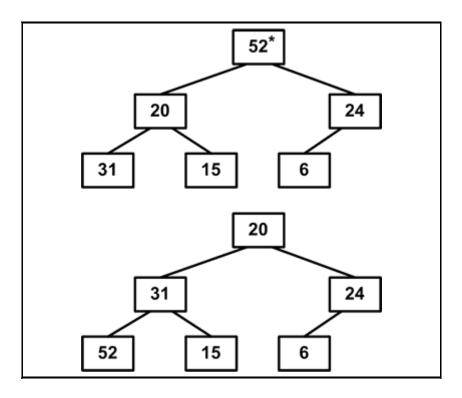


Рис. 3.13

3. Продолжим процесс. В итоге массив будет отсортирован по убыванию (рис. 3.14).

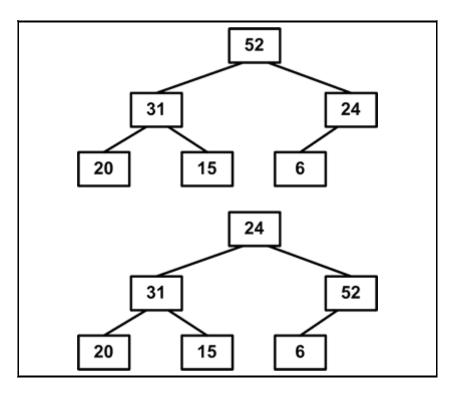


Рис. 3.14

Алгоритм пирамидальной сортировки

```
void heapSort(int numbers[], int array size)
  int i, temp;
  for (i = (array size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array size);
  for (i = array size-1; i >= 1; i--)
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
}
void siftDown(int numbers[], int root, int bottom)
  int done, maxChild, temp;
  done = 0;
  while ((root*2 \le bottom) \&\& (!done))
    if (root*2 == bottom)
     maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;
    if (numbers[root] < numbers[maxChild])</pre>
    {
     temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
```

Анализ алгоритма [3]. Несмотря на некоторую внешнюю сложность, пирамидальная сортировка является одной из самых эффективных. Алгоритм сортировки эффективен для больших n. В худшем случае требуется $n \cdot \log n$ шагов, сдвигающих элементы. Среднее число перемещений примерно равно $(n/2) \cdot \log n$, и отклонения от этого значения относительно невелики.

§3.12. Быстрая сортировка

Рассмотрим усовершенствованный метод сортировки, основанный на принципе обмена. Пузырьковая сортировка является самой неэффективной из всех трех алгоритмов прямой сортировки. Однако усовершенствованный алгоритм является лучшим из известных до сего времени методом сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель Ч. Хоар назвал алгоритм *быстрой сортировкой* [1, 3, 9, 10, 13].

Сортировка основана на том факте, что для достижения наибольшей эффективности желательно производить обмен чисел на больших расстояниях. В массиве выбирается некоторый элемент, называемый разрешающим. Затем он помещается в то место массива, где ему полагается быть после упорядочивания всех элементов. В процессе отыскания подходящего места для разрешающего элемента производятся перестановки элементов так, что слева от него находятся элементы, меньшие разрешающего, а справа — большие (предполагается, что массив сортируется по возрастанию). Тем самым массив разбивается на две части: не отсортированные элементы слева от разрешающего элемента и не отсортированные элементы справа от него. Чтобы отсортировать эти два меньших подмассива, алгоритм вызывает сам себя.

Запишем алгоритм:

<u>если</u> надо сортировать больше одного элемента, то <u>выбрать</u> в массиве разрешающий элемент; переупорядочить массив, помещая элемент на его окончательное место; <u>отсортировать</u> с помощью данного алгоритма элементы слева от разрешающего; <u>отсортировать</u> с помощью данного алгоритма элементы справа от разрешающего.

быстрой Ключевым элементом сортировки является алгоритм переупорядочения. Для его реализации используем указатель left на крайний левый элемент массива. Указатель движется вправо, пока элементы, на которые он показывает, остаются меньше разрешающего. Указатель right поставим на крайний правый элемент списка и движется влево, пока элементы, на которые он показывает, больше разрешающего. Движение указателей остаются останавливается, как только встречаются элементы, порядок расположения которых относительно разрешающего неправильный. Эти числа меняются местами и движение указателей возобновляется. Процесс продолжается до тех пор, пока right не окажется слева от left. Тем самым будет определено правильное место разрешающего элемента.

Рассмотрим сортировку на примере массива:

10, 4, 2, 14, 67, 2, 11, 33, 1, 15.

Пусть крайний левый элемент — разрешающий. Установим left на следующий за ним элемент; right — на последний (рис. 3.15).

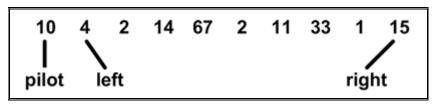


Рис. 3.15

Теперь алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы. Указатель *left* перемещается до тех пор, пока не покажет элемент больше 10; *right* движется, пока не покажет элемент меньше 10. Эти элементы меняются местами (рис. 3.16).

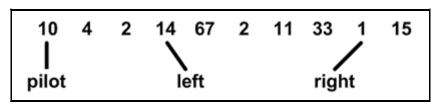


Рис. 3.16. Массив после первого шага сортировки

Левый и правый элементы меняются местами; встречное же движение указателей продолжается (рис. 3.17).

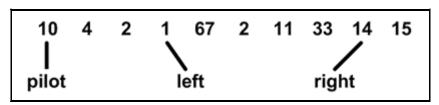


Рис. 3.17

Следующий шаг сортировки элементов показан на рис. 3.18.

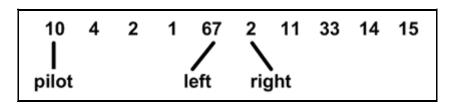


Рис. 3.18

Перестановка элементов показана на рис. 3.19.

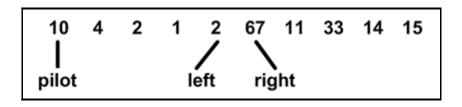


Рис. 3.19

После очередного шага указатели перейдут друг через друга. Это значит, что найдено положение разрешающего элемента (рис. 3.20).

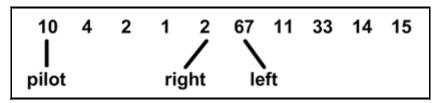


Рис. 3.20

Осуществляется перестановка разрешающего элемента с элементом, на который указывает right (рис. 3.21).

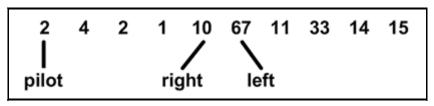


Рис. 3.21

Разрешающий элемент находится в нужном месте: элементы слева от него имеют меньшие значения; справа — большие. Алгоритм рекурсивно вызывается для сортировки элементов слева от разрешающего и справа от него.

Функция быстрой сортировки

```
void quickSort(int numbers[], int array_size)
{
   q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
   int pivot, l_hold, r_hold;
   l_hold = left;
```

```
r hold = right;
pivot = numbers[left];
while (left < right)</pre>
  while ((numbers[right] >= pivot) && (left < right))</pre>
    right--;
  if (left != right)
    numbers[left] = numbers[right];
    left++;
  while ((numbers[left] <= pivot) && (left < right))</pre>
    left++;
  if (left != right)
    numbers[right] = numbers[left];
    right--;
  }
numbers[left] = pivot;
pivot = left;
left = 1 hold;
right = r hold;
if (left < pivot)</pre>
  q sort(numbers, left, pivot-1);
if (right > pivot)
  q sort(numbers, pivot+1, right);
```

Анализ алгоритма [3]. Ожидаемое число обменов равно (n-1/n)/6. Если предположить, что в качестве разрешающего элемента всегда выбирается медиана, то каждое разделение разбивает массив на две равные части, и число проходов, необходимых для сортировки, равно $\log n$. Тогда общее число сравнений составит $n \cdot \log n$, а общее число обменов — $(n/6) \cdot \log n$. Вероятность попадания на медиану составляет 1/n. Однако, если граница выбирается случайным образом, эффективность алгоритма в среднем хуже оптимального варианта лишь в $2 \cdot \ln 2$ раз. Основной недостаток алгоритма — недостаточно высокая производительность при небольших n.

§3.13. Сравнение методов сортировки массивов

Сравним эффективность методов сортировки массивов. Для всех прямых методов сортировки можно дать точные аналитические формулы [3]. Они представлены в табл. 3.5.

Для усовершенствованных методов сортировки нет простых и точных формул. Существенно, однако, что в случае сортировки Шелла вычислительные затраты составляют с $\cdot n^{1,2}$, а для пирамидальной и быстрой сортировок — $c \cdot n \cdot \log n$, где c — соответствующий коэффициент.

Опытным путем были получены следующие результаты [3]:

- 1. Пузырьковая сортировка наихудший метод из всех сравниваемых.
- 2. Быстрая сортировка лучше в 2—3 раза, чем пирамидальная. Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

Сравнение прямых методов сортировки

Таблица 3.5

Вид сортировки	Показатели	Min	Среднее	Max
Прямое	C =	n - 1	$(n^2 - n - 2)/4$	$(n^2 - n)/2 - 1$
включение	M =	2(n-1)	$(n^2 - 9n - 10)/4$	$(n^2 - 3n - 4)/2$
Прямой	C =	$(n^2 - n)/2$	$(n^2-n)/2$	$(n^2 - n)/2$
выбор	M =	3(n-1)	$n \cdot (\ln n + 0.57)$	$n^2/4 + 3(n-1)$
Прямой	C =	$(n^2 - n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
обмен	M =	0	$0,75\cdot(n^2-n)$	$(n^2 - n)$ 1,5

§3.14. Сортировка файлов методом прямого слияния

Алгоритмы сортировки, рассмотренные ранее, неприменимы, если сортируемые данные расположены в файле с последовательным доступом (на диске), который характеризуется тем, что в каждый момент имеется непосредственный доступ к одному и только одному компоненту [1, 3, 9, 10, 13].

Основной применяемый метод — сортировка *слиянием*. Слияние означает объединение двух (или более) последовательностей в одну *упорядоченную последовательность* при помощи циклического выбора элементов, доступных в данный момент. Одна из сортировок на основе слияния называется *сортировкой простым слиянием*.

Метод заключается в следующем [3]:

- 1. Последовательность a разбивается на две половины: b и c.
- 2. Последовательности b и c сливаются при помощи объединения отдельных элементов в упорядоченные пары.
- 3. Полученной последовательности присваивается имя a, и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в упорядоченные четверки.
- 4. Предыдущие шаги повторяются: четверки сливаются в восьмерки и так далее, пока не будет упорядочена вся последовательность.

В качестве примера возьмем следующую последовательность

Первый шаг: разбиение дает две последовательности

Слияние в упорядоченные пары дает последовательность

Новое разбиение пополам и слияние упорядоченных пар дает последовательность

Третье разбиение и слияние приводят к нужному результату

Операция, которая однократно обрабатывает множество данных, называется фазой, а наименьший подпроцесс, который, повторяясь, образует процесс сортировки, называется *проходом* или *этапом*. В примере сортировка производится за три прохода, каждый проход состоит из фазы разбиения и фазы слияния. Для выполнения требуются три магнитные ленты (три файла), поэтому процесс называется *трехленточным слиянием*.

Собственно говоря, фазы разбиения не относятся к сортировке; их можно удалить, объединив фазы разбиения и слияния. Вместо того, чтобы сливать элементы в одну последовательность, результат слияния сразу распределяют на два файла, которые на следующем проходе будут входными. Этот метод называется однофазным или сбалансированным слиянием. Он имеет явные преимущества, так как требует вдвое меньше операций переписи, но это достигается ценой использования четвертой ленты.

Вместо двух файлов можно использовать один, если просматривать его с двух концов. Таким образом общий вид объединенной фазы слияния-разбиения имеет вид, показанный на рис. 3.22. Направление пересылки сливаемых элементов меняется после каждой упорядоченной пары на первом проходе, после каждой четверки на втором и так далее. Таким образом равномерно заполняются две выходные последовательности, представленные двумя концами выходного

массива. После каждого прохода массивы «меняются ролями», выходной становится входным и наоборот.

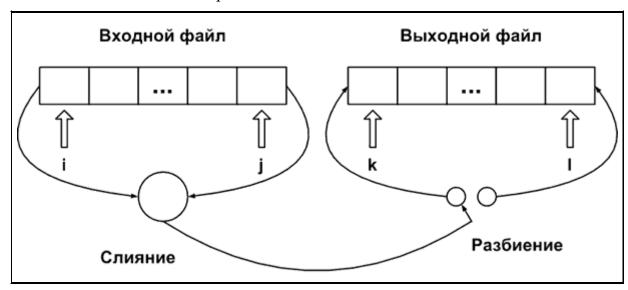


Рис. 3.22. Схема сортировки прямым слиянием для двух файлов

Программу можно упростить, объединив два массива в один массив двойной длины. Итак, массив данных представим следующим образом:

```
a = (int^*)malloc(2 * n * sizeof(int));
```

Пусть l, j — фиксируют два исходных элемента; k, l — два выходных. Исходные данные — это элементы $a_1, ..., a_n$. Для указания направления пересылки введем логическую переменную up. Если up = 1, то в текущем проходе компоненты $a_i, ..., a_n$ движутся на место $a_{n+1}, ..., a_{2n}$, если же up = 0, то $a_{n+1}, ..., a_{2n}$ пересылаются в $a_i, ..., a_n$. Между последовательными проходами значение up изменяется на противоположное. Пусть p — длина сливаемых последовательностей. Начальное значение p равно 1, и перед каждым последующим проходом она удваивается. Для простоты мы предполагаем, что всегда n равно степени двойки.

Первая версия программы сортировки с помощью простого слияния имеет следующий вид

```
int i, j, k, l;
int up, p;
up=1;
p=1;
/* индексация индексов */
do
{
   if (up==1)
   {
    i=1;
```

```
j=n;
    k=n+1;
    l=2*n;
}
    else
    {
        k=1;
        l=n;
        i=n+1;
        j=2*n;
}
/* слияние р-наборов из i- и j-
входов в k- и l-выходы */
        up=-up;
        p=2*p;
}
while (p==n);
```

Уточним действия, описанные на естественном языке. Ясно, что процесс слияния n элементов сам представляет собой последовательность слияний последовательностей, т. е. p-наборов. После каждого такого частичного слияния выход переключается с нижнего на верхний конец выходного массива и наоборот, что гарантирует одинаковое распределение в обоих направлениях. Если сливаемые элементы направляются в левый конец выходного массива, то направление задается индексом k, и после пересылки очередного элемента он увеличивается на единицу. Если же элементы направляются в правый конец, то направление задается индексом l и он каждый раз уменьшается. Для упрощения фактического оператора слияния будем считать, что направление всегда задается индексом k, но после слияния p-набора будем менять местами значения k и l, приращение же всегда обозначается через h, имеющее значение либо 1, либо -1. Высказанные соображения приводят к такому уточнению программы.

```
h=1;

m=n;

/*m — число сливаемых элементов*/

do

{

    q=p;

    r=p;

    m=m-2*p;
```

```
/*слияние q-элементов из i-входа с r элементами из j-входа; индекс выхода k, затем k увеличиваем на h*/

h=-h;
/*обмен значениями k и l*/
} while (m==0);
```

Дальнейшее уточнение ведет уже к формулированию самого оператора слияния. При этом следует учитывать, что остаток подпоследовательности, оставшийся после слияния непустой подпоследовательности, добавляется простым копированием к выходной последовательности.

```
while ( (q!=0) && (r!=0))

if (a[i]<a[j])

{
/* элемент из i-входа пересылается
на k-выход; i и k продвигаются*/
  q=q-1;
  }
  else
  {
/* элемент из j-входа посылается на
k-выход; j и k продвигаются*/
  r=r-1;
  }
}
/*копирование хвоста i-массива;
копирование хвоста j-массива;*/
```

Если теперь уточнить операции копирования остатков, то программа станет совершенно точной. Однако, прежде чем сделать это, мы хотим избавиться от ограничения на n: пока оно должно быть степенью двойки. На какие части алгоритма влияло это условие? Легко убедить себя, что наилучший способ справиться с более общей ситуацией продолжать действовать, насколько это возможно, по-старому. В данной ситуации это означает, что мы будем продолжать слияние p-наборов до тех пор, пока не останутся последовательности размером менее p. Этот процесс затрагивает только ту

часть, где определяются значения q и r — длины сливаемых последовательностей. Вместо трех операторов

$$q = p;$$

$$r = p;$$

$$m = m - 2*p;$$

используется следующая конструкция:

```
if(m >= p)
q = p;
else
q = m;
m = m - q;
if(m >= p)
r = p;
else
r = m;
m = m - r;
```

Здесь m — общее число элементов в двух входных последовательностях. Кроме того, для гарантии окончания программы условие p=0, управляющее внешним циклом, заменяется на p >= n. Проделав все эти модификации, мы получаем следующую программу.

```
int i, j, k, l, t;
int h, m, p, q, r;
int up;
/*У массива а индексы 1, ...,2*n.*/
up=1;
p=1;
do
{
   h=1;
   m=n;
   if (up==1)
      i=1;
      j=n;
      k=n+1;
      1=2*n;
   }
   else
      k=1;
```

```
l=n;
      i=n+1;
      j=2*n;
   }
   do
   {
/* слияние серий из і и ј в k-
выход*/
/* q — длина серии из i, r — длина
серии из ј */
      if (m>=p)
         q=p;
      else
          q=m;
      m=m-q;
      if (1>=p)
         r=p;
      else
          r=m;
      m=m-r;
      while (q!=0 \&\& r!=0)
       /* слияние */
           if (a[i] < a[j])</pre>
               a[k]=a[i];
               k=k+h;
               i=i+1;
               q=q-1;
           }
           else
               a[k]=a[j];
               k=k+h;
               j=j-1;
               r=r-1;
      }
/* копирование остатка серии из ј*/
```

```
while (r!=0)
           a[k]=a[j];
           k=k+h;
           j=j-1;
           r=r-1;
       }
/* копирование остатка серии из і*/
      while (q!=0)
           a[k]=a[i];
           k=k+h;
           i=i+1;
           q = q - 1;
       }
      h=-h;
      t=k;
      k=1;
      l=t;
   while (m==0);
   up = -up;
   p=2*p;
while (p>=n);
if (up==0)
   for(i=1; i<n; i++)
      a[i]=a[i+n];
```

Анализ алгоритма. Поскольку на каждом проходе p удваивается и сортировка заканчивается при p >= n, то всего требуется $\log n$ проходов. На каждом проходе по определению копируются по одному разу все n элементов. Поэтому общее число пересылок

$$M = n \cdot \log n$$
.

Число сравнений ключей C даже меньше M, поскольку при копировании остатков никаких сравнений не производится. Однако, поскольку сортировки слиянием обычно употребляются в ситуациях, где приходится пользоваться внешними запоминающими устройствами, то затраты на операции пересылки на несколько порядков превышают затраты на сравнения. Поэтому детальный анализ числа сравнений особого практического интереса не представляет.

Алгоритм сортировки слиянием выдерживает сравнение даже с усовершенствованными методами сортировки. Однако, хотя здесь относительно высоки затраты на работу с индексами, решающее значение играет необходимость работать с памятью размером 2*n*. Поэтому сортировка слиянием для массивов, т. е. для данных, размещаемых в оперативной памяти, используется редко.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

- 1. Ахо, А.В. Структуры данных и алгоритмы / А.В. Ахо, Д.Д. Ульман, Д.Э. Хопкрофт; пер. с англ. и ред. А.А. Минько. М.:Вильямс, 2000. 384 с.
- 2. Болски, М.И. Язык программирования Си: справочник / М.И. Болски; пер. с англ. С.В. Денисенко. М.: Радио и связь, 1988. 96 с.
- 3. Вирт, Н. Алгоритмы и структуры данных/Н. Вирт; пер. с англ. М.: Мир, 1989. 360 с.
- 4. Бочков, С.О. Язык программирования Си для персонального компьютера / С.О. Бочков, Д.М. Субботин. М.: Радио и связь, 1990. 384 с.
- 5. Громов, Ю.Ю. Программирование на языке Си: учебное пособие / Ю.Ю. Громов, С.И. Татаренко. Тамбов, 1995. 169 с.
- 6. Давыдов, В.Г. Программирование и основы алгоритмизации: учебное пособие / В.Г. Давыдов. М.: Высш. шк., 2003. 447 с.
- 7. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи; пер. с англ. СПб.: Нев. диалект, 2001 —351 с.
- 8. Кнут, Д.Э. Искусство программирования. В 3 т. Т 1. Основные алгоритмы / Д.Э. Кнут; пер. с англ. 3-е изд. М.: Издательский дом "Вильямс", 2000. 720 с.
- 9. Кнут, Д.Э. Искусство программирования. В 3 т. Т 3: Сортировка и поиск / Д.Э. Кнут; пер. с англ. 2-е изд. М.: Издательский дом "Вильямс", 2000. 832 с.
- 10. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест; пер. с англ. А. Шеня. М.: МЦНМО, 2001. 955 с.
- 11. Лэнгсам, Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Оренстайн, А. Тененбаум; пер. с англ. Л. П. Викторова и др. М.: Мир, 1989. 586 с.
- 12. Подбельский, В.В. Программирование на языке Си: Учеб. Пособие / В.В. Подбельский, С.С. Фомин. 2-е изд., доп. М.: Финансы и статистика, 2002. -600 с.
- 13. Седжвик, Р. Фундаментальные алгоритмы на С++: В 2 т. Т. 1: Анализ и структуры данных. Сортировка. Поиск / Р. Седжвик; пер. с англ. М.; СПб; Киев: ДиаСофт, 2001. 687 с.
- 14. Мартин, Д. Язык Си. Руководство для начинающих / Д. Мартин, С. Прата, М. Уэйт; пер. с англ. М.: Мир, 1988. 512 с.

ОГЛАВЛЕНИЕ

Предисловие	3
Глава 1. Язык программирования Си	
§1.1. Введение в язык Си	4
§1.1. Структура программы	5
§1.2. Объекты языка Си и их типы	7
§1.3. Простые объекты	7
§1.5. Операции	11
§1.6. Ввод и вывод информации	13
§1.7. Операторы	15
§1.8. Функции	23
§1.9. Прототипы функций	28
§1.10. Препроцессор	30
§1.11. Математические функции	32
§1.12. Специальные операции	33
§1.13. Глобальные и локальные объекты	35
§1.14. Модификация объектов	38
§1.15. Указатели	40
§1.16. Модели памяти	43
§1.17. Массивы	44
§1.18. Передача массива в функцию	49
§1.19. Многомерные массивы	50
§1.20. Динамическое распределение памяти	52
§1.21. Динамическое распределение памяти под массивы	54
§1.21. Массивы указателей	57
§1.22. Структуры	60
§1.23. Объединения	63
§1.25. Битовые поля	65
§1.26. Указатели и структуры	66
§1.27. Классификация функций ввода-вывода	70
§1.28. Функции ввода-вывода высокого уровня	71
§1.29. Работа с файлами данных	75
§1.30. Функции обработки строк	80
§1.31. Работа со строками	81
§1.31. Логический тип данных	93
§1.32. Программная реализация стека	95
Глава 2. Структуры данных	
§2.1. Введение в структуры данных	105
§2.1. Стек	105
§2.2. Однонаправленные связанные списки	113
§2.3. Однонаправленные циклические списки	116
§2.5. Двунаправленные связанные списки	117
§2.6. Очереди	125

§2.7. Бинарные деревья	131
Глава 3. Сортировка и поиск	
§3.1. Введение в поиск	139
§3.1. Последовательный поиск	139
§3.2. Поиск в упорядоченной таблице	140
§3.3. Хеширование таблиц	142
§3.5. Введение в сортировку	144
§3.6. Сортировка с помощью прямого включения	145
§3.7. Сортировка с помощью прямого выбора	147
§3.8. Сортировка с помощью прямого обмена	149
§3.9. Сортировка включениями с убывающим приращением	151
§3.10. Сортировка с помощью дерева	153
§3.11. Пирамидальная сортировка	154
§3.11. Быстрая сортировка	160
§3.12. Сравнение методов сортировки массивов	163
§3.13. Сортировка файлов методом прямого слияния	164
Библиографический список.	173

Электронное издание

Сергей Тимурович Касюк
 КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Конспект лекций

Издательский центр Южно-Уральского государственного университета

Подписано в печать 09.04.2010. Формат 60×84 1/16. Усл. печ. л. 10,23. Уч.-изд. л. 10,38. Заказ 109.