

```
#include <iostream>
#include <string>
```

Практикум по программированию на C++

.....
push_front

- *Практический курс от простейших задач до двоичных файлов и наследования*
- *Методика изложения материала через прикладные задачи*
- *Навыки чтения и анализа готовых программ*
- *200 стандартных программных решений и более 300 тестов*

sort

A.assign(

```
cout << endl;
Rational operator+(int b)
Divide(w, 10, 2)
```

**ПРАКТИКУМ
ПО ПРОГРАММИРОВАНИЮ
НА C++**

Е. Л. Романов

**Практикум
по программированию
на C++**

Санкт-Петербург

«БХВ-Петербург»

2004

УДК 519.682(075.8)
ББК 32.973.26-018.1я73
Р69

Романов Е. Л.

Р69 Практикум по программированию на С++: Уч. пособие. СПб: БХВ-Петербург; Новосибирск: Изд-во НГТУ, 2004. — 432 с.

ISBN 5-94157-553-X (БХВ-Петербург)

ISBN 5-7782-0478-7 (НГТУ)

Практический курс программирования на Си/Си++ для начинающих. Содержит более 200 стандартных программных решений и более 300 тестовых заданий по 22 темам: от простейших вычислительных задач до двоичных файлов и наследования. Отдельная глава посвящена навыкам «чтения» и анализа готовых программ, «словарному запасу» программиста — стандартным программным контекстам и их использованию в традиционной технологии структурного программирования.

Рекомендуется студентам направления «Информатика и вычислительная техника», а также всем самостоятельно изучающим язык Си и технологию программирования на нем. Книга будет полезна при постановке 2-3-семестрового курса программирования, включающего лабораторный практикум.

УДК 519.682(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Редактор	<i>Н. А. Лукашова</i>
Технический редактор	<i>Г. Е. Телятникова</i>
Художник-дизайнер	<i>А. В. Волощина</i>
Компьютерная верстка	<i>Н. В. Беловой</i>

Рецензенты:

*В.И. Хабаров, д-р техн. наук, проф. кафедры информационных технологий Сибирского государственного университета путей сообщения,
директор Института информационных технологий на транспорте*

*Б.М. Глинский, д-р техн. наук, проф., заведующий кафедрой
вычислительных систем Новосибирского государственного университета*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 18.08.04.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 34,83.

Тираж 3000 экз. Заказ № 3506

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-553-X (БХВ-Петербург)
ISBN 5-7782-0478-7 (НГТУ)

© Романов Е. Л., 2003
© Новосибирский государственный технический университет, 2003
© ООО "БХВ-Петербург", 2004

ПРЕДИСЛОВИЕ

Спят подружки вредные безмятежным сном.
Снятся мышкам хлебные крошки под столом,
Буратином – досточки, кошкам – караси,
Всем собакам – косточки, программистам – Си.

*Е. Романов. Колыбельная.
«Болдинская осень». 1996*

Для начала – чем не является эта книга. Это – не справочник по языку Си или системе программирования на нем, это – не учебник, начинающийся с азов, и, надеюсь, не просто набор примеров и вопросов к ним. Эта книга имеет отношение не столько к языку, сколько к практике программирования на нем и к практике программирования вообще.

Первую часть книги можно было бы назвать «программирование здравого смысла». Она содержит в концентрированном виде то, чего не хватает начинающему программисту и на чем обычно не акцентируют внимание ни учебники, ни, тем более, справочники. Это – «джентльменский набор» программных конструкций, которые позволяют программисту свободно выражать свои мысли. Это – изложение основ чтения (анализа и понимания) чужих программ, что является, по убеждению автора, обязательным этапом перед написанием собственных. Это – программные решения, которые опираются на формальную логику, здравый смысл, образные аналогии и которые составляют значительную часть любой типовой, в меру оригинальной, программы. Это – обсуждение самого процесса проектирования программы.

Каждая тема, а их более 20, содержит сжатое изложение приемов программирования, примеры стандартных программных ре-

шений, контрольные вопросы, задания к лабораторному практикуму (не менее 15), тестовые задания в виде фрагментов программ и функций (10–20). Темы сгруппированы в три раздела в порядке возрастания сложности: «программист начинающий» (арифметика, сортировка, работа со строками, типы данных, указатели), «программист системный» (структуры данных, массивы указателей, списки, деревья, рекурсия, файлы, управление памятью) и «программист объектно-ориентированный» (классы и объекты, переопределение операций, наследование и полиморфизм).

Объем книги соответствует двух-трехсеместровому курсу программирования, включающему лабораторный практикум. Ее можно использовать и для организации тестирования и проверки уровня знаний по языку. И, наконец, она может быть рекомендована тем, кто делает первые шаги и испытывает трудности в освоении науки, искусства, ремесла (ненужное зачеркнуть) программирования.

Автор выражает свою признательность студентам факультета автоматизации и вычислительной техники Новосибирского государственного технического университета, безропотно сносившим обкатку и усовершенствование представленного здесь материала.

Отзывы и замечания по содержанию книги можно направлять непосредственно автору по E-mail: romanow@vt.cs.nstu.ru. Дополнительные учебно-методические материалы и исходные тексты приведенных в книге примеров программ можно найти на сайте кафедры ВТ НГТУ <http://ermak.cs.nstu.ru/cprog>.



1. АНАЛИЗ И ПРОЕКТИРОВАНИЕ ПРОГРАММ

1.1. ПРЕЖДЕ ЧЕМ НАЧАТЬ

Разруха сидит не в клозетах, а в головах.

М. Булгаков. Собачье сердце

Тот, кто считает, что процесс программирования заключается во вводе в компьютер различных команд и выражений, написанных на языке программирования, глубоко ошибается. Программа, на самом деле, пишется в голове и переносится по частям в компьютер, поскольку голова не самый удобный инструмент для выполнения программы.

Здесь я хотел бы сразу же снять некоторые заблуждения, которые возникают у начинающих.

Первое. Компьютер – это инструмент программирования, никакие достоинства инструмента не заменят навыков работы с ним. И уж тем более нельзя объяснять низкое качество производимого продукта только несовершенством инструмента. В устах шофера это звучало бы так: сейчас я плохо маневрирую на «Жигулях», а вот дайте мне «Мерседес», уж тогда я «зарулю».

Второе. Компьютер никогда не будет «думать за вас». Если вы работаете с готовой программой, тогда может сложиться такая иллюзия. Если же вы разрабатываете свою, следить за ее работой должны именно вы. То есть ее нужно параллельно с компьютером «прокручивать» в собственной голове. Процесс отладки в том и состоит, что вы сами отслеживаете разницу между работой той идеальной программы, которая пока находится у вас в голове, и той реальной, имеющей ошибки, которая в данный момент «крутится» в компьютере.

Третье. В любом виде деятельности имеется своя технология – это совокупность знаний, навыков, инструментов, правил работы. В программировании также есть своя технология. Ее нужно изучить и приспособить под свой образ мышления.

Программирование тем и отличается от всех других видов деятельности, что представляет собой в концентрированном виде формально-логический образ мышления. Как известно, человек воспринимает мир «двумя полушариями» – образно-эмоционально и формально-логически. Компьютер содержит в себе вторую крайность – он в состоянии воспроизвести с большой скоростью заданный набор формально-логических действий, именуемых иначе программой. В принципе, человек может делать то же самое, но в ограниченных масштабах. Как было метко сказано: *«Компьютер – это идиот, но идиот быстродействующий»*.

Любой набор формальных действий всегда дает определенный результат, который уже является внешней стороной процесса. Какого-либо «смысла» для самой формальной системы (программы) этот результат не имеет. То есть компьютер в принципе не ведает, что творит. Программист же, в отличие от компьютера, должен знать, что он делает. Он отталкивается от цели, результата, для которых он старается создать соответствующую им программу, используя всю мощь своего разума и интеллекта. А здесь нельзя обойтись без образного мышления, интуиции и, если хотите, вдохновения.

В своей работе программист руководствуется образным представлением программы, он видит ее «целиком» в процессе выполнения и лишь затем разделяет ее на отдельные элементы, которые являются в дальнейшем частями алгоритмов и структур данных. В этом коренное отличие программиста от компьютера, который не в состоянии сам писать программы.

1.2. КАК РАБОТАЕТ ПРОГРАММА

Трудность начального этапа программирования в том и заключается, что программист «видит» за текстом программы нечто большее, чем начинающий, и даже нечто большее, чем сам компьютер. Об этом несколько сумбурно было сказано выше. То есть программист «видит» весь процесс выполнения данной конструкции языка, а также результат ее выполнения, который и составляет «смысл» конструкции. Начинающий же «видит» кучу взаимосвязанных переменных, операций и операторов. Кроме того, слож-

ность заключается еще и в том, что конструкции языка вкладываются друг в друга, а не пристыковываются подобно кирпичам в стене.

Поэтому следует начинать с обратного: с приобретения навыков «чтения» и понимания смысла программ и их отдельно взятых конструкций, фрагментов, контекстов.

О РАЗНЫХ МЕТОДАХ УБЕЖДЕНИЯ

Назначение любой программы – давать определенный результат для любых входных значений. Результат же – это набор значений, удовлетворяющих некоторым условиям, или набор, обладающий некоторыми свойствами. Если посмотреть на программу с этой точки зрения, то окажется, что она имеет много общего с математической теоремой. Действительно, теорема утверждает, что некоторое свойство имеет место на множестве элементов (например, теорема Пифагора устанавливает соотношение для гипотенузы и катетов всех прямоугольных треугольников). Программа обладает тем же самым свойством: для различных вариантов входных данных она дает результат, удовлетворяющий определенным условиям. Поэтому анализ программы – это не что иное, как формулировка и доказательство теоремы о том, какой результат она дает.

Анализ программы – формулировка теоремы о том, какой результат она дает для всех возможных значений входных переменных.

Убедиться, что теорема верна, можно различными способами. (Обратите внимание – *убедиться, но не доказать*). Точно так же можно убедиться, что программа дает тот или иной результат:

- выполнить программу в компьютере или проследить ее выполнение на конкретных входных данных «на бумаге» (анализ методом **единичных проб**, или **«исторический» анализ**);

- разбить программу на фрагменты с известным «смыслом» и попробовать соединить результаты их выполнения в единое целое (анализ на **уровне неформальной логики и «здравого смысла»**);

- формально доказать с использованием логических и математических методов (например, метода математической индукции), что фрагмент дает заданный результат для любых значений входных переменных (**формальный анализ**).

Те же самые методы можно использовать, если результат и «смысл» программы не известны. Тогда при помощи единичных проб и разбиения программы на фрагменты с уже известным «смыслом» можно догадаться, каков будет результат. Такой же процесс, но в обратном направлении, имеет место при разработке программы. Можно попытаться разбить конечный результат на ряд промежуточных, для которых уже имеются известные фрагменты.

«ИСТОРИЧЕСКИЙ» АНАЛИЗ

Первое, что приходит в голову, когда требуется определить, что делает программа, это понаблюдать за процессом ее выполнения и догадаться, что она делает. Для этого даже не обязательно иметь под рукой компьютер: можно просто составить на листе бумаги таблицу, в которую записать значения переменных в программе после каждого шага ее выполнения: отдельного оператора, тела цикла.

```
int A[10]={3,7,2,4,9,11,4,3,6,3};
int k,i,s;
for (i=0,s=A[0]; i<10; i++)
    if (A[i]>s) s=A[i];
```

Проследим за выполнением программы, записывая значения переменных до и после выполнения тела цикла.

i	A[i]	s до if	s после if	Сравнение
0	3	3	3	Ложь
1	7	3	7	Истина
2	2	7	7	Ложь
3	4	7	7	Ложь
4	9	7	9	Истина
5	11	9	11	Истина
6	4	11	11	Ложь
7	3	11	11	Ложь
8	6	11	11	Ложь
9	3	11	11	Ложь
10	Выход		11	

Закономерность видна сразу: значение s все время возрастает, причем в переменную записываются значения элементов массива. Легко догадаться, что в результате она будет принимать максимальное. Чтобы окончательно убедиться в этом, необходимо помнить содержимое массива и проследить за выполнением программы.

Аналогичные действия можно произвести, используя средства отладки системы программирования: они позволяют выполнять программу «по шагам» в режиме трассировки и следить при этом за значениями интересующих нас переменных.

Естественные ограничения «исторического» подхода состоят в том, что он применим для достаточно простых программ и требует очень развитой интуиции, чтобы уловить зависимость, которая присутствует в обрабатываемых данных и определяет результат. Реально же интуитивное видение результата программы – это следствие опыта программирования, результат тренировки. Кроме того, многообразие входных данных, с которыми может работать программа, не гарантирует того, что вы сразу заметите закономерность.

Отсюда следует, что «исторический» анализ программы является вспомогательным средством. Сначала необходим логический анализ программы и выделение стандартных общепринятых фрагментов (стандартных программных контекстов), результат работы каждого из которых известен. И только затем, для понимания тонкостей работы программы, связанных с взаимодействием этих фрагментов, можно применять «исторический» анализ. Что же касается входных данных, то они должны быть выбраны на этапе анализа как можно более простыми, чтобы легко можно было уловить закономерность их изменения.

ЛОГИЧЕСКИЙ АНАЛИЗ: СТАНДАРТНЫЕ ПРОГРАММНЫЕ КОНТЕКСТЫ

Как это ни странно, программист при анализе программы не мыслит категориями языка: переменными или операторами, как говорящий не задумывается над отдельными словами, а использует целые фразы разговорного языка. Точно так же, любая в меру оригинальная программа на 70–80 % состоит из стандартных решений, которые реализуются соответствующими фрагментами – стандартными программными контекстами. Смысл их заранее известен программисту и не подвергается сомнению, поскольку находится для него на уровне очевидности и здравого смысла. Стандартные программные контексты обладают свойством **инвариантности**: они дают один и тот же результат, будучи помещенными в другие конструкции языка, причем даже не в виде единого целого, а по частям. Более того, их общий смысл не меняется, если меняется синтаксис входящих в них элементов. В программе, находящей

индекс минимального элемента массива, исключая отрицательные, вы без труда заметите контекст предыдущего примера.

```
int A[10]={3,7,2,4,9,11,4,3,6,3};
int k,i,s;
for (i=0,k=-1; i<10; i++){
    if (A[i]<0) continue;
    if (k==-1 || A[i]<A[k]) k=i;
}
```

Он состоит в том, что обязательно должен быть цикл по множеству элементов, сравнение текущего с теми данными, которые характеризуют минимум, и присваивание этому минимуму характеристик текущего элемента, если сравнение прошло успешно (в пользу очередного).

ФОРМАЛЬНЫЙ АНАЛИЗ: МЕТОД МАТЕМАТИЧЕСКОЙ ИНДУКЦИИ

Формальный анализ программы базируется на специальных разделах дискретной математики. Здесь мы упомянем единственный метод, который полезен не столько при доказательстве правильности программ, сколько как теоретическое подтверждение некоторых принципов разработки программ.

Метод математической индукции является средством доказательства справедливости утверждения на любой (даже бесконечной) последовательности шагов: если утверждение справедливо на начальном шаге, а из справедливости утверждения на произвольном (i) шаге доказывается его справедливость на следующем ($i+1$), то такое утверждение справедливо всегда.

Метод математической индукции хорош прежде всего для циклических и рекурсивных программ. Во-первых, как дополнительный аргумент в доказательстве того, что фрагмент программы делает именно то, что должен делать. Типичный пример – нахождение максимального элемента массива.

```
for (s=0,i=0; i<10; i++) if (A[i]>s) s=A[i];
```

То, что фрагмент действительно делает, что от него требуется, мы уже наблюдали в «историческом» подходе. Формальная логика и «здравый смысл» тоже могут быть использованы как дополнительные способы убеждения. Фрагмент `if (A[i]>s) s=A[i]` читается буквально так: если очередной элемент массива больше, чем то, что нужно нам, мы его запоминаем, иначе оставляю старое, осуществляя обычный принцип выбора «большого из двух зол». Фор-

мальное доказательство звучит так: если на очередном шаге переменная s содержит максимальное значение для элементов $A[0]...A[i-1]$, полученное на предыдущих шагах, то после выполнения $\text{if } (A[i]>s) \ s=A[i]$ она будет содержать такой же максимум, но уже с учетом текущего шага. То есть из справедливости утверждения на текущем шаге вытекает справедливость его же на следующем.

Но главное, что аналогичный подход должен использоваться и при проектировании циклов: нужно начинать обдумывать циклическую программу не с первого шага цикла, а с произвольного, и постараться сформулировать для него условие, которое сохраняется от предыдущего шага к последующему (**инвариант цикла**, см. раздел 1.7). Тогда в соответствии с принципом индукции этот цикл будет давать верный результат при любом количестве шагов.

ОТЛАДКА: ДВЕ ПРОГРАММЫ – В КОМПЬЮТЕРЕ И В ГОЛОВЕ

Большинство начинающих искренне считают, что их программа должна работать уже потому, что она написана. Однако отладка программы – еще более трудное дело, чем ее написание. Это только так кажется, что программист в состоянии контролировать разработанную им программу. На самом деле число возможных вариантов ее поведения, обусловленных как логикой программы, так и ошибками, разбросанными там и сям по ее тексту, чрезвычайно велико. Отсюда следует, что к собственной программе следует относиться скорее как к противнику в шахматной игре: фигуры расставлены, правила известны, число возможных ходов не поддается логическому анализу.

Основной принцип отладки: работающая программа на самом деле находится в голове программиста. Реальная программа в компьютере – лишь грубое к ней приближение. Программист должен отследить, когда между ними возникает расхождение – в этом месте и находится очередная ошибка. Для этой цели служат средства отладки. Они позволяют наблюдать поведение программы: значения выбранных переменных при пошаговом ее выполнении, при выполнении ее до заданного места (точки остановки) либо до момента выполнения заданных условий.

В отладке программы, как и в ее написании, существует своя технология, сходная со структурным программированием:

– нельзя отлаживать все сразу. На каждом этапе проверяется отдельный фрагмент, для чего программа должна проходить только по уже протестированным частям, «внушающим доверие»;

– отладку программы нужно начинать на простых тестовых данных, обеспечивающих прохождение программы по уже отлаженным фрагментам. Входные данные для отладки лучше не вводить самому, а задавать в виде статических последовательностей в массивах или в файлах;

– если поведение программы не поддается анализу и определить местонахождение ошибки невозможно, необходимо произвести «следственный эксперимент»: проследить выполнение программы на различных комбинациях входных данных, набрать статистику и уже на ее основе строить догадки и выдвигать гипотезы, которые в свою очередь нужно проверять на новых данных;

– модульному программированию соответствует модульное тестирование. Отдельные модули (функции, процедуры) следует сначала вызывать из головной программы (**main**) и отлаживать на тестовых данных, а уже затем использовать по назначению. Вместо ненаписанных модулей можно использовать «заглушки», дающие фиксированный результат;

– нисходящему программированию соответствует нисходящее тестирование. Внутренние части программы аналогично могут быть заменены «заглушками», позволяющими частично отладить уже написанные внешние части программы.

Ошибки лучше всего различать не по сложности их обнаружения и не по вызываемым ими последствиям, а по затратам на их исправление:

– мелкие ошибки типа «опечаток», которые обусловлены просто недостаточным вниманием программиста. К таковым относятся неправильные ограничения цикла (плюс-минус один шаг), использование не тех индексов или указателей, одной переменной одновременно в двух «смыслах» и т.п.;

– локальные ошибки логики программы, состоящие в пропуске одного из возможных вариантов ее работы или сочетания входных данных;

– грубые просчеты, связанные в неправильным образным представлением того, что и как должна делать программа.

И последнее. Народная мудрость гласит, что любая программа в любой момент содержит как минимум одну ошибку.

1.3. СТАНДАРТНЫЕ ПРОГРАММНЫЕ КОНТЕКСТЫ

Когда чужой мои читает письма,
заглядывая мне через плечо...

В. Высоцкий. Я не люблю

ЗАЧЕМ ЧИТАТЬ ЧУЖИЕ ПРОГРАММЫ?

Мое глубокое убеждение: изучение программирования нужно начинать с чтения чужих программ. Риторический вопрос – зачем? Естественно, не для того, чтобы убедиться, какие это умные люди – другие программисты. И, естественно, читать надо не какие-то произвольные программы, а нарочно для этого подобранные.

Обычный разговорный язык не так богат, как кажется. То же самое касается программ. В них довольно большой процент составляют «стандартные фразы», а многообразие программ на самом деле заключается в комбинировании таких фраз. Действительно оригинальные алгоритмы в практике обычного программиста встречаются довольно редко. Обычно он занят рутинной – конструированием тривиальных алгоритмов из стандартных заготовок.

Но к процессу самого проектирования обратимся позднее. Пока предстоит освоить «джентльменский набор» фрагментов программ. Тут необходимо сделать два замечания. Во-первых, в отличие от обычного текста, синтаксические фрагменты программы не только следуют друг за другом, но и *вкладываются друг в друга*. Поэтому «хвост» фрагмента может отстоять от «головы» на достаточно большом расстоянии. Во-вторых, определяющим является некий логический каркас фрагмента, а составные его части могут быть произвольными. Например, поиск максимального значения элемента по-разному выглядит в таких структурах данных, как массив, массив указателей, список и дерево, но имеет неизменную, **инвариантную** ко всем структурам данных, часть.

```
int      F(int A[],int n){                // Массив
in i,s;
for (i=0,s=A[0]; i<n; i++)
    if (A[i]>s) s=A[i];
return s; }
```

```
int      F(int *A[]){                     // Массив указателей
int i,k;
for (i=k=0; A[i]!=NULL; i++)
    if (*A[i] > *A[k]) k=i;
return *A[k];}
```

```

int F(list *ph) { list *p,*q;                // Список
for (p=q=ph; p!=NULL; p=p->next)
    if (p->val > q->val) p=q;
return q->val; }

int F(xxx *q){                               // Дерево
int i,n,m; if (q==NULL) return 0;
for (n=q->v,i=0; i<4; i++)
    if ((m=F(q->p[i])) > n) n=m;
return n;}

```

Из сравнения программ видно, что в них имеются сходные конструкции, заключающиеся в условном присваивании в теле цикла, вид их не зависит ни от структуры данных, ни от того, находится ли максимум в виде самого значения, указателя на него или его индекса. Неважно также, каким образом просматривается последовательность элементов. Если оставить только общие части, то получится даже не конструкция языка, а некоторая логическая схема:

```

for (s=«первый объект», «цикл по множеству объектов»)
    if («очередное» > s) s=«очередное»;

```

Эта схема имеет двоякое значение. Во-первых, в каких бы контекстах она ни встречалась – результат один и тот же. Во-вторых, она определяет *смысл* переменной *s*.

Кроме того, есть еще некоторое количество логических конструкций программы, понимание которых требует обращения не столько к логике, сколько к здравому смыслу. Убедительность и доказательность их состоит в их очевидности. А очевидность заключается в том, что им можно найти аналогии в обычном «физическом» мире, например, в виде перемещений, сдвигов и других взаимосвязанных движений объектов в пространстве.

Таким образом, умение читать программы – это не просто повторение того, что написано на языке программирования, но другими словами. Это даже не интерпретация, то есть не последовательное выполнение операторов программы в голове или на бумаге. Чтение программы – это умение «видеть» знакомые фрагменты, выделять их и уже затем воссоздавать результат ее работы путем логического соединения в единое целое.

Итак, процесс понимания программы (кстати, как и процессы ее написания и трансляции) не является линейным. Научно выражаясь, он представляет собой *диалектическое единство анализа и синтеза*:

– разложение программы на стандартные фрагменты, формулировка смысла каждого из них, а также смысла переменных;

– соединение полученных частей в единое целое и формулировка результата. Вот здесь для понимания сущности взаимодействия фрагментов друг с другом можно интерпретировать (выполнять, прокручивать) части программы в голове, на бумаге или в отладчике. Это позволяет увидеть *вторичный* смысл программы, который в явном виде не присутствует в ее тексте.

Итак, для более-менее свободного общения на любом языке программирования необходимо знать некоторый минимум «расхожих фраз» – общеупотребительных программных контекстов.

ПРИСВАИВАНИЕ КАК ЗАПОМИНАНИЕ

Без сомнения, присваивание является самой незаслуженно обиженной операцией в изложении процесса программирования. Утилитарно понимаемое присваивание – это запоминание результата, что характерно прежде всего при взгляде на программу как на калькулятор с памятью. А ведь на самом деле присваивание поднимает уровень поведения программы *от инстинктивного до рефлексорного*. Аналогия с животным миром вполне уместна. Инстинктивное поведение – это воспроизведение заданной последовательности действий, хоть и зависящих от внешних обстоятельств, но не включающих в себя запоминания и, тем более, обучения. Присваивание – это запоминание фактов, событий в жизни программы, которые затем могут быть востребованы.

Присваивание – запоминание фактов и событий в истории работы программы.

Такая интерпретация ориентирует программиста на постановку вопросов: что и когда должна запоминать программа, и с какими ее фрагментами связано это запоминание?

Место (конструкция алгоритма), где происходит запоминание, определяется условиями, при которых программа туда попадает. Например, при обменной сортировке место перестановки пары элементов запоминается в том фрагменте программы, где эта перестановка происходит.

```
for (i=0; i<n-1; i++)  
    if (A[i]>A[i+1])  
        {  
            c=A[i]; A[i]=A[i+1]; A[i+1]=c;  
            b1=i; // Запоминание индекса в момент перестановки  
        }  
// Условие перестановки  
// Перестановка
```

Запоминающая переменная имеет тот же самый *смысл* (ту же смысловую интерпретацию), что и запоминаемая. Так, в предыдущем примере, если переменная *i* является индексом в массиве, то **b1** также имеет смысл индекса.

Если запоминание производится в цикле, то по окончании цикла будет сохранено значение *последнего из возможных*. Так, в нашем примере **b1** – это индекс *последней* перестановки. Если же требуется запомнить значение *первого* из возможных, то присваивание нужно сопроводить альтернативным выходом из цикла через **break**. Если требуется запоминание максимального/минимального значения, то присваивание нужно выполнить в контексте выбора максимума/минимума.

ПЕРЕМЕННАЯ-СЧЕТЧИК

Переменная считает количество появлений в программе того или иного события, количество элементов, удовлетворяющих тому или иному условию. Ключевая фраза, определяющая смысл переменной-счетчика:

```
for (m=0,...) { if (...удовлетворяет условию...) m++; }
```

Логика данного фрагмента очевидна: переменная-счетчик увеличивает свое значение на 1 при каждом выполнении проверяемого условия. Остается только сформулировать смысл самого условия. В следующем примере переменная **m** подсчитывает количество положительных элементов в массиве.

```
for (i=0, m=0; i<n; i++) if(A[i]>0) m++;
```

Необходимо также обратить внимание на то, когда «сбрасывается» сам счетчик. Если это делается однократно, то процесс подсчета происходит однократно во всем фрагменте. Если же счетчик сбрасывается при каком-то условии, то такой процесс подсчета сам является повторяющимся. В следующем примере переменная-счетчик последовательно нумерует (считает) символы в каждом слове строки, сбрасываясь по пробелу между словами:

```
for(m=0,i=0; c[i] != 0; i++)  
    if (c[i] == ' ') m=0;  
    else    m++;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Сформулируйте результат выполнения фрагмента (функции) и определите роль переменной-счетчика.

```
//-----13-01.cpp
//-----1
for (i=0,s=0; i<10; i++)
    if (A[i]>0) s++;
//-----2
for (i=1,s=0; i<10; i++)
    if (A[i]>0 && A[i-1]<0) s++;
//-----3
for (i=1,s=0,k=0; i<10; i++)
{
    if (A[i-1]<A[i]) k++;
    else { if (k>s) s=k;
          k=0;
        } }
//-----4
for (s=0,n=2; n<a; n++)
{ if (a%n==0) s++; }
if (s==0) printf("Good\n");
//-----5
void sort(int in[],int out[],int n)
{ int i,j ,cnt;
for (i=0; i< n; i++)
{
    for ( cnt=0,j=0; j<n; j++)
        if (in[j] > in[i]) cnt++;
    else
        if (in[j]==in[i] && j>i) cnt++;
    out[cnt]=in[i];
}}
//-----6
void F(char *p)
{ char *q; int n;
for (n=0, q=p; *p !='\0'; p++)
{
    if (*p !=' ')
        { n=0; *q++ = *p; }
    else
        { n++; if (n==1) *q++ = *p; }
}}

```

ПЕРЕМЕННАЯ-НАКОПИТЕЛЬ

Не собирайте себе сокровищ на земле,
где моль и ржа истребляют, и где воры
подкопывают и крадут.

Евангелие от Матфея, гл. 6., ст. 19

Смысл накопительства: к тому, что уже имеешь, добавляй то, что получаешь. Если эту фразу перевести на язык программирования, а под накопленным значением подразумевать сумму или произведение, то получим еще один ключевой фрагмент:

```
for (s=0,...;...;...) { получить k; s=s+k; }
```


Он дает переменной s единственный смысл: переменная накапливает сумму значений k , полученных на каждом из шагов выполнения цикла. Этот факт достаточно очевиден и сам по себе – на каждом шаге к значению переменной s добавляется новое k и результат запоминается в том же самом s . Для особо неверующих в качестве строгого доказательства можно привлечь метод математической индукции. Действительно, если на очередном шаге цикла s содержит сумму, накопленную на предыдущих шагах, то после выполнения $s=s+k$ она будет содержать сумму уже с учетом текущего шага. Кроме того, утверждение должно быть верно в самом начале – этому соответствует обнуление переменной s для суммы и установка ее в 1 для произведения.

```
for (s=0,i=0; i<10; i++) s=s+A[i];
```

```
for (s=1,i=0; i<10; i++) s=s*A[i];
```

Накопление может происходить в разных контекстах, но они не меняют самого принципа. В приведенных примерах накапливается сумма значений, полученных разными способами и от разных источников:

```
for (s=0,i=0; i<n; i++)           // Сумма элементов массива
    s+=A[i];
```

```
for (s=0,i=0; i<n && A[i]>=0; i++) // Сумма элементов массива до первого
    s+=A[i];                     // отрицательного
```

```
for (s=0,i=0; i<n; i++)           // Сумма положительных элементов
    if (A[i]>0) s+=A[i];          // массива
```

```
for (s=0,x=0; x<=1; x+=0.1)       // Сумма значений функции sin
    s+=sin(x);                   // в диапазоне 0..1 с шагом 0.1
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Сформулируйте результат работы фрагмента и назначение переменной-накопителя.

```
//-----13-02.cpp
//-----1
for (s=1, i=1; i<10; i++) s = s * i;
//-----2
for (s=1, i=0; i<10; i++) s = s * 2;
//-----3
for (i=0, s=1; s < n; i++) s = s * 2;
printf("%d",i);
//-----4
for (s=0,i=0; i<n && A[i]>=0; i++) s+=A[i];
//-----5
for (s=0,i=0; i<n; i++)
```

```

        if (A[i]>0) s+=A[i];
//-----6
for (s=0, i=0, k=0; i < 10 && k ==0; i++)
    { s = s + A[i]; if (A[i]<=0) k=1; }
//-----7
struct tree { int v; tree *p[4]; };
int F(tree *q)
{ int i,n,m;
if (q==NULL) return 0;
for (n=q->v,i=0; i<4; i++)
    n+=F(q->p[i]);
return n; }

```

ПЕРЕМЕННАЯ-МИНИМУМ (МАКСИМУМ)

Фрагмент, выполняющий поиск минимального или максимального значения в последовательности, встречается даже чаще, чем остальные, но почему-то менее «узнаваем» в окружающем контексте. Следующая логическая схема дает переменной *s* единственный смысл – переменная находит максимальное из значений *k*, полученных на каждом из шагов выполнения цикла.

```
for (s=меньше меньшего,...;...;...) { получить k; if (k>s) s=k; }
```

Доказать это не сложнее, чем в случае с переменной-накопителем. Фрагмент **if(k>s) s=k;** читается буквально так: если новое значение больше, чем то, которое имеется у нас, вы его запоминаете, иначе оставляете старое. То есть осуществляется обычный принцип выбора «большого из двух зол». Формальное доказательство – опять же с использованием метода математической индукции: действительно, если на очередном шаге *s* содержит максимальное значение, полученное на предыдущих шагах, то после выполнения **if (k>s) s=k;** она будет содержать такой же максимум, но уже с учетом текущего шага. То есть из справедливости утверждения на текущем шаге доказана справедливость его же на следующем. Однако здесь следует обратить внимание на первый (начальный) шаг. Начальное значение *s* должно быть меньше первого значения *k*. Обычно в качестве *s* выбирают первый элемент последовательности, а алгоритм начинают со второго (или же с первого). Если таковой сразу не известен, то состояние поиска первого элемента обозначается специальным значением (признаком).

Типичный пример – нахождение максимального элемента массива.

```
for (s=A[0],i=1; i<10; i++) if (A[i]>s) s=A[i];
```

Рассмотрим более сложные вариации на эту тему. Следующий фрагмент запоминает не само значение максимума, а номер элемента в массиве, где оно находится.

```
for (i=1,k=0; i<10; i++) if (A[i]>A[k]) k=i;
```

И, наконец, если в просматриваемой последовательности в поиске максимума/минимума используются не все элементы, а ограниченные дополнительным условием (например, минимальный из положительных), в программе должен быть учтен тот факт, что она начинает работу при отсутствии элемента, выбранного в качестве первого максимального/минимального.

```
for (i=0,k=-1; i<10; i++) // k=-1 - нет элемента, принятого за минимальный
{ if (A[i]<0) continue;
  if (k==-1 || A[i]<A[k]) k=i;
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Найдите фрагмент поиска минимума (максимума) и сформулируйте результат работы программы.

```
//-----13-03.cpp
//-----1
for (i=1,s=A[0]; i<10; i++)
    if (A[i]>s) s=A[i];
//-----2
for (i=1,k=0; i<10; i++)
    if (A[i]>A[k]) k=i;
//-----3
for (i=0,k=-1; i<10; i++)
{ if (A[i]<0) continue;
  if (k==-1) k=i;
  else
    if (A[i]<A[k]) k=i;
}
//-----4
for (i=0,k=-1; i<10; i++)
{ if (A[i]<0) continue;
  if (k==-1 || A[i]<A[k]) k=i;
}
//-----5
char *F6(char *p[]) // strlen(char *) – длина строки
{ int i,sz,l,k;
  for (i=sz=k=0; p[i]!=NULL; i++)
    if ((l=strlen(p[i])) >sz) { sz=l; k=i; }
  return(p[k]); }
//-----6
struct tree { int v; tree *p[4]; };
int F(tree *q)
{
```

```
int i,n,m;
if (q==NULL) return 0;
for (n=q->v,i=0; i<4; i++)
    if ((m=F(q->p[i])) >n) n=m;
return n;}
```

ПЕРЕМЕННАЯ-ПРИЗНАК

Признак бродит по Европе – признак коммунизма.

*Реминисценция к «Манифесту коммунистической партии»
К.Маркса и Ф.Энгельса*

Отмеченная выше роль присваивания как средства запоминания истории работы программы наглядно проявляется в переменных-признаках. Признак – это логическая переменная, принимающая значения 0 (ложь) или 1 (истина) в зависимости от наступления какого-либо события в программе (событие наступило – 1 или не наступило – 0). В одной точке программы проверяется это условие и устанавливается признак, в другой – наличие или отсутствие признака влияет на логику работы программы, в третьей – признак сбрасывается. Простой пример – суммирование элементов массива до первого отрицательного включительно.

```
for (s=0, k=0, i=0; i<n && k==0; i++)
{
    s+=A[i];
    if (A[i]<0) k=1;
}
```

В данном случае переменная-признак **k** устанавливается в 1 после обнаружения и добавления к сумме отрицательного элемента массива. Установка этого признака нарушает условие продолжения и прекращает выполнение цикла. Эквивалентный вариант с использованием **break** позволяет обойтись без такого признака.

```
for (s=0, i=0; i<n; i++)
{
    s+=A[i];
    if (A[i]<0) break;
}
```

Сложнее распознать роль признака при его многократной установке и сбрасывании, например, если признак устанавливается или сбрасывается на каждом шаге цикла. Нужно учитывать тот факт, что установленное значение сохраняется некоторое время, в данном случае – до следующего шага. То есть в начале шага признак хранит свое значение, полученное на предыдущем.

```

for (i=0,s=0,k=0; i<10; i++)
    if (A[i]<0) k=1;
    else
        { if (k==1) s++; k=0; }

```

Несложно догадаться, что смысл переменной-признака **k** – элемент массива является отрицательным, причем в начале следующего шага признак сохраняет свое значение, полученное на предыдущем. Счетчик **s** увеличивается, если выполняется ветка **else** – текущий элемент массива положителен, и в то же самое время условие **k==1** – соответствует отрицательному значению *предыдущего* элемента массива, поскольку его сброс в 0 происходит позже. Следовательно, фрагмент подсчитывает количество пар элементов вида «отрицательный–положительный».

Еще один пример – обнаружение комментариев в строке. Признак **com** устанавливается в 1, если программа находится «внутри комментария». Процесс переписывания происходит при нулевом значении признака, то есть «вне комментария».

```

void copy(char dst[], char src[])
{ int i,com=0,j=0;
  for (com=0,i=0; src[i]!=0; i++)
  if (com==1)
      { // Внутри комментария
        if (src[i]=='*' && src[i+1]=='/')
            { com=0; i++; } // Не в комментарии, пропустить символ
        }
  else
      { // Вне комментария
        if (src[i]=='/' && src[i+1]=='*')
            { com=1; i++; } // В комментарии, пропустить символ
        else
            dst[j++] = src[i]; // Переписать символ в выходную строку
        }
  dst[j]=0; }

```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите смысл и назначение переменных-признаков.

```

//-----13-04.cpp
//-----1
int F1(char c[])
{ int i,old,nw;
  for (i=0, old=0, nw=0; c[i] !='\0'; i++){
    if (c[i]==' ') old = 0;
    else { if (old==0) nw++; old=1; }
    if (c[i]=='\0') break;
  }
  return nw; }
//-----2
void F2(char c[])

```

```

{ int i, k;
for (i=0, k=1; c[i] !='\0'; i++){
    if (c[i]=='.') k = 1;
    if (c[i]>='a' && c[i]<='z' && k==1)
        { k=0; c[i]+='A'-'a'; };
    }}

```

ПРАВИЛО ТРЕХ СТАКАНОВ

Простая житейская мудрость – для обмена содержимого двух стаканов (без смешивания) необходим третий стакан – дает в результате простой алгоритм обмена значений двух переменных:

```

// Обмен значений переменных a, b с использованием переменной c
int a=5,b=6;
int c;
c=a;    // Перелить содержимое первого стакана в пустой (третий) стакан
a=b;    // Перелить второй в первый
b=c;    // Перелить третий во второй

```

Данный контекст настолько очевиден, насколько и распространен.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Найдите контекст «три стакана» и объясните его назначение в программе.

```

//-----13-05.cpp
//----- 1
void F1(int in[],int n)
{ int i,j,k,c;
for (i=1; i<n; i++)
    { for (k=i; k !=0; k--)
        { if (in[k] > in[k-1]) break;
          c=in[k]; in[k]=in[k-1]; in[k-1]=c;
        } } }

//-----2
void F2(int A[], int n)
{ int i,found;
do { found=0;
    for (i=0; i<n-1; i++)
        if (A[i] > A[i+1])
            { int cc; cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
              found++;
            }
    } while(found !=0); }

//-----3
void F3(char c[])
{ int i,j;
for (i=0; c[i] !='\0'; i++);
for (j=0,i--; i>j; i--,j++)
    { char s; s=c[i]; c[i]=c[j]; c[j]=s; }
}

```

ПРЕДЫДУЩИЙ, ТЕКУЩИЙ, ПОСЛЕДУЮЩИЙ

Сталин – это Ленин сегодня.

Из лозунгов

Еще одна простая формальность, необходимая для чтения программ: если имеется последовательность адресуемых по номерам элементов, например, элементов массива, то по отношению к i -му элементу, с которым программа работает на текущем шаге цикла, $i-1$ будет предыдущим, а $i+1$ – последующим. Так и следует, особенно не задумываясь, переводить с формального на естественный язык и обратно.

```
int F(char c[]){
int nw=0;
if (c[0]!=0) nw=1;           // Строка начинается не с пробела – 1 слово
for (int i=1; c[i]!=0; i++)   // Сочетание не пробел, а перед ним – пробел
    if (c[i]!=' ' && c[i-1]!=' ') nw++;
return nw;}
```

Если *текущий* символ строки – не пробел и одновременно *предыдущий* символ строки – пробел, то к счетчику добавляется 1. Сочетание «пробел–не пробел», как нетрудно догадаться (а этого уже в программе не увидите), является началом слова. Таким образом, программа подсчитывает количество слов в строке, реагируя на их начало. Если строка начинается со слова и перед ним нет пробела, то такая ситуация отслеживается отдельно.

Если же элементы последовательности прямо не адресуются по номерам, то предыдущий и «более ранние» можно фиксировать «исторически». При переходе к следующему шагу цикла данные о расположении текущего элемента (например, указатель) можно запомнить в отдельной переменной, которая на следующем шаге будет играть роль «предыдущей». Такой прием используется в односвязном списке, исключая движение «вспять», – для вставки перед заданным элементом необходимо помнить указатель на предыдущий.

```
//-----13-06.cpp
//--- Включение в односвязный с сохранением порядка
// pr - указатель на предыдущий элемент списка
void InsSort(list *amp, int v)
{ list *q, *pr,*p;
q=new list; q->val=v;
// Перед переходом к следующему элементу указатель на текущий
// запоминается как указатель на предыдущий
for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p, p=p->next);
if (pr==NULL)           // Включение перед первым
    { q->next=ph; ph=q; }
```

```

else                                     // Иначе после предыдущего
    { q->next=p;                         // Следующий для нового = текущий
      pr->next=q; }                     // Следующий для предыдущего = новый

```

Включение с сохранением порядка происходит перед первым, бoльшим включаемого, при этом предыдущий элемент должен ссылаться на новый.

Аналогичные присваивания производятся в **итерационных циклах**, где каждый шаг характеризуется «текущим» значением переменной, вычисляемой или выводимой из ее «предыдущих» значений, точнее, значений на предыдущих шагах того же цикла. В них при переходе к следующему шагу «текущее» значение становится «предыдущим», а иногда и «вчерашнее» – «позавчерашним» (см. раздел 2.3).

КОНТРОЛЬНЫЕ ВОПРОСЫ

Сформулируйте условия, проверяемые программой в терминах «текущий, предыдущий, следующий». Определите переменные, имеющие смысл «текущей» и «предыдущей».

```

//-----13-07.cpp
//----- 1
int F1(int A[], int n){
for (int m=0, k=0, i=1; i<n; i++)
    if (A[i-1]<A[i]) k++;
    else { if (k>m) m=k; k=0; }
return m;}
//-----2
void F2(int A[], int n)
{ int i,found;
do { found=0;
    for (i=0; i<n-1; i++)
        if (A[i] > A[i+1])
            { int cc; cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
              found++;
            }
    } while(found !=0); }
//-----3
int F3(int A[], int n) {
for (int i=0, k=-1, nn=0; i<n; i++){
    if (A[i]<0) continue;
    if (k!=-1 && A[k] < A[i])  nn++;
    k=i;
}
return nn; }

```


ПЕРЕМЕЩЕНИЕ ЭЛЕМЕНТОВ В МАССИВЕ

С понятием текущего, предыдущего и последующего связаны регулярные перемещения элементов на один вправо-влево. Для восприятия этих примеров достаточно простой аналогии с книжной полкой: сдвиг элементов массива (последовательности) сопровождается их перемещением на предыдущую (последующую), но обязательно свободную позицию, что, в свою очередь, делается через присваивание. При этом сами перемещаемые «тома» берутся в последовательности, обратной направлению перемещения. Сказанное хорошо видно на примере удаления и вставки символа в строку на k -ю позицию (рис. 1.1).

```
//-----13-19.cpp
void insert(char c[], int k, char vv){
for (int n=0; c[n]!=0; n++);    // Длина строки
if (k>=n) return;              // Нет такого символа
for(int j=n; j>=k; j-- )        // Движение справа налево –
    c[j+1]=c[j];               // Перенести текущий в следующий
c[k]=vv;                        // Запись на освободившееся место
}
char remove(char c[], int k){
for (int n=0; c[n]!=0; n++);    // Длина строки
if (k>=n) return 0;            // Нет такого символа
char vv=c[k];                  // Сохранить удаляемый символ
for (int j=k; j<n; j++)          // Движение слева направо –
    c[j]=c[j+1];               // Перенести следующий в текущий
return vv; }

```

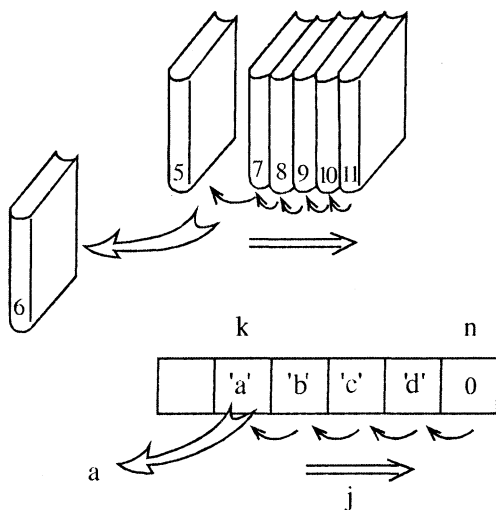


Рис. 1.1

Если производится вставка или исключение не одного, а нескольких подряд элементов, то схема процесса не меняется за исключением того, что перенос происходит не на один, а на несколько элементов вперед или назад. Например, функция, удаляющая в строке слово с заданным номером, после того как она определит индексы его начала и конца, должна выполнить процесс посимвольного перенесения «хвоста» строки. В нем вместо индексов **j** и **j+1** нужно использовать индексы **j** и **j+m**, «разнесенные» на длину слова **m**, либо индексы начала и конца слова **i** и **j** (рис. 1.2).

```
//-----13-08.cpp
//----- Удаление слова с заданным номером
void CutWord(char c[], int n){
int j=0;                                     // j - индекс конца слова
for (j=0; c[j]!=0; j++)
    if (c[j]!=' ' && (c[j+1]!=' ' || c[j+1]==0))
        if (n--==0) break; // Обнаружен конец n-го слова
if (n== -1 && c[j]!=0){ // Действительно был выход по концу слова
    for (int i=j; i>=0 && c[i]!=' '; i--); // Поиск начала слова
    i++; // Вернуться на первый символ слова
    for(j++; c[j]!=0; i++, j++) // Перенос очередного символа
        c[i]=c[j]; // ближе к началу
    c[i]=0; // Сам конец строки не был перенесен
}}
```

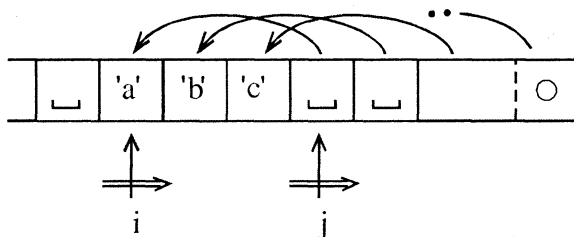


Рис. 1.2

КОНТРОЛЬНЫЕ ВОПРОСЫ

Содержательно опишите процесс перемещения элементов массива.

```
//-----13-09.cpp
//-----1
for (s=A[0], i=1; i < 10; i++) A[i-1] = A[i];
A[9] = s;
//-----2
for (i=0; i<5; i++)
    { c=A[i]; A[i]=A[9-i]; A[9-i]=c; }
//-----3
for (i=0, j=9; i < j; i++, j--)
    { c=A[i]; A[i]=A[j]; A[j]=c; }
```

ИНДЕКС КАК СТЕПЕНЬ СВОБОДЫ ПРИ ДВИЖЕНИИ ПО МАССИВУ

Степень свободы – независимая координата перемещения механической системы.

Определение (механика)

Образно говоря, программы, работающие с массивами, осуществляют различные «движения» по их элементам. Аналогии с механикой и физикой здесь не только уместны, но и необходимы, ибо помогают образно представлять программу, что является основой ее проектирования. Итак, работа с массивом – это движение по его элементам, которое определяется значениями индексов. Выбирая индексы и задавая алгоритм их изменения, мы тем самым выбираем закон движения – последовательный, равномерный, возвратно-поступательный, параллельный и т.д.

Вторая аналогия с механикой – каждому независимому перемещению по массиву должен соответствовать свой индекс. В предельной механике это соответствует термину «степень свободы».

Количество индексов в программе соответствует количеству независимых перемещений по массиву (степеней свободы).

Часто встречающаяся ошибка – попытка «убить одним индексом (в оригинале – выстрелом) несколько зайцев», то есть запрограммировать одним индексом несколько независимых перемещений. Другое дело, что вариантов выделения «степеней свободы» в программе может быть несколько. В каждом случае необходимо осмыслить «траекторию» движения выделенных индексов и дать им необходимую словесную интерпретацию.

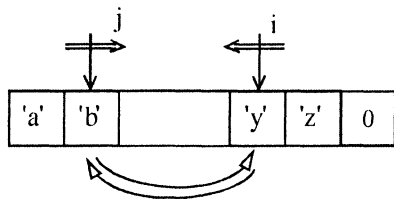


Рис. 1.3

Функция, «переворачивающая» строку, моделирует встречное движение двух индексов по строке от концов к середине. По отношению к каждой паре символов применяется «правило трех стаканов» для обмена их местами. Поскольку оба «движения» равномерны, они могут быть смоделированы двумя независимыми индексами, изменяемыми в заголовке цикла (рис. 1.3).

```
//-----13-10.cpp
//---- "Переворот" строки
void swap(char c[])
{ int i,j;
for (i=0; c[i] !='\0'; i++);          // Поиск конца строки
for (j=0,i--; i>j; i--,j++)           // Движение от концов к середине
    { char s; s=c[i]; c[i]=c[j]; c[j]=s; }    // Три стакана
}
```

По большей части перемещения по массивам – линейные, поступательные (последовательные). Им соответствует регулярное изменение индекса типа **i++** или **j--** в заголовке цикла. Соблюдается принцип: один шаг цикла – один элемент массива. Если же перемещение линейное, но не равномерное, а это бывает, когда оно обусловлено какими-то дополнительными моментами (и находится, соответственно, внутри каких-то условных конструкций), то индекс нужно менять там, где реально производится переход к следующему элементу.

В примере слияния последовательностей мы видим в одном цикле целых три индекса с различными «динамическими» свойствами. **Слияние** – это процесс соединения двух упорядоченных последовательностей в одну общую, тоже упорядоченную. Каждый шаг слияния включает выбор минимального из двух очередных элементов и перенос его в выходную последовательность. Каждая последовательность (массив) имеет собственный индекс, но только индекс выходного массива меняется линейно, поскольку за один шаг производится одно перемещение (рис. 1.4). Переход к следующему элементу во входной последовательности происходит только в одной из них (где выбран минимальный элемент), поэтому индексы изменяются (неравномерно) внутри условной конструкции. И еще одна деталь: каждая из входных последовательностей может закончиться раньше, чем противоположная, и это также необходимо отслеживать.

```
//-----13-11.cpp
//---- Слияние упорядоченных последовательностей
void sleeve(int out[], int in1[], int in2[], int n){
int i,j,k;          // Каждой последовательности – по индексу
for (i=j=k=0; i<2*n; i++){
    if (k==n) out[i]=in1[j++]; // Вторая кончилась – сливать первую
    else
    if (j==n) out[i]=in2[k++]; // Первая кончилась – сливать вторую
    else                // Слить меньший из очередных
    if (in1[j] < in2[k]) out[i]=in1[j++];
    else out[i]=in2[k++];
    }}
}
```

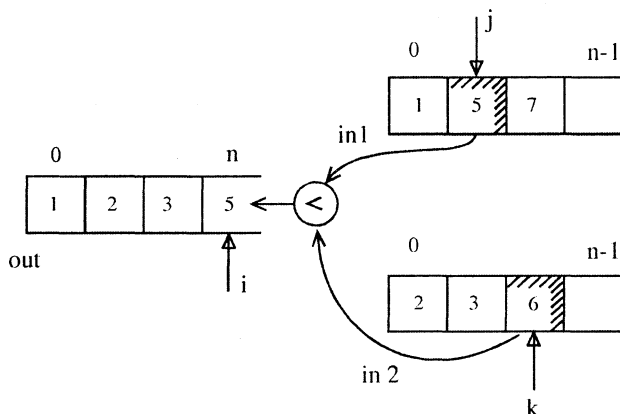


Рис. 1.4

Обратите внимание, что синтаксис `=in1[j++]` понимается как «взять очередной и переместиться к следующему».

Похожая картина имеет место в разделении. **Разделение** – это разбиение последовательности (массива) на две части по принципу «меньше-больше» относительно некоторого среднего значения, обычно называемого медианой. Пусть требуется разделить содержимое массива таким образом, чтобы в левой части выходного оказались значения, меньше медианы, а в правой – больше. Это легко можно сделать, заполняя выходной массив с двух концов. Здесь также потребуется три индекса (на два массива), причем только во входном индекс будет «двигаться» равномерно (рис. 1.5).

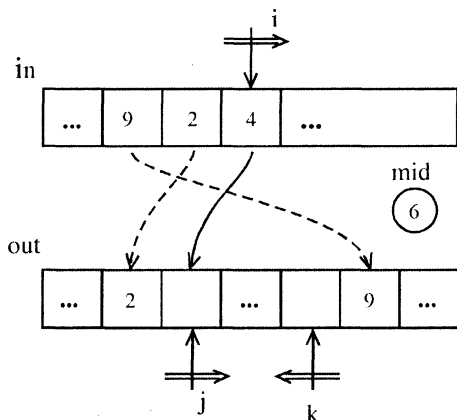


Рис. 1.5

```
//-----13-12.cpp
//--- Разделение массива относительно медианы
int two(int in[], int out[], int n, int mid){
int i,j,k;
for (i=0,j=0,k=n-1; i<n; i++){           // j, k – по концам выходного массива
    if (in[i]<mid) out[j++]=in[i]; // Переписать в левую часть
    else out[k--]=in[i];          // Переписать в правую часть
} return j; }                          // Вернуть точку разделения
```

Еще один маленький нюанс. Индексы **j, k** указывают на *очередные свободные* позиции выходного массива, а синтаксис **out[j++]** понимается как «записать очередным и переместиться к следующему свободному».

ВЛОЖЕННЫЕ ЦИКЛЫ – ПРИНЦИП ОТНОСИТЕЛЬНОСТИ

Наличие в программе линейных независимых «движений» – единственный случай. Часто эти перемещения по массивам и последовательностям имеют «возвратно-поступательный», «циклический» или какой-нибудь другой сложный геометрический характер. Но такое движение также раскладывается на линейные составляющие, другое дело, что в процессе выполнения программы они, как минимум, складываются или вычитаются. В этом случае образной модели помогает принцип относительности. Заключается он в том, что при анализе процесса, проходящего во внутреннем цикле, внешний можно считать «условно неподвижным». При этом нужно отказаться от попытки «исторически» отследить выполнение программы с первого шага внешнего цикла, а считать внутренний цикл выполняющимся в некотором его произвольном шаге.

```
//-----13-13.cpp
//--- Поиск подстроки в строке
int search(char c1[],char c2[]){
for ( int i=0; c1[i] !='\0'; i++){
    for ( int j=0; c2[j] !='\0'; j++)
        if (c1[i+j] != c2[j]) break;
    if (c2[j] =='\0') return i;
} return -1;}
```

Анализ программы необходимо начать с внутреннего цикла, содержащего суммируемый индекс **i+j**. Для его восприятия нужно зафиксировать внешний цикл, то есть производить рассуждения, исходя из анализа тела внешнего цикла для произвольного **i**-го символа. Тогда **c1[i+j]** следует понимать как **j**-й символ относительно текущего, на котором находится внешний цикл. Отсюда мы видим параллельное движение с попарным сравнением символов

по двум строкам, но вторая рассматривается от начала, а первая – от i -го символа (рис. 1.6). Теперь, определив характер процесса,

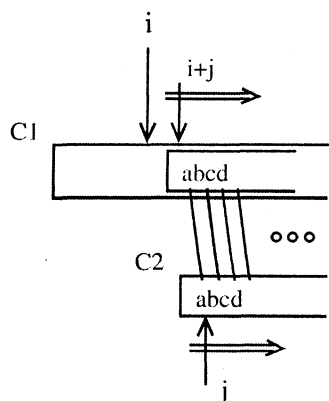


Рис. 1.6

можно анализировать условия его завершения. Попарное сравнение продолжается, пока не закончится вторая строка и пока фрагмент первой строки и вторая строка совпадают (совпадение очередной пары продолжает цикл). И наконец завершение цикла по концу второй строки свидетельствует о том, что вторая строка содержится в первой, начиная с i -го символа. Обнаружение этого условия приводит к тому, что функция завершается и возвращает этот индекс в качестве результата.

Анализ внешнего цикла тривиален.

Он просто выполняет описанное выше действие для каждого начального символа первой строки. Таким образом, функция находит первое вхождение подстроки в строке.

РЕЗУЛЬТАТ ЦИКЛА – В ЕГО ЗАВЕРШЕНИИ

Постой, паровоз, не стучите, колеса!
Кондуктор, нажми на тормоза...

*Песня из к/ф «Операция Ы
и другие приключения Шурика»*

Как известно, тело цикла представляет собой описание повторяющегося процесса, а заголовок – параметры этого повторения. Можно представить себе **«бестелесный»** цикл. Тогда возникает резонный вопрос: зачем он нужен? Ответ: результатом цикла является *место его остановки*. Оно, в свою очередь, определяется значениями переменных, которые используются в заголовке цикла. Такие циклы либо вообще не имеют тела (пустой оператор), либо содержат проверку условий, сопровождаемых альтернативными выходами через **break**.

Сортировка вставками. Принцип сортировки вставками: из неотсортированной части выбирается очередной элемент и помещается в уже отсортированную последовательность с сохранением упорядоченности. В этом алгоритме можно по-разному задать способ поиска места включения. Например, очередной элемент срав-

нивается подряд со всеми из упорядоченной последовательности в порядке возрастания, *пока не встретит элемент (первый), больше себя*. Другое естественное условие остановки – конец упорядоченной последовательности. В обоих случаях он должен останавливаться на элементе, на место которого будет произведено включение. Рассмотрим, как это выглядит на обычном массиве.

```
//-----13-14.cpp
//-----Сортировка массива вставками
void sort(int A[], int n)
{ int i,k;           // i граница отсортированной части
  for (i=1; i<n; i++) // Вставка A[i] в упорядоченную часть 0...i-1
  {
    //1. Сохранить текущий
    int v=A[i];
    //2. Поиск места включения k
    for (k=0; k<i && A[k]<v; k++);
    //3. Сдвиг вправо на один в диапазоне k..i-1
    for (int j=i-1; j>=k; j--) A[j+1]=A[j];
    //4. Вставка на освободившееся место
    A[k]=v;
  }
}
```

Аналогичный пример для односвязного списка учитывает тот факт, что для вставки перед заданным элементом необходимо корректировать указатель в предыдущем. С этой целью в цикле поиска места включения нужно сохранять указатель на предыдущий элемент.

```
//-----13-15.cpp
//----- Сортировка односвязного списка вставками
struct list { int val; list *next; };
list *F8(list *ph)      // Заголовок входного списка
{
  list *q ;              // Исключаемый – вставляемый
  list *pp, *pr;          // Текущий, предыдущий – место вставки
  list *tmp = NULL;       // Выходной список
  while (ph !=NULL)       // Пока входной список не пуст
  {
    //1. Исключить очередной элемент из входного
    q = ph; ph = ph->next;
    //2. Поиск места включения для q
    for (pp = tmp, pr=NULL; pp!=NULL && pp->val < q->val;
         pr=pp, pp=pp->next);
    //3. Вставка перед pr и после pr
    q->next = pr;
    if (pr==NULL) tmp=q; else pr->next=q;
  }
  return tmp;             // Вернуть новый список
}
```


КОНТРОЛЬНЫЕ ВОПРОСЫ

Найдите «пустые» циклы и объясните их назначение.

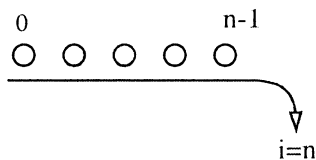
```
//-----13-16.cpp
//----- 1
void F1(char c[])
{ int i,j;
  for (i=0; c[i] !='\0'; i++);
  for (j=0,i--; i>j; i--,j++)
    { char s; s=c[i]; c[i]=c[j]; c[j]=s; } }
//----- 2
void F2(char c[], int n)
{ int nn,k;
  for (nn=n, k=0; nn!=0; k++, nn/=10);
  for ( c[k--]=0; k >=0; k--, n /= 10)
    c[k] = n % 10 + '0';
}
```

УСЛОВИЯ ВСЕОБЩНОСТИ И СУЩЕСТВОВАНИЯ

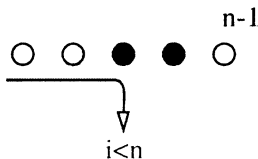
Ваше кредо? Всегда.

*И. Ильф и Е. Петров. Двенадцать
стульев. Из высказываний О. Бендера*

В программах часто производится проверка, все ли элементы из заданного множества обладают некоторым свойством (условие



Для всех ($\forall$)



Существует ($\exists$)

Рис. 1.7

всеобщности, свойство «для всех»), либо, наоборот, существует ли элемент – исключение из общего правила (условие существования). Например, простое число – это число (N), которое делится только на 1 и на само себя, то есть не делится ни на одно число в диапазоне от 2 до $N/2$. Первое, о чем необходимо напомнить в этом случае: свойство «для всех» может быть достоверно обнаружено только по завершении просмотра всего множества, в то время как обнаружение первого элемента, удовлетворяющего условию, уже достоверно свидетельствует о выполнении условия существования (рис. 1.7). С позиций структурного проектирования желательно

в любом случае вынести использование обнаруженного свойства за пределы цикла проверки. Цикл проверки можно организовать

формально, завершая его двумя условиями – достижением конца множества и обнаружением условия существования.

```
for (int i=0; i<размерность множества; i++)
    if (A[i] удовлетворяет условию X) break;
...
if (i<n) существует A[i], удовлетворяющее X
for (int i=0; i<размерность множества; i++)
    if (A[i] не удовлетворяет условию Y) break;
...
if (i==n) все A[i] удовлетворяют Y
```

Напомним также, что условия существования и всеобщности взаимосвязаны: невыполнение условия всеобщности говорит о существовании элемента с обратным условием, и наоборот. Поэтому оба приведенных фрагмента практически идентичны.

Перейдем от формальных схем к конкретным программам.

```
int A[20]={...};
for (int i=0; i<20; i++)
    if (A[i]<0) break;
if (i==20) puts("все положительные");
else puts("есть и отрицательный");
```

Наличие **break** в примерах – для простоты восприятия, его можно убрать, внося обратное условие в заголовок цикла, не содержащего тела.

```
int A[20]={...};
for (int i=0; i<20 && A[i]>=0; i++);
if (i==20) puts("все положительные");
else puts("есть и отрицательный");
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Сформулируйте условия, проверяемые циклами.

```
//-----13-17.cpp
//-----1
for (i=0; i<10; i++)
    if (A[i]<0) break;
if (i==10) printf("Good\n");
//-----2
for (i=2; i<a; i++)
    if (a%i==0) break;
if (i==a) printf("Good\n");
//-----3
for (n=a; n!=0; n/=10){
    k=n%10;
    for (i=2; i<k; i++)
        if (k%i==0) break;
    if (k!=i) break;
}
if (n==0) printf("Good\n");
```

```
//-----4
for (i=0; i<10; i++) {
    for (j=2; j<A[i]; j++)
        if (A[i]%j==0) break;
    if (j==A[i]) break;
}
if (i!=10) printf("Good\n");
//-----5
for (i=0, a=2; a<10000; a++) {
    for (n=2; n<a; n++)
        { if (a%n==0) break; }
    if (n==a) A[i++]=a;
} A[i]=0;
//-----6
for (i=0, a=2; a<10000; a++){
    for (j=0; j<i; j++)
        { if (a%A[j]==0) break; }
    if (j==i) A[i++]=a;
} A[i]=0;
```

ПЕРВЫЙ, ПОСЛЕДНИЙ, МАКСИМАЛЬНЫЙ, НАИМЕНЬШИЙ ИЗ ВОЗМОЖНЫХ

Большинство алгоритмов поиска подходящих вариантов, нахождения объектов, удовлетворяющих заданным свойствам, устроены достаточно примитивно: они просто перебирают все возможные значения, пока не встретят нужного. О таком простом подходе не следует забывать, ибо все остальное применимо, когда он не срабатывает. Приведем несколько очевидных логических схем организации таких программ:

- если программа перебирает множество и прерывает цикл просмотра при обнаружении элемента, удовлетворяющего условию, то она находит *первый из возможных*;

- если программа запоминает элемент, удовлетворяющий условию (его значение, индекс, адрес), то по окончании цикла просмотра она обнаружит *последний из возможных*;

- для поиска элемента с *максимальным* или *минимальным* значением необходимо перебрать все множество с использованием соответствующего контекста;

- если программа просматривает множество в порядке возрастания значений и прерывает цикл просмотра при обнаружении элемента, удовлетворяющего условию, то она находит *минимальный из возможных*;

- тот же самый процесс в порядке убывания приводит к обнаружению *максимального из возможных*.

Для сравнения приведем варианты поиска первого, последнего и минимального положительного элемента в массиве.

```

int A[20]={...};
int i,k;
for (k=-1,i=0; i<20; i++){           // Первый положительный
    if (A[i]<0) continue;
    k=i; break; }
for (k=-1,i=0; i<20; i++){           // Последний положительный
    if (A[i]<0) continue;
    k=i; }
for (k=-1,i=0; i<20; i++){           // Минимальный положительный
    if (A[i]<0) continue;
    if (k==-1 || A[i] < A[k]) k=i;
}

```

Оценить влияние направления поиска можно в примерах, находящихся *наибольший* общий делитель (в процессе убывания) и *наименьшее* общее кратное (в процессе возрастания).

```

int i,n1,n2;
for (i=n1-1; !(n1 % i ==0 && n2 % i ==0); i--);
printf("%d",i);

i = n1; if (i < n2) i=n2;
for (; !(i % n1 ==0 && i % n2 ==0); i++);
printf("%d",i);

```

ЖИТЕЙСКИЙ СМЫСЛ ЛОГИЧЕСКИХ ОПЕРАЦИЙ

Безусловно, нет нужды повторять определение логических операций **И**, **ИЛИ**, **НЕ**, используемых в любом языке программирования. Уместно напомнить, как «переводятся» эти операции на естественный язык при чтении программ:

- содержательный смысл логической операции **И** передается фразой «**одновременно оба...**» и заключается в совпадении условий;
- содержательный смысл логической операции **ИЛИ** передается фразой «**хотя бы один...**» и заключается в объединении условий;
- содержательный смысл логической операции **НЕ** передается фразой «**не выполняется...**» и заключается в проверке обратного условия.

Несколько замечаний можно сделать относительно эквивалентных преобразований логических выражений, часто используемых в программах:

- все условия, записанные в заголовках циклов Си-программ, являются условиями *продолжения цикла*. Если программисту удобнее сформулировать условие завершения, то в заголовке цикла его нужно записать, предварив операцией логической инверсии;

```

// Цикл завершается при обнаружении пары "меньше 0 - больше 0"
for (i=1; i<20 && !(A[i-1]<0 && A[i]>0); i++);

```

– оператор прерывания цикла **break**, по условию размещенный в начале тела цикла, может быть вынесен в заголовок цикла в виде инвертированного условия продолжения цикла, объединенного с имеющимся ранее по **И**:

```
for (int i=0; i<20; i++){           // До конца массива
    if (A[i]<0) break;              // Отрицательный - выход
    ...}
for (int i=0; i<20 && A[i]>=0){      // Пока не кончился массив
    ...}                           // И элемент неотрицательный
```

– инверсия условий, объединенных по **И**, раскрывается как объединение по **ИЛИ** обратных условий, и наоборот.

```
// Цикл прекращается, когда одновременно оба равны 0
for (i=1; !(A[i-1]==0 && A[i]==0); i++)...
// Цикл продолжается, пока хотя бы один не равен 0
for (i=1; A[i-1]!=0 || A[i]!=0; i++)...
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите формальные и содержательные условия завершения циклов.

```
//-----13-18.cpp
//-----1
for (i=2; n % i !=0; i++);
printf("i=%d\n",i);
//-----2
for (i=n1-1; !(n1 % i ==0 && n2 % i ==0); i--);
printf("i=%d\n",i);
//-----3
i = n1; if (i < n2) i=n2;
for (; ! (i % n1 ==0 && i % n2 ==0); i++);
printf("i=%d\n",i);
//-----4
for (n=2; n<a; n++)
    { if (a%n==0) break; }
if (n==a) printf("Good\n");
//-----5
for (s=0,n=2; n<a; n++)
    { if (a%n==0) s++; }
if (s==0) printf("Good\n");
//-----6
for (n=a; n%a!=0 || n%b!=0; n++);
printf("i=%d\n",n);
//-----7
for (n=a-1; a%n!=0 || b%n!=0; n--);
printf("i=%d\n",n);
//-----8
for (s=0, i=0; i < 10 && A[i] >0; i++)
    s = s + A[i];
//-----9
for (s=0, i=0, k=0; i < 10 && k ==0; i++)
    { s = s + A[i]; if (A[i]<=0) k=1; }
```

О ВЕЩАХ ВИДИМЫХ И НЕ ВИДИМЫХ НЕВООРУЖЕННЫМ ГЛАЗОМ

Если бы программа представляла собой механическое соединение стандартных фрагментов, то ее результат можно было бы определить простым соединением «смыслов», заключенных в стандартных программных контекстах. На самом деле фрагменты взаимодействуют через общие данные, что уже невозможно увидеть в тексте программы. Поэтому следующим этапом является анализ взаимодействия фрагментов в процессе их выполнения, а здесь нельзя обойтись без «исторического» подхода.

Попытаемся «прочитать» и понять следующий пример.

```
for (s=-1,m=0, k=0, i=1; i<20; i++)  
    if (A[i-1]<A[i])  
        k++;  
    else {  
        if (k>m) { m=k; s=i-k-1; }  
        k=0;  
    }
```

Для начала просто перечислим известные «ключевые фразы» и определим их смысл:

1. Смысл цикла **for()** – последовательный просмотр элементов массива, **i** – индекс текущего элемента.

2. Смысл переменной **m** из выражения **if (k>m) m=k;** – выбор максимального значения из последовательности получаемых значений **k**.

3. Параллельно с запоминанием максимального значения **k** запоминается выражение **i-k-1**, которое, очевидно, как-то связано с расположением искомого фрагмента или свойства в массиве, поскольку использует индекс в нем.

4. **A[i]** – текущий элемент массива, **A[i-1]** – предыдущий элемент массива, **A[i-1]<A[i]** имеет смысл: два соседних элемента массива (предыдущий и текущий) расположены в порядке возрастания.

5. Смысл переменной **k** из выражения **if () k++;** – переменная-счетчик.

6. Смысл фрагмента **if (A[i-1]<A[i]) k++;** – подсчет количества пар соседних элементов, расположенных в порядке возрастания.

Далее необходимо соединить фрагменты в единое целое. Поскольку все они включены в тело одного цикла, нужно промоделировать поведение программы на нескольких его шагах, точнее попытаться оценить возможные сочетания их последовательного вы-

полнения. В нашем примере необходимо ответить на вопрос, как поведет себя программа при разных сочетаниях возрастающих и убывающих пар.

7. После фиксации очередного значения **k** на предмет определения максимума в **m** его значение сбрасывается, то есть процесс подсчета начинается сначала.

8. Очевидно, что процесс подсчета **k** связан каким-то образом с процессом возрастания значений **A[i]**. Если несколько значений расположены подряд в порядке возрастания, то выполняется одна и та же ветка **if**, а **k** последовательно увеличивается. При появлении первого убывающего значения в последовательности счетчик сбрасывается. Таким образом, счетчик **k** считает количество *подряд расположенных* возрастающих пар.

i=	0	1	2	3	4	5	6	7	8
A[]	3	4	5	2	1	3	4	6	2
k=0		k++	k++	k=0	k=0	k++	k++	k++	k=0
		1	2	0	0	1	2	3	0
m=0				m=k					m=k
				2					3
s=-1				s=i-k-1					s=i-k-1

9. Для понимания того, какое же значение фиксируется в качестве максимального, необходимо обратить внимание на *место*, в котором находится этот фрагмент. Максимум фиксируется перед тем, как счетчик сбрасывается при обнаружении убывающей пары, то есть по окончании процесса возрастания. Таким образом, переменная **m** сохраняет значение *максимальной длины последовательности возрастающих значений* в массиве, а **s** – индекс ее начала.

10. Есть еще тонкость, которая не нарушает получившейся идиллии. Если несколько пар расположены в порядке убывания, то фиксация максимума выполняется для каждой из них, но реально сработает только для первой, поскольку счетчик уже будет сброшен.

1.4. ПРОЦЕСС ПРОЕКТИРОВАНИЯ ПРОГРАММЫ

ОБРАЗНАЯ И ЛОГИЧЕСКАЯ СТОРОНЫ ПРОГРАММИРОВАНИЯ

Допустим, вы изучили синтаксис языка программирования, то есть знаете, как пишутся выражения, операторы, функции и что значит каждое из этих понятий в отдельности. Допустим, вы разобрались в стандартных программных контекстах и обладаете

«джентльменским набором» программистских фраз и умеете «читать» чужие программы. Допустим, вы слышали о технологии структурного программирования – модульного, нисходящего, пошагового, «без goto». И что же? Как правило, даже при понимании сущности программы, которую необходимо разработать, начинающий не знает, с чего начать и как соединить воедино все известные ему факты, имеющие к ней отношение. Видимо, есть еще нечто, не имеющее отношения к перечисленному выше. Попробуем очертить границы этой части процесса проектирования программ.

То, что язык программирования, как таковой, не имеет отношения к процессу написания программ – это факт из того же разряда, что столярный инструмент не гарантирует качества табуретки и не определяет последовательность технологических операций при ее изготовлении. Отсюда следует практическая бесполезность в этом плане многочисленной литературы по системам программирования.

Любая технология программирования имеет отношение прежде всего к формальной стороне проектирования. Так, структурное программирование предполагает последовательное движение от внешних программных конструкций к внутренним, но что определяет направление этого движения?

Программы не создаются из набора заготовок путем их механического или стохастического (вероятностного) соединения. Даже если известны составные части программы, в какой последовательности их соединять?

Все эти вопросы останутся без ответа, пока мы будем рассматривать только формально-логическую сторону процесса программирования. Но в самом начале и на любом промежуточном шаге проектирования программы имеет место *образное* ее представление в виде целостной «движущейся картинки», из которой очевидно, как выполняется процесс, приводящий к результату. Словесные формулировки алгоритма типа «переместить выбранный элемент к концу массива» уже сочетают в себе образное и формально-логическое (алгоритмическое) описание действия. Следовательно, программирование – это движение в направлении от образной модели к словесным формулировкам составляющих действий, а уже затем – к формальной их записи на языке программирования.

Попробуем для начала определить составляющие этого процесса (рис. 1.8).

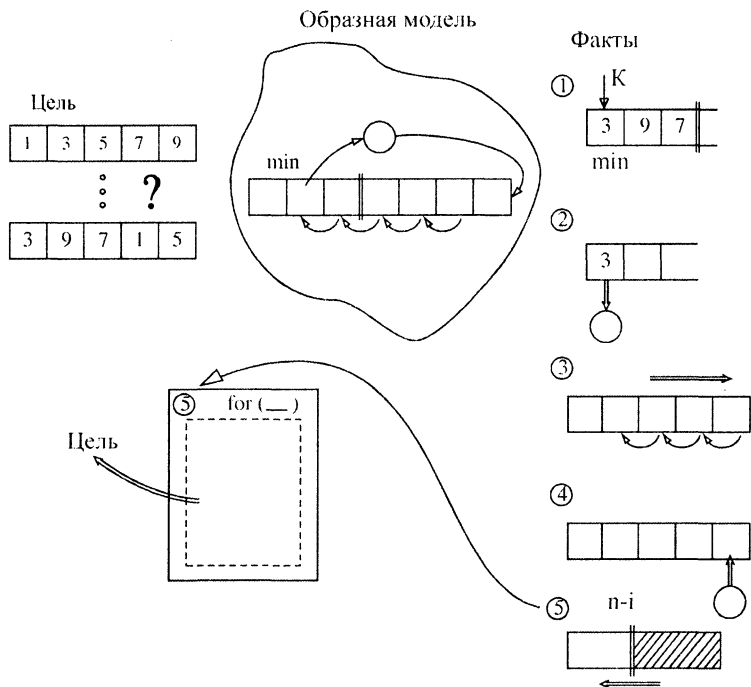


Рис. 1.8

1. Цель работы программы. Целью выполнения любой программы является получение результата, а результат – это данные с определенными свойствами. Например, цель программы сортировки – создание последовательности из имеющихся данных, расположенных в порядке возрастания. Точно так же любой промежуточный шаг программы имеет свою цель: получить данные с нужными свойствами в нужном месте.

2. Образная модель программы. Формальное проектирование программы не продвинется ни на шаг, если программист «не видит», как это происходит. То есть первоначально «действующая модель» программы должна присутствовать в голове. Понятно, что к формальной логике это не имеет никакого отношения. Это – область образного мышления, грубо говоря, «правого полушария». Изобразительные средства здесь уместны любые – словесные, графические. Здесь работают интуиция, аналогия, фантазия и другие элементы творческого процесса. На этом уровне справедлив тезис, что программирование – это искусство. Насколько подробно программист «видит» модель в движении и насколько он способен

описать это словами – настолько он близок к следующему этапу проектирования.

3. Факты, касающиеся программы. Формальная сторона проектирования начинается с перечисления фактов, касающихся образной модели программы. К таковым относятся: переменные и их смысловая интерпретация, известные программные решения и соответствующие им стандартные программные контексты. Сразу же надо заметить, что речь идет не об окончательных фрагментах программы, а о фрагментах, которые могут войти в готовую программу. Иногда при их включении потребуются доопределить некоторые параметры (например, границы выполнения цикла, которые не видны на этапе сбора фактов). Иногда они могут быть эквивалентно преобразованы (то есть иметь другой синтаксис). Умение «видеть» в алгоритме известные частные решения тоже приобретается с опытом: для этого и нужно учиться «читать» программы.

4. Выстраивание программы из набора фактов. Эта часть процесса программирования вызывает наибольшие затруднения, ибо здесь начинается то, что извне обычно и воспринимается как «программирование»: написание текста программы. Особенность заключается в том, что обычно фрагменты вложены друг в друга, то есть один является частью другого, а потому в значительной степени взаимозависимы. Кроме того, в программе есть еще и данные: их проектирование должно идти параллельно. Различие подходов состоит в том, с какой стороны начать этот процесс и в каком направлении двигаться.

«Историческое» проектирование соответствует естественному ходу рассуждений по линии наименьшего сопротивления. Программист просто записывает очередной оператор, который, по его мнению, должен выполняться программой. Ошибочность такого принципа состоит в том, что текст программы и последовательность ее выполнения – это не одно и то же, и расхождение между ними рано или поздно обнаружится. Хорошо, если это случится, когда большая часть программы уже написана, и проблема скорректируется несколькими «заплатками» в виде операторов **goto**. Заметим, что «историческим» подходом грешны не только программы, но и любые другие структурированные тексты (например, магистерская диссертация), если автор не уделяет должного внимания логике их построения.

Восходящее проектирование – проектирование программы «изнутри», от самой внутренней конструкции к внешней. Привле-

кательность этого подхода обусловлена тем, что внутренние конструкции программы – это частности, которые всегда более «на виду», чем внешние конструкции, реализующие обобщенные действия. Частности составляют большую часть фактов в образной модели программы и, что самое ценное, могут быть непосредственно записаны на языке программирования. Поэтому программа при написании не нуждается, как и в «историческом» подходе, в иных средствах описания, кроме самого языка программирования. Недостатки тоже очевидны:

- не факт, что программу удастся «свести» в единое целое, особенно сложную;

- поскольку параметры внутренних конструкций могут зависеть от внешних (например, диапазон поиска минимального значения во внутреннем цикле зависит от шага внешнего цикла), то внутренние конструкции не есть «истины в последней инстанции» и по мере написания программы тоже должны корректироваться.

Нисходящее проектирование – проектирование программы, начиная с самой внешней ее конструкции. Самое правильное направление движения от общего к частному, но и самое трудное:

- трудно выбрать самую внешнюю конструкцию;

- после записи выбранной конструкции ее содержимое (вложенные операторы) не удастся сразу же записать средствами языка программирования, поскольку оно тоже может быть сколь угодно сложным.

Отсюда следует, что нисходящее проектирование должно сочетать в тексте программы формальное (то есть записанное на языке программирования) и неформальное (то есть словесное или даже образное) представления.

5. Последовательное приближение к результату. То, что опытный программист пишет программу, не пользуясь дополнительными обозначениями для еще не написанных фрагментов и не делая никаких «заметок на полях», еще не значит, что проектирование идет непрерывным потоком. На самом деле после записи очередной порции текста программы (например, заголовка цикла) в голове формируется цель, словесная формулировка или образное представление того, что должен делать следующий, ненаписанный, кусок. Чем меньше опыта и возможностей держать это в голове, тем больше изобразительных средств и средств документирования должно привлекаться к этому процессу.

Обзор подходов к проектированию программ начнем с неправильных.

«ИСТОРИЧЕСКОЕ» ПРОГРАММИРОВАНИЕ

Коль скоро программа представляет собой последовательность выполняемых действий, то начинающий программист обычно так и поступает: начинает записывать ход своих рассуждений, переводя его на язык логических конструкций языка программирования. Соответственно, получается так называемый «исторический» подход (рис. 1.9). Как правило, на третьей или четвертой конструкции человек начинает терять нить рассуждений и останавливается. Такой принцип изложения характерен для художественной литературы, да и то не всегда. По той причине, что литературный текст является последовательным, хотя и допускает вложенность («лирические отступления») и даже параллелизм сюжетных линий. С программой сложнее: ее логика включает не только последовательность действий, но и вложенность одних конструкций в другие. Поэтому начало некоторой конструкции может отстоять от ее конца на значительном расстоянии, но обдумываться она должна как единое целое. Тем не менее, эта технология программирования существует и даже имеет свое название: «метод северо-западного угла». Имеется в виду экран монитора или лист бумаги.

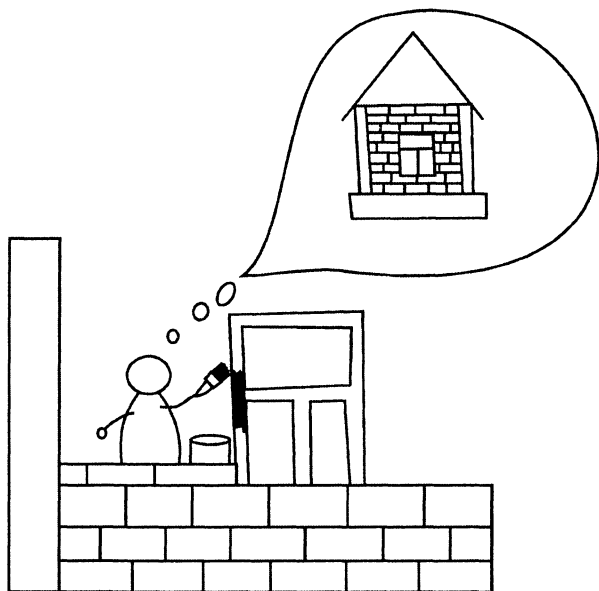


Рис. 1.9

Есть несколько признаков, по которым можно отличить «исторического» программиста:

- никогда сразу не закрывает синтаксическую конструкцию (оператор), пока не напишет содержимое вложенных в нее конструкций до конца. «Структурный» программист сначала пишет конструкцию, например, пару скобок { }, а потом начинает обдумывать и записывать ее содержимое;

- начинает обсуждение цикла с первого шага. «Структурный» программист сначала определяет условия протекания циклического процесса, а затем работает с произвольным его шагом.

Самый простой пример. Приводимый ниже пример использовался сначала для определения «нулевого» уровня знаний в разделе «Работа со строками». Оказалось, что его можно с равным успехом использовать для проверки, насколько «исторический» принцип превалирует над логическим. Итак, задана строка в массиве символов. Требуется дописать в конец символ «*». Некоторый процент начинающих рассуждает примерно так: необходимо найти конец строки, для чего надо написать цикл движения по строке. Далее: если встречается символ конца строки (символ с кодом 0), то необходимо заменить его на символ «*», а вслед за ним записать код конца строки. В результате получается примерно следующее:

```
char c[80]="aaaaaaaaa";
for (i=0; c[i]!=0; i++)
    if (c[i]==0) { c[i]='*'; c[i+1]=0; }
```

Даже невооруженным взглядом заметно, что эта программа работать не будет. Хотя бы потому, что внутри цикла проверяется условие, которое ограничивает этот же цикл. После указания на это противоречие некоторые программисты исправляют ошибку:

```
for (i=0; i<80; i++)
    if (c[i]==0) { c[i]='*'; c[i+1]=0; break; }
```

В таком виде программа работоспособна. Но на самом деле есть более естественный вариант, который предполагает последовательное выполнение двух действий: поиск конца строки и замена его на символ «*».

```
for (i=0; c[i]!=0; i++);
c[i]='*'; c[i+1]=0;
```

Этот вариант более предпочтителен хотя бы потому, что каждое из действий – независимо друга от друга в том смысле, что не подчиняется одно другому, не вложено одно в другое. То есть отношение последовательности и равноправия более приемлемо для

них, нежели отношение подчиненности и вложенности. Почему же с самого начала это не было видно? Именно потому, что первоначально алгоритм не рассматривался как последовательность *абстрактных* действий высокого уровня, не имеющих прямого выражения в простых операциях языка – поиск конца строки.

Шиворот-навыворот. «Исторический» подход в проектировании программы по своей природе выделяет фрагменты программы, лежащие «ближе к земле», то есть самого внутреннего уровня вложенности. При этом внешних, наиболее абстрактных, конструкций программист не замечает. К пониманию того, что они должны быть, он приходит уже позднее, и в дело вступают различные «заплатки» в виде операторов **goto** для возвращения на уже пройденные участки программы, как замаскированное проявление не замеченных вовремя циклов. В качестве примера достаточно посмотреть на любую сортировку (например, выбором или вставками). Начав с проектирования процесса выбора или вставки (причем конкретно для первого шага), программист оказывается перед фактом, что уже написанная часть программы должна повторяться. Благо, если ему достанет ума включить уже написанную часть в тело цикла и несколько подкорректировать границы протекания процессов. Сохраняющий приверженность «историческому» принципу напишет **goto**, сопровождая его манипуляциями с индексами. Например, в сортировке выбором ищется минимальный из неотсортированной части и переставляется с первым из неотсортированных. В результате неотсортированная часть сокращается слева на один элемент.

Итак, «историк» напишет фрагмент поиска минимального во всем массиве и обмен с первым. Это не составит труда, если программист помнит контекст поиска индекса минимального и правило «трех стаканов».

```
void sort(int A[], int n){
for (int i=1,k=0; i<n; i++)
    if (A[i]<A[k]) k=i;
int c=A[0]; A[0]=A[k]; A[k]=A[0];
...
}
```

После чего возникнет необходимость «отказаться» от 0-го элемента. Он будет заменен на j-й, а в программу добавится **goto** для повтора уже написанного фрагмента.

```
void sort(int A[], int n){
int j=0;
retry:
for (int i=j+1,k=j; i<n; i++)
```

```

        if (A[i]<A[k]) k=i;
int c=A[j]; A[j]=A[k]; A[k]=c;
j++; if (j!=n) goto retry;
}

```

Хорошо еще, если сразу станет понятно, что переменная **j** соответствует длине упорядоченной части массива и все «нули» в написанном фрагменте нужно заменить на **j**.

Два в одном – шампунь и бальзам-ополаскиватель. В некоторых достаточно простых случаях удастся решить задачу с помощью «исторического» подхода, когда программа заканчивается раньше, чем программист теряет логическую нить рассуждений. Но при этом довольно часто несколько независимых управляющих конструкций алгоритма оказываются «слитыми» в одну. Конечно, это делает программу более компактной, но не более воспринимаемой и управляемой. Простой пример: поиск подстроки в строке. «Исторический» подход. Берем по очереди **i**-й символ строки и проводим в цикле попарное сравнение его с **k**-м символом второй строки. Если они совпадают, то переходим к следующей паре (**k++** в самом теле цикла и **i++** в заголовке). Если не совпадают, то возвращаемся на начало второй строки и к следующему символу первой (от начала совпадающего фрагмента). Успешное завершение поиска – достижение конца второй строки.

```

int find(char c1[], char c2[]){
for (int k=0,i=0; c1[i]!=0; i++){
    if (c2[k]==0) return i-k;
        if (c1[i]==c2[k]) k++;
        else { i=k; k=0; }
    }
return -1; }

```

Не будем придираться. Программа работает, хотя процессы, происходящие в ее единственном цикле, можно обозначить как «возвратно-поступательные». Это видно из того, что индекс **i**, который определяет характер протекания цикла, меняется в самом этом цикле, да притом при выполнении определенного условия. С точки зрения понимания процесса «движения» программы по циклу это «не есть хорошо». Интуитивно ясно, что такое «движение» раскладывается на две составляющие: движение по первой строке и параллельное движение по второй и по первой строкам (относительно текущего символа первого цикла). То, что это не было замечено, характеризует «исторический» подход: мысль о необходимом «откате» возникла уже после описания процесса параллельного движения по строкам. Если же проектировать программу, то схематичное описание логики алгоритма выглядит так: для каждого символа первой:

произвести параллельное движение по обоим строкам до обнаружения расхождения. Если при этом мы остановимся на конце второй строки, то фрагмент найден и функция должна быть завершена, иначе процесс продолжается.

```
int find(char c[]){
for (int i=0; c[i]!=0; i++)
{
// 1. Попарное сравнение символов c2 - от начала и c1 - от i.
// 2. Если достигли конца c2 - выйти и вернуть i
}
return -1; }
```

Понятно, что «исторически» достигнуть условия в пункте 2 довольно трудно. Логически же две эти конструкции конкретизируются «в легкую» с использованием общей переменной – индекса **k**.

```
int find(char c[]){
for (int i=0; c[i]!=0; i++)
{
// 1. Попарное сравнение символов c2 - от начала и c1 - от i.
for (int k=0; c2[k]!=0 && c1[i+k]==c2[k]; k++);
// 2. Если достигли конца c2 - выйти и вернуть i
if (c2[k]==0) return i;
}
return -1; }
```

Иногда это удается. В простейших случаях удастся довести «исторический» процесс программирования до конца и получить даже более компактный код. В качестве примера рассмотрим программу поиска наименьшего общего кратного для массива значений. «Историк» будет рассуждать так:

1. Установим начальное значение делителя, равное первому элементу массива, и начнем просматривать массив.

```
int F(int A[], int n)
{ int NOK=A[0];
for (int i=0; i<n; i++)...
```

2. Если **NOK** делится на очередной элемент, переходим к следующему.

```
if (NOK % A[i]==0) continue;
```

3. Иначе нужно увеличить **NOK** на 1 и повторить просмотр с первого элемента.

```
NOK++; i=-1;
```

4. Если цикл дойдет до конца, то текущее значение **NOK** и будет наименьшим общим кратным. Собрав все «до кучи» и убрав лишние ветви, получим


```

int F(int A[], int n)
{ int NOK=A[0];
  for (int i=0; i<n; i++)
    if (NOK % A[i]!=0)
      { NOK++; i=-1; }
return NOK; }

```

Для разработанной программы характерно «возвратно-поступательное» изменение индекса **i**. Это происходит потому, что в теле цикла индекс, регулярно изменяемый в заголовке, периодически сбрасывается. Таким образом, внешне «правильный» цикл ведет себя не так, как это обозначено в заголовке. А это «не есть хорошо». Этот процесс можно разбить на два последовательных, вложенных друг в друга процесса (цикла): внешний перебирает возможные значения **NOK**, а внутренний – элементы массива с целью определения условия делимости текущего **NOK**.

```

int F(int A[], int n)
{
for (int NOK=A[0];1;NOK++)    // Последовательно проверять NOK
{
  for (int i=0; i<n; i++)      // Последовательно проверять делимость
    if (NOK % A[i]!=0) break;
  if (i==n) break;           // Все поделились – выход, можно return
}
return NOK; }

```

НЕСКОЛЬКО СЛОВ В ЗАЩИТУ ВОСХОДЯЩЕГО ПРОГРАММИРОВАНИЯ

Способ разработки программ от самой внутренней конструкции к внешней – менее тяжкий грех, чем «исторический» подход. Иногда восходящий подход проявляет себя в модульном проектировании программ: если вы видите, что некоторая частная задача непременно будет присутствовать в вашем алгоритме, то ее можно реализовать в виде отдельного модуля (функции) уже на первом этапе проектирования.

Основная проблема восходящего проектирования: внутренняя, частная задача, будучи реализована в каком-то одном варианте, при обрамлении ее внешними конструкциями выполняется (однократно или с повторами) уже в других условиях, параметрах и ограничениях, которые необходимо включить в уже разработанный фрагмент, то есть изменять уже написанный код. Как избежать этого? По возможности проектировать фрагмент для наиболее общих условий, с учетом наибольшего числа внешних параметров.

Алгоритмы сортировки довольно убедительно демонстрируют особенности восходящего проектирования. Тем более что сам принцип упорядочения реализуется внутренним циклом, а внешний цикл его просто повторяет заданное число раз.

Сортировка погружением заключается в помещении очередного элемента в уже упорядоченную часть массива следующим образом: первоначально он помещается в конец упорядоченной последовательности, а затем в цикле перемещения к началу меняется местами с предыдущим, пока «не достигнет дна» либо не встретит элемент, меньше себя. Цикл погружения нового элемента следует написать для произвольной длины упорядоченной части **k** (элементы массива от 0 до **k-1** упорядочены).

```
int A[...];          // Массив, в котором хранится упорядоченная часть
int k;              // Размер упорядоченной части
int v;              // Новое значение
A[k]=v;
for (int i=k; i>0; i--){          // Погружение в обратном направлении
    if (A[i]>A[i-1]) break;        // Встретил меньшего себя
    int c=A[i]; A[i]=A[i-1]; A[i-1]=c; // Погружение – обмен с предыдущим
}
```

Этот фрагмент нужно включить в основной цикл сортировки, который повторяет процесс погружения для всех подряд элементов входного массива. Во-первых, исходный и упорядоченный массивы можно совместить в одном: неупорядоченная часть будет расположена справа, а **A[k]** – очередной элемент из этой части. Во-вторых, сортировку можно начинать с **k=1** – погружение первого элемента в упорядоченную часть, состоящую из единственного элемента **A[0]**.

```
//-----14-01.cpp
//----- Сортировка погружением
void sort(int A[], int n){      // Сортировка для A размерности n
for (int k=1; k<n; k++){
    for (int i=k; i>0; i--){ // Погружение в обратном направлении
        if (A[i]>A[i-1]) break; // Встретил меньшего себя
        int c=A[i]; A[i]=A[i-1]; A[i-1]=c;
    } // Погружение - обмен с предыдущим
}}

```

То, что фрагмент погружения удалось «вшить» в основной цикл практически без изменений, объясняется тем, что он был продуман для произвольной размерности упорядоченной части.

1.5. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

ГЛАВА I. СУТЬ ДЕДУКТИВНОГО МЕТОДА ХОЛМСА. Шерлок Холмс взял с камина пузырек и вынул из аккуратного сафьянового несесера...

А. Конан-Дойль. Знак четырех

ОТ ОБЩЕГО К ЧАСТНОМУ

Мы уже выяснили, что проектирование программы заключается не в одних формальных рассуждениях и в записи их на языке программирования, но включает в себя и образное представление программы, выделение в ней известных программисту составляющих, представленных стандартными программными контекстами, выстраивание их в определенном порядке, выполнение этой процедуры в виде последовательности шагов, приближающих нас к результату. В эту картину нужно добавить правильное, но самое трудное направление «движения» при построении программы – от общего к частному. И тогда получим примерно такую картину (рис. 1.10).

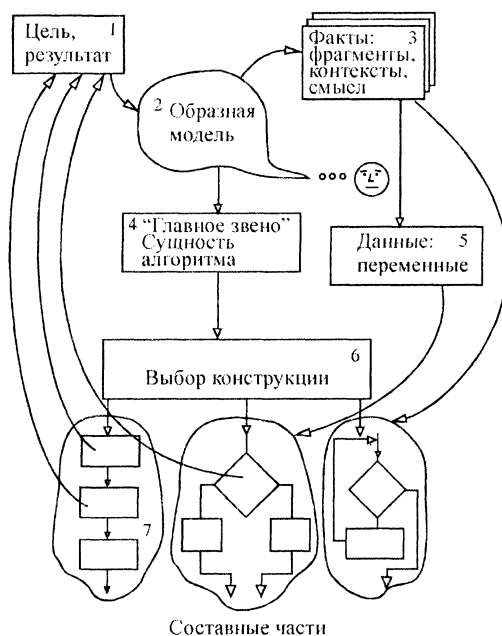


Рис. 1.10

1. Исходным состоянием процесса проектирования является более или менее точная формулировка цели алгоритма или результата, который должен быть получен при его выполнении. Формулировка, само собой, производится на естественном языке.

2. Создается образная модель происходящего процесса, используются графические и какие угодно способы представления, образные «картинки», позволяющие лучше понять выполнение алгоритма в динамике.

3. Выполняется сбор фактов, касающихся любых характеристик алгоритма, затем – попытка их представления средствами языка. Такие факты – это наличие определенных переменных и их «смысл», а также соответствующие им программные контексты. Понятно, что не все факты удастся сразу выразить в виде фрагментов программы, но они должны быть сформулированы хотя бы на естественном языке.

4. В образной модели выделяется самая существенная часть – «главное звено», для которой подбирается наиболее точная словесная формулировка.

5. Определяются переменные, необходимые для формального представления данного шага алгоритма, и формулируется их «смысл».

6. Выбирается одна из конструкций – простая последовательность действий, условная конструкция или цикл. Составные части выбранной формальной конструкции (например, условие, заголовок цикла) должны быть переписаны в словесной формулировке в терминах цели или результата.

7. Для оставшихся неформализованных частей алгоритма (в словесной формулировке) перечисленная последовательность действий повторяется.

Здесь много непривычного (рис. 1.11). Во-первых, на любом промежуточном шаге программа состоит из смеси конструкций языка, соответствующих пройденным шагам проектирования, и словесных формулировок, соответствующих еще не раскрытым вложенным конструкциям нижнего уровня. Во-вторых, процесс заключается в последовательной замене словесных формулировок конструкциями языка. На каждом шаге в программу добавляется всего одна конструкция, а содержимое ее составных частей снова формулируется в терминах «цель» или «результат». В третьих, «свобода выбора» ограничена тремя управляющими конструкциями языка: последовательностью действий, ветвлением или циклом.

При этом даже не принципиален конкретный синтаксис оператора, важен лишь вид конструкции, например, что это цикл, а не последовательность.

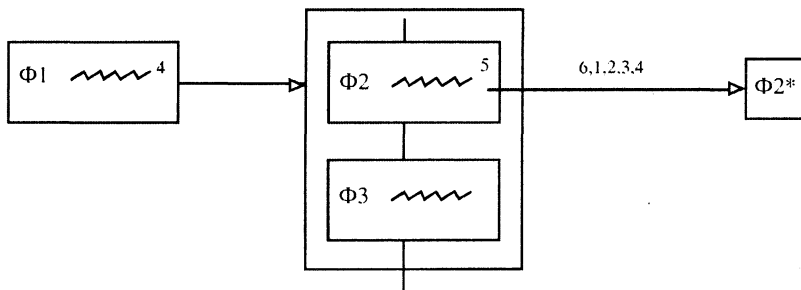


Рис. 1.11

Как и любая технология, структурное проектирование задает лишь «правила игры», но не гарантирует получение результата. Основная проблема – выбор синтаксической конструкции и замена формулировок – все равно технологией формально не решается. И здесь находится камень преткновения для начинающих программистов. «Главное звено» – это не столько особенности реализации алгоритма, которые всегда на виду и составляют его специфику, сколько действие, которое включает в себя все остальные. То есть все равно программист должен «видеть» в образной модели все элементы, отвечающие за поведение программы, и выделять из них главный, в смысле – самый внешний, или объемлющий. Единственный совет: постараться извлечь из образной модели как можно больше фактического материала.

ЗАПОВЕДИ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Обычно технология структурного программирования формулируется в виде «заповедей», о содержательной интерпретации которых уже легко догадаться.

1. Нисходящее проектирование.
2. Пошаговое проектирование.
3. Структурное проектирование (программирование без **goto**).
4. Одновременное проектирование алгоритма и данных.
5. Модульное проектирование.
6. Модульное, нисходящее, пошаговое тестирование.

Нисходящее пошаговое структурное проектирование.

В структурном программировании достаточно сложно отделить друг от друга принципы нисходящего, пошагового и структурного проектирования, поскольку каждый из них по отдельности достаточно тривиальный, а весь эффект состоит в их совместном использовании в рамках процесса проектирования:

- нисходящее проектирование программы – это процесс формализации от самой внешней синтаксической конструкции алгоритма к самой внутренней, движение от общей формулировки алгоритма к частной формулировке составляющего его действия;

- структурное проектирование заключается в замене словесной формулировки алгоритма на одну из синтаксических конструкций – последовательность, условие или цикл. При этом синтаксическая вложенность конструкций соответствует последовательности их проектирования и выполнения. Использование оператора перехода **goto** запрещается из принципиальных соображений;

- пошаговое проектирование состоит в том, что на каждом этапе проектирования в текст программы вносится только одна конструкция языка, а составляющие ее компоненты остаются в неформальном, словесном описании, что предполагает аналогичные шаги в их проектировании.

Нисходящее пошаговое структурное проектирование алгоритма представляет собой движение «от общего к частному» в процессе формулировки действий, выполняемых программой. В записи алгоритма это соответствует движению от внешней (объемлющей) конструкции к внутренней (вложенной) и конкретно выражается в том, что любая словесная формулировка действий (алгоритма) может заменяться на одну из трех формальных конструкций языка программирования:

- простая последовательности действий (блок);
- конструкция выбора (условный оператор);
- конструкция повторения (оператор цикла).

Выбранная формальная конструкция представляет собой часть процесса перевода словесного описания алгоритма на формальный язык. Естественно, что эта конструкция не определяет полностью всего содержания алгоритма. Поэтому составными ее частями остаются словесные формулировки более конкретных (вложенных) действий. В результате проектирования получается программа, в которой принципиально отсутствует оператор перехода **goto**, по-

этому структурное программирование иначе называется как **программирование без goto**.

Другое достоинство нисходящего проектирования: при обнаружении «тупика», то есть ошибки в логических рассуждениях, можно вернуться на несколько уровней вверх и продолжить процесс проектирования в другом направлении.

Одновременное проектирование алгоритма и структур данных. При нисходящей пошаговой детализации программы необходимые для работы структуры данных и переменные появляются по мере перехода от неформальных определений к конструкциям языка, то есть процессы детализации алгоритма и данных идут параллельно. Это касается прежде всего отдельных локальных переменных и внутренних параметров. С самой же общей точки зрения предмет (в нашем случае – данные) всегда первичен по отношению к выполняемым с ним действиям (в нашем случае – алгоритм). Поэтому на деле способ организации данных в программе более существенно влияет на структуру ее алгоритма, чем что-либо другое, и процесс проектирования структур данных должен опережать процесс проектирования алгоритма их обработки.

Нисходящее пошаговое модульное тестирование. Кажется очевидным, что отлаживать можно только написанную программу. Но это далеко не так. Разработка программы по технологии структурного программирования может быть произведена не до конца. На нижних уровнях можно поставить «заглушки», воспроизводящие один и тот же требуемый результат, можно обойти в процессе отладки еще не написанные части, используя ограничения во входных данных. То же самое касается модульного программирования. Можно проверить уже разработанные функции на тестовых данных. Сказанное означает, что отладка программы производится в некоторой степени параллельно с ее детализацией.

ОДНО ИЗ ТРЕХ

Обратим внимание на некоторые особенности процесса, которые остались за пределами «заповедей» и которые касаются содержательной стороны проектирования.

Цель (результат) = действие + цель (результат). Каждый шаг проектирования программы заключается в том, что словесная формулировка алгоритма заменяется на одну из трех возможных конструкций языка, элементы которой продолжают оставаться в неформальной, словесной формулировке. Однако это всего лишь внешняя сторона. Рассмотрим этот процесс подробнее.

1. Первоначальная формулировка шага алгоритма (например, **Ф1: Сделать «что-то» с массивом А размерности n**) определяется обычно в терминах цели работы фрагмента, или ее результата. В данном случае «сделать что-то с массивом» – это получить массив с заданными свойствами. На этом этапе работает образная модель процесса. Используя ее, а также накопленные факты, программист переходит к следующей формулировке.

2. Следующая формулировка (например, **Ф1а: Для каждого элемента массива выполнить проверку и, если нужно, сделать «что-то» с ним**) только на первый взгляд представляет собой детализацию действия, предусмотренного предыдущей формулировкой. На самом деле это и есть программирование. Происходит важный переход, который осуществляется в голове программиста и не может быть формализован, – переход от цели (результата) работы программы к последовательности действий, которые этого результата достигают. То есть алгоритм уже формулируется, но только с использованием образного мышления и естественного языка.

3. Далее для детализированной формулировки выбирается одна из трех логических конструкций алгоритма: линейная последовательность действий, ветвление (условная) или повторение (циклическая). Основное правило: конструкция должна полностью «покрывать» формулировку, точнее, соответствовать самой внешней логической конструкции алгоритма. В нашем примере – это цикл для каждого элемента массива. Выбрав его, мы получим примерно такую перефразировку: **Ф1б: Цикл для каждого элемента массива: выполнить проверку и, если нужно, сделать «что-то» с ним**. Первая часть соответствует заголовку цикла, вторая – его телу. Заметьте, что фразы «если нужно», «сделать что-то» – тоже формулировки частей будущего алгоритма, но на этом шаге они попадают на второй план, то есть не существенны.

4. Для формального представления конструкции языка необходимо выбрать переменные, определенные на предыдущих шагах, а также найти переменные, которые будут характеризовать текущую конструкцию. Для цикла, работающего с массивом, – это индекс текущего элемента **i**.

5. И только теперь можно приступить к почти механической замене частей формулировки синтаксическими элементами языка. Оставшаяся часть формулировки, выходящая за рамки заголовка выбранной конструкции, попадает в своем первоначальном виде: **for(int i=0; i<n; i++) { Ф2: выполнить проверку и, если нужно,**

сделать «что-то» с ним }. Она становится основой для следующего шага детализации. В нашем случае она уже тоже выглядит как словесная формулировка алгоритма. Это значит, что на предыдущем шаге мы забежали немного вперед. Если в полученной формулировке нет необходимой ясности, нужно перефразировать ее, представить в виде «цель – результат» и провести следующий шаг по всем правилам, начиная с самого начала.

Последовательность действий, связанных результатом. Многие почему-то считают, что основа логики сложной программы – условные и циклические действия. Понятно, что они определяют лицо программы. Но на самом деле чаще всего используется простая последовательность действий. Поэтому первое, что необходимо

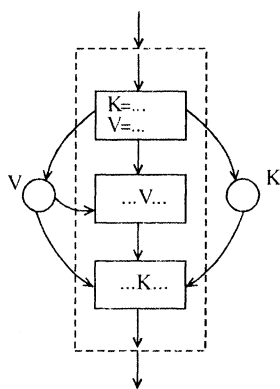


Рис. 1.12

сделать на очередном шаге детализации алгоритма, – проверить, нельзя ли его представить в виде последовательности шагов «делай раз, делай два...». Во-вторых, между различными шагами существуют связи через общие переменные: предыдущий шаг формирует значение переменной, придавая ей определенный «смысл», последующие шаги ее используют (рис. 1.12). Это обязательный элемент проектирования, без него нельзя продвигаться дальше в детализации выделенных шагов.

О том, какая конструкция должна быть выбрана на следующем шаге детализации, можно судить и по внешнему виду формулировки. Другое дело, что эта формулировка должна как можно точнее отражать сущность алгоритма и, что самое главное, «покрывать» его целиком, не оставляя не оговоренных действий:

- если в формулировке присутствует набор действий, объединенных соединительным союзом **И**, то речь скорее всего идет о последовательности действий. Например, шаг сортировки выбором: выбрать минимальный элемент **И**, перенести его в выходную последовательность **И**, удалить его из входной путем сдвига «хвоста» последовательности влево;

- когда в формулировке присутствует слово **ЕСЛИ**, речь идет об условной конструкции (конструкции выбора);

- если в формулировке присутствуют обороты типа **«ДЛЯ КАЖДОГО... ВЫПОЛНИТЬ»** или **«ПОВТОРЯТЬ...ПОКА»**, речь идет о циклической конструкции.

То, что программист должен не просто проверять возможность применения последовательности действий, но отдавать ей предпочтение, проиллюстрируем продолжением предыдущего примера. Как поступать далее с формулировкой: **Ф2: выполнить проверку и, если нужно, сделать «что-то» с ним**. Можно выбрать в качестве основной фразу «если нужно», подразумевая, что она включает в себя действия, связанные с проверкой. А можно считать, что речь идет о последовательности действий, связанных переменной-признаком, первое из которых проверяет условие и устанавливает признак, а второе – проверяет признак и при его наличии выполняет «что-то» с элементом массива.

Ф2а: последовательность действий

Ф31: Выполнить проверку, установить k

Ф32: Если $k=1$, сделать «что-то» с $A[i]$

Преимущества шага в этом направлении: достаточно сложные действия, связанные с проверкой, вынесены в отдельную, синтаксически независимую часть программы.

И последнее достоинство: шаги последовательности действий, после того как они определены, могут конкретизироваться в любом порядке, например, «по линии наименьшего сопротивления» от простых к более сложным.

«Среда обитания» программы. Каждая конструкция языка не просто встраивается в программу, а определяет свойства используемых ею данных, «смысл» переменных, которые появились в программе одновременно с ней. Поэтому при использовании исключительно вложенных конструкций мы получим в каждой точке программы определенный набор выполняемых условий, своего рода «среду обитания» алгоритма (рис. 1.13). Эти переменные служат исходными данными для очередного шага детализации алгоритма.

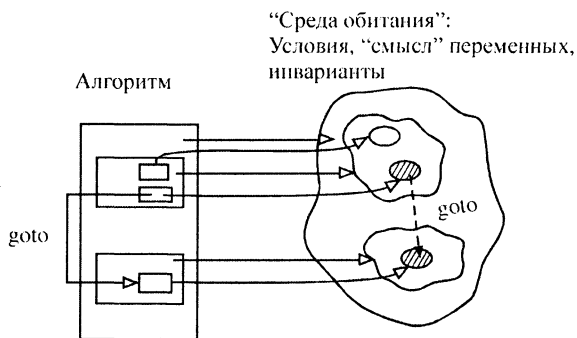


Рис. 1.13

ПРОГРАММИРОВАНИЕ БЕЗ GOTO

Почему «программирование без goto»? Нисходящее пошаговое проектирование исключает использование оператора **goto**, более того, запрещает его применение как нарушающего структуру программы. И дело здесь не в том, что «бог любит троицу» и трех основных логических конструкций достаточно. **Goto** страшен не тем, что «неправильно» связывает разные части алгоритма, а тем, что переводит алгоритм из одних условий в другие: в точке перехода они составлены без учета того, что кто-то сюда войдет «не по правилам».

Допустимые случаи использования goto. Чрезвычайными обстоятельствами, вынуждающими прибегнуть к помощи оператора **goto**, являются глобальные нарушения логики выполнения программы, например, грубые неисправимые ошибки во входных данных. В таких случаях делается переход из нескольких вложенных конструкций либо в конец программы, либо к повторению некоторой ее части. В других обстоятельствах его использование свидетельствует скорее о неправильном проектировании структуры программы – наличии неявных условных или циклических конструкций (см. в разделе 1.4 «Историческое» программирование). Пример правильного использования **goto**:

```
retry: for(...) { for (...)  
    {  
        if () goto retry;... // Попытаться сделать все сначала  
        if () goto fatal; } // Выйти сразу же к концу  
    }  
fatal:
```

Все равно, при использовании оператора перехода нужно изменить условия текущего выполнения программы применительно к точке перехода, например, переоткрыть файлы, установить начальные (заключительные) значения переменных.

Операторы continue, break и return. Чаше встречаются случаи более «мягкого» нарушения структурированной логики выполнения программы, не выходящие за рамки текущей синтаксической конструкции: цикла или функции. Они реализуются операторами **continue**, **break**, **return**, которые рассматриваются как ограниченные варианты **goto**:

continue – переход в завершающую часть цикла;

break – выход из внутреннего цикла;

return – выход из текущего модуля (функции).

```

void    F(){
for (i=0; i<n; m1: i++)
    {
        if (A[i]==0) continue;    //goto m1;
        if (A[i]==-1) return;     //goto m3;
        if (A[i] <0) break;       //goto m2;
    }
m2:    ... продолжение тела функции
m3:    }

```

Хотя такие конструкции нарушают чистоту подхода, все они имеют простые структурированные эквиваленты с использованием дополнительных переменных – признаков.

```

for (i=0; i<n; i++)          // Выход по break при обнаружении
    { if (...a[i]...) break; ... } // элемента с заданным свойством
if (i==n) A else B           // Косвенное определение причин выхода

int found;                   // Эквивалент с признаком обнаружения элемента
for (found=0, i=0; i<n && !found; i++)
    { if (...a[i]...) found++; ... }
if (!found) A else B

```

При отсутствии в массиве элемента с заданным свойством выполняется **A**, в противном случае – **B**. Во втором фрагменте используется специальный признак для имитации оператора **break**.

ОСОБЕННОСТИ ВЫПОЛНЕНИЯ ОТДЕЛЬНЫХ ОПЕРАТОРОВ

Отметим некоторые синтаксические особенности операторов, связанные с логической структурой алгоритма.

Роль символа «;». Символ «;» ограничивает любое выражение, превращая его в оператор. При отсутствии ограничителя ошибка обычно «наводится» в последующей части программы, когда транслятор наконец-то догадывается, что это «уже не выражение». Конечно, здесь много зависит от особенностей транслятора, но чтобы не проверять его на «сообразительность», лучше приучить себя вовремя ставить этот ограничитель.

Выражение + «;» = оператор

Пустой оператор. Символ «;», встречающийся в программе, «сам по себе» обозначает пустой, бездействующий оператор. Пустой оператор используется там, где по синтаксису требуется наличие оператора, но никаких действий производить не нужно. Например, в цикле, где все необходимое делается в его заголовке.

```
for (i=0; i<n; i++) s = s + A[i];    // Обычный цикл
for (i=0; A[i]!=0 && i<n; i++);      // Цикл с пустым оператором
```

Последовательность операторов (блок). Основная логика алгоритма – отсутствие логики, то есть простая последовательность действий. Любая последовательность операторов, заключенная в фигурные скобки ({}), выступает в конструкции верхнего уровня как единая синтаксическая единица (блок).

Условия в операторах цикла. Условия во всех операторах цикла являются условиями продолжения цикла.

Синтаксические варианты тела цикла. Существуют три варианта реализации тела цикла:

1) цикл с пустым оператором, не содержащий тела, в котором все необходимые действия отражены в заголовке;

2) цикл с телом – единственным оператором (который тем не менее может иметь сколь угодно большую вложенность);

3) цикл с телом – блоком, последовательностью операторов, заключенных в скобки {}.

Типичная ошибка: после заголовка цикла второго или третьего вида «для надежности» ставится точка с запятой, которая превращает этот цикл в цикл с пустым оператором. В результате настоящее его тело выполняется один раз после завершения цикла, получившегося из заголовка.

```
int i,s,A[20];
for (s=0,i=0; i<20; i++); // Для надежности !!!
    s=s+A[i];              // Настоящее тело цикла
                          // Будет s=A[20] – один раз и неправильно
```

Особенности оператора switch. Оператор **switch** можно назвать множественным переходом по группе значений выражения. Он является сочетанием условного оператора и оператора перехода.

```
switch (n)                                // Эквивалент
{
    // if (n==1) goto m1;
    // if (n==2) goto m2;
    // if (n==4) goto m3;
    // goto md;
case 1:  n=n+2; break;                    // m1: n=n+2; goto mend;
case 2:  n=0;  break;                     // m2: n=0;  goto mend;
case 4:  n++;  break;                     // m3: n++;  goto mend;
default: n=-1;                           // md: n=-1;
}                                           // mend: ...
```

Вычисляется значение выражения, стоящего в скобках. Затем последовательно проверяется его совпадение с каждой из констант, стоящих после ключевого слова **case** и ограниченных двоеточием. Если произошло совпадение, то производится переход на

идушую за константой простую последовательность операторов. Отсюда следует, что если не предпринять никаких действий, то после перехода к **n**-й последовательности операторов будет выполнена **n+1**-я и все последующие. Чтобы этого не происходило, в конце каждой из них ставится оператор **break**, который в данном случае производит выход за пределы оператора **switch**. И наконец: метка **default** обозначает последовательность, которая выполняется «по умолчанию», то есть когда не было перехода ни по какой другой ветви.

Если несколько ветвей оператора **switch** должны содержать идентичные действия (возможно, с различными параметрами), то можно использовать общую последовательность операторов в одной ветви, не отделяя ее оператором **break** от предыдущих.

```
sign=0;           // Ветвь для значения с, равного '+',
switch (c){       // используется и предыдущей ветвью для значения '-'
case '-':         sign=1;
case '+':         Sum(a,b,sign); break;
}
```

ПРИМЕР ПРОЕКТИРОВАНИЯ. СОРТИРОВКА ВЫБОРОМ

Для начала рассмотрим пример, в котором само задание уже содержит описание образной модели. Сортировка выбором основана на выборе на очередном шаге минимального элемента из входной последовательности, на исключении его оттуда и перенесении в конец выходной последовательности. Предлагается найти минимальный элемент, извлечь из массива, сдвигая вправо все, находящиеся слева от него, и поместить его на освободившееся место в конце. Повторение этого действия приведет к тому, что в правой части будет накапливаться возрастающая последовательность элементов.

Образная модель. Сбор фактов. В образной модели сразу же бросаются в глаза действия, реализуемые стандартными программными контекстами. Кроме того, что они являются заготовками будущей программы, они дают нам связанные с ними переменные и определяют их «смысл». Другое дело, к ним нельзя относиться как к истинам в последней инстанции, некоторые их характеристики пока неизвестны, они окончательно проявятся только при выстраивании фрагментов.

1. Сортировка выбором базируется на выборе минимального из множества оставшихся. В нашей модели неупорядоченные элементы находятся в левой части массива (размерность этой части пока

неизвестна). Кроме того, нужно знать местонахождение элемента, то есть его индекс.

```
// k - индекс минимального элемента
for (i=k=0; i< граница неотсорт.части; i++)
    if (A[i] < A[k]) k=i;
```

2. Выбранный элемент необходимо сохранить в промежуточной переменной.

3. Для сдвига элементов на один влево также имеется стандартный программный контекст:

```
for (int i=a; i<b; i++) A[i]=A[i+1];
```

4. Выбранный элемент помещается в конец массива.

5. Процесс сортировки повторяющийся. Каждый его шаг подразумевает выполнение перечисленных действий, причем справа в массиве располагается отсортированная часть, а слева – оставшаяся исходная. На каждом шаге граница частей смещается влево.

Начало проектирования. В соответствии с принципами модульного проектирования программа представляет собой функцию, получающую все входные данные через формальные параметры. Массив передается по ссылке, то есть сортировка производится в нем самом.

```
//---- Сортировка выбором. Шаг 0
void sort(int A[], int n){
    Ф1: сортировать A[] выбором
}
```

Пошаговое нисходящее проектирование. Это простой пример, потому что внешняя конструкция прямо бросается в глаза. Сущность сортировки заключается в повторении выполнения одного и того же действия, шага сортировки, о чем в списке фактов говорит пункт 5.

```
//---- Сортировка выбором. Шаг 1.
void sort(int A[], int n){
    Ф1а: повторять шаг сортировки (п.5 из списка фактов)
}
```

Однако для записи цикла необходимо определить его параметр и содержательную интерпретацию – «смысл». Пусть это будет длина отсортированной части – i . Тогда длина неотсортированной части вычисляется как $n-i$ (понадобится в дальнейшем).

```
//---- Сортировка выбором. Шаг 1
void sort(int A[], int n){
    Ф1б: for(int i=0; i<n; i++){
        Ф2: шаг сортировки. i – длина отсортированной части
    }
}
```

Шаг сортировки включает в себя последовательность действий, перечисленных в пунктах 1–4 списка фактов. Поставленные «для надежности» фигурные скобки в теле цикла оказались кстати: синтаксически последовательность действий образует блок. Для связи шагов последовательности необходимо определить две переменные: индекс минимального элемента – **k** и извлеченное значение, хранимое в переменной **v**.

```
//---- Сортировка выбором. Шаг 2
void sort(int A[], int n){
Ф16: for(int i=0; i<n; i++){
    Ф2а: последовательность действий пп.1-4
    }
}

//---- Сортировка выбором. Шаг 2.
void sort(int A[], int n){
for(int i=0; i<n; i++){          // i – длина отсортированной части
    int k;                      // k – индекс минимального элемента
    int v;                      // v – сохраненное выбранное значение
    Ф3: найти min в неотсортированной части      // k ←
    Ф4: сохранить минимум в v                    // k → v ←
    Ф5: сдвинуть «хвост» влево                   // k, n →
    Ф6: записать сохраненный последним          // v, n →
    }
}
```

Дальнейшая формализация фрагментов – по линии наименьшего сопротивления. Для начала просто переведем «слова» в операции и операнды, используя «смысл» уже определенных переменных: сохранение и запись – присваивание, сохраненный – **v**, последний – **A[n-1]**, минимальный – **A[k]**.

```
Ф4: v=A[k];
Ф6: A[n-1]=v;
```

Для оставшихся фрагментов используются стандартные программные контексты, в которых в заголовках циклов поставлены необходимые границы. В каждом цикле используется своя рабочая переменная **j** – индекс текущего элемента.

```
jnt j;
Ф3: for(k=j=0; j<n-i; j++)      // До границы неотсортированной части
    if (A[j]<A[k]) k=j;
Ф5: for(j=k; j<n-1; j++)        // От минимального до конца
    A[j]=A[j+1];
```


Окончательный вариант:

```
//-----15-05.cpp
//---- Сортировка выбором. Окончательный вариант
void sort(int A[], int n){
for(int i=0; i<n; i++){           // i – длина отсортированной части
    int k;                       // k – индекс минимального элемента
    int v;                       // v – сохраненное выбранное значение
    int j;
    for(k=j=0; j<n-i; j++)       // Ф3
        if (A[j]<A[k]) k=j;
    v=A[k];                      // Ф4
    for(j=k; j<n-1; j++)         // Ф5
        A[j]=A[j+1];
    A[n-1]=v;                   // Ф6
}}
```

1.6. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Divide et impera (Разделяй и властвуй).

Латинская формулировка принципа империалистической политики, возникшая уже в новое время

Модульное проектирование – самая очевидная вещь в технологии программирования. Тем более, что любая промышленная технология производства сложных изделий рано или поздно приходит к сборке их из набора совместимых и взаимозаменяемых деталей. Никому не надо объяснять термин «интерфейс». Но совсем не просто соблюдать эту заповедь: разрабатывать модульные программы. Отчасти это происходит потому, что взаимодействие модулей в программе несколько отличается от их взаимодействия в другой технической системе.

ОСОБЕННОСТИ ФУНКЦИИ КАК МОДУЛЯ

В чем разница между модулями (функциями) в программе и модулями в другой технической системе, например, в автомобиле. Там и здесь речь идет о завершенных изделиях, имеющих стандартные интерфейсы соединения модулей (например, шланг подачи бензина или провод подключения аккумулятора в силовом агрегате автомобиля) (рис. 1.14). Но в конкретной технической системе модули соединяются раз и навсегда, а в интерфейсах протекают непрерывные процессы: по бензопроводу подается горючее, а от аккумулятора – напряжение. Все модули работают непрерывно и параллельно. В программных модулях в каждый момент времени

выполняется одна функция (**F**). Если в теле функции **F** в выражении встречается **вызов** – имя другой функции (**G**), то между ними устанавливается временная связь: выполнение первой функции прекращается до тех пор, пока не выполнится вторая. Этот принцип выполнения называется **вложенностью вызовов функций** и может быть повторен многократно (рис. 1.15).

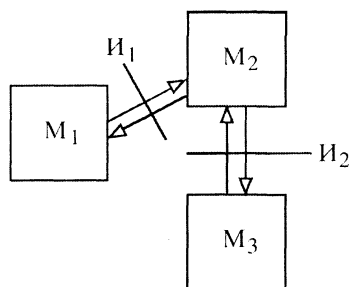


Рис. 1.14

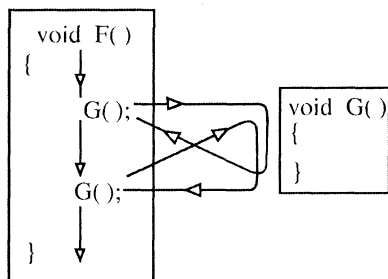


Рис. 1.15

Итак, первое, в чем нельзя ошибаться: функции синтаксически записываются как *независимые* модули, связи между ними устанавливаются через вложенные вызовы в процессе выполнения, то есть динамически (рис. 1.16).

Далее необходимо установить различие между формальными и фактическим параметрами. Прежде всего это два разных взгляда на программный интерфейс функции. Формальные параметры – это описание интерфейса изнутри. Оно дается в виде **определения переменных**, то есть описания

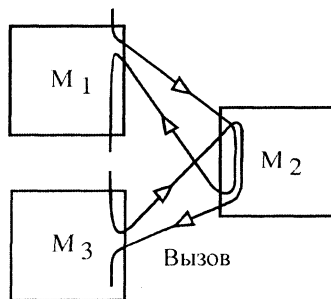


Рис. 1.16

свойств объекта, который может быть передан на вход. Имя формального параметра – это обобщенное (абстрактное) обозначение некоторой переменной, видимой в процессе работы функции *изнутри*. Например, функция обрабатывает абстрактный массив с именем **A** и размерностью **n**. При вызове функции в списке присутствуют фактические параметры, имеющие синтаксис **выражений**, то есть уже определенных переменных или промежуточных результатов, которые в данном вызове *ставятся в соответствие* формальным параметрам. Таким обра-

зом, они представляют взгляд на тот же самый интерфейс, но уже со стороны вызывающей функции (рис. 1.17).

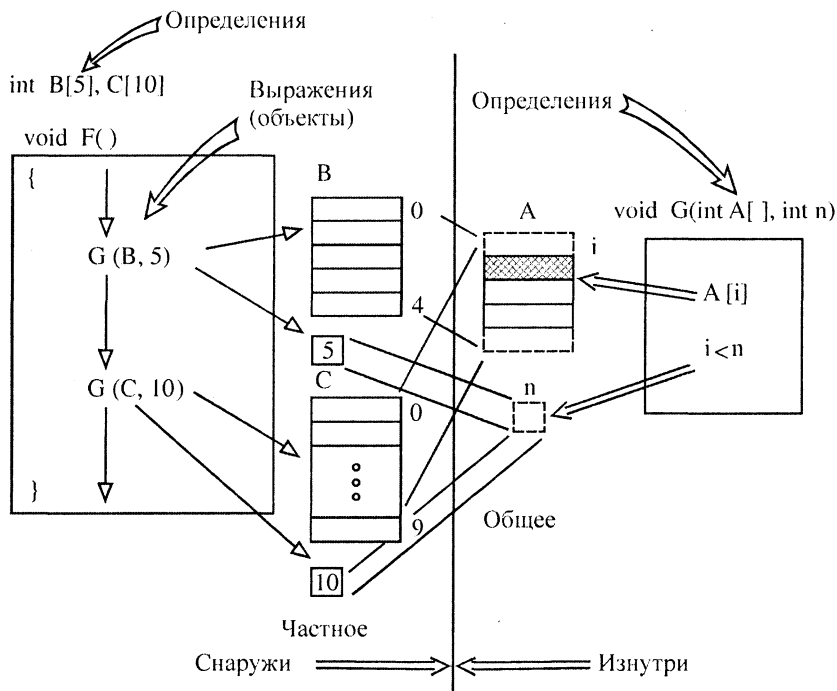


Рис. 1.17

Итак, формальные и фактические параметры имеют принципиально разный синтаксис: описания переменных (определения) и использования их (выражения). Связь между ними устанавливается в момент вызова динамически.

Главное, к чему необходимо привыкнуть: функция пишется для обработки данных *вообще*, то есть это обобщенное описание алгоритма для некоторых произвольных данных, имена которых представляют собой их «будущие обозначения» при работе функции. Что же касается транслятора, то для него формальные параметры — это «ожидаемые на входе значения», своего рода «заглушки», поэтому функция и транслируется применительно к имеющимся определениям (именам и типам).

Вызов функции, наоборот, представляет собой *частный* случай выполнения алгоритма для конкретных данных.

Рассмотренная модель может применяться сама к себе: реальная программа представляет собой иерархию вызовов функции, а формальные параметры функции верхнего уровня могут быть фактическими параметрами в следующем (вложенном) вызове.

Итак, главное необходимое условие модульного программирования – научиться абстрагироваться от конкретных обрабатываемых данных и выносить их «за пределы» проектируемого алгоритма.

По отношению к результату функции можно сформулировать те же самые принципы: результат – это обобщенное значение, которое возвращается после вызова функции в конкретное выражение, где расположен вызов.

Все здесь сказанное настроено на образное понимание того, что есть функция и *как* она вызывается. Насколько ваши представления соответствуют формальным определениям, можно убедиться, читая раздел 2.8.

МОДУЛЬНОСТЬ И СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММ

Разделение программы на модули позволяет преодолеть основное противоречие структурного программирования: процесс детализации программы состоит в движении от общего к частному, но в то же время наиболее очевидными являются, наоборот, фрагменты нижнего уровня. При усложнении программы технология пошагового проектирования сверху-вниз становится в тупик: слишком много фактов, причем внешние из них плохо просматриваются. Естественный выход: выделение из программы логически завершенных частей со строго определенным описанием их взаимодействия, каждая из которых допускает независимое проектирование.

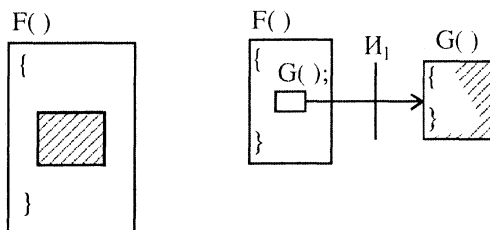
Модульность синтаксическая. Если выделенная часть программы оформляется в виде функции, то она видна «невооруженным глазом». Причем обе части программы – вызываемая функция и вызывающий ее модуль – после определения интерфейса (заголовка функции) могут проектироваться независимо и в любой последовательности. Вызываемая функция может быть также и отложена (рис. 1.18, *а*).

Замечание: если в процессе разработки алгоритма возникает непреодолимое желание повторить уже выполненную последовательность действий, возможны следующие варианты:

- выполнить **goto** к имеющемуся фрагменту (категорически не рекомендуется);

- повторить текст фрагмента в новом месте (не эффективно);
- оформить повторяющийся фрагмент в виде модуля с вызовом в двух точках программы (лучше всего).

а) Модульность синтаксическая



б) «Грязная» программа

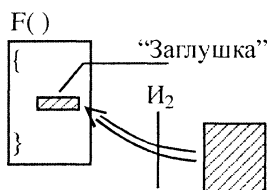


Рис. 1.18

Модульность и восходящее программирование. Возможность применения принципа модульности уже обсуждалась как разумная альтернатива восходящему проектированию. При попытке решения сложной задачи можно пойти по линии наименьшего сопротивления и выделить понятные части алгоритма, оформив их в виде модулей с соблюдением всех перечисленных принципов. Тогда оставшаяся часть задачи будет выглядеть значительно проще.

«Грязное» программирование. Под «грязным» программированием обычно понимается написание программы, грубо воспроизводящей требуемое поведение. Такая программа может быть быстро разработана и отлажена, а затем использована для уяснения последующих шагов либо для наложения «заплаток» для получения требуемого результата (рис. 1.18, б). Хотя это «не есть хорошо» с точки зрения технологии проектирования, но может быть оправдано при следующих условиях:

- «грязная» программа воспроизводит требуемое поведение на самом верхнем уровне;
- в дальнейшем в нее могут встраиваться контексты и фрагменты, не меняющие ее поведения, но конкретизирующие ее в нужном направлении.

Основным в «грязной» программе является соблюдение соотношений, которые она устанавливает в процессе своей работы. Эти соотношения необходимо сохранять при включении в программу новых фрагментов, они являются **инвариантами**.

Модульность формальная и истинная. Формально соблюдаемая модульность – синтаксическая: программа состоит из множества вызывающих друг друга функций (модулей), размер модуля ограничен определенным числом строк текста программы. Но не любая программа, разбитая на функции, будет модульной. Соблюдение духа, но не буквы модульного программирования, требует исполнения следующих принципов:

логическая завершенность. Функция (модуль) должна реализовывать логически законченный, целостный алгоритм;

ограниченность. Функция (модуль) должна быть ограничена в размерах, в противном случае ее необходимо разбить на логически завершенные части – модули, вызывающие друг друга;

замкнутость. Функция (модуль) не должна использовать глобальные данные, иметь «связь с внешним миром» помимо программного интерфейса, не должна содержать ввода и вывода результатов во внешние потоки – результаты должны быть размещены в структурах данных;

универсальность. Функция (модуль) должна быть универсальна, параметры процесса обработки и сами данные должны передаваться извне, а не подразумеваться или устанавливаться постоянными;

принцип «черного ящика». Функция (модуль) должна иметь продуманный «программный интерфейс» – набор фактических параметров и результат функции, через который она «подключается» к другим частям программы (вызывается).

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Суперпростое число. Число 1997 обладает замечательным свойством: само оно простое, простыми также являются любые разбиения его цифр на две части, то есть 1-997, 19-97, 199-7. Требуется найти все такие числа для заданного количества значащих цифр.

Поскольку проверка, является ли число простым, будет применяться многократно по отношению как к самому числу, так и к его частям, проверку, является ли заданное число простым, оформим в виде функции, применив тем самым принцип модульного программирования.

```
//-----16-01.cpp
//---- Функция проверки, является ли число простым
int PR(int a){
if (a==0) return 0;           // 0 это не простое число
for ( int n=2; n<a; n++)
    { if (a%n==0) return 0 ; } // Если делится, можно выйти сразу
return 1;}                   // Дошли до конца - простое
```

Дополнительная проверка «крайностей»: 1 – простое число, но для нее цикл ни разу не выполнится и будет возвращено значение «истина»; 0, вообще говоря, простым числом не является, поэтому должен быть «отсечен».

1. Сам алгоритм представляет собой полный перебор **n**-значных чисел. Прежде всего необходимо получить сам диапазон. Для этого 1 умножается в цикле **n** раз на 10. Верхняя граница – в 10 раз больше нижней. Полученные числа не сохраняются – просто выводятся.

```
void super(int n){
long v,a; int i;
for (v=1,i=0; i<n; i++) v*=10;           // Определение нижней границы
for (a=v; a<10*v; a++){
    // Проверить число на суперпростоту
    if (... суперпростое...) printf("%d\n",a);
}}
```

2. Фрагмент проверки – это скорее «технологическая заглушка», обозначающая общую логику процесса. Саму проверку удобнее произвести по принципу просеивания: если очередное условие не соблюдается, выполняется переход к следующему шагу цикла оператором **continue**. Первое условие, что само число является простым, проверяется вызовом функции. Сложнее проверить его части. Для получения частей необходимо рассмотреть частные и остатки от деления этого числа на 10, 100 и так далее до **v** – нижней границы диапазона. Если хотя бы одно частное или остаток из них не является простым, то все число также не является суперпростым. Грубо процесс проверки можно представить так:

```
if (PR(a)==0) continue;
for (long ll=10; ll<v; ll*=10){ // ll пробегает значения 10,100, 1000 < v
    ... PR(a/ll)...           // Проверка старшей части
    ... PR(a%ll)...           // Проверка младшей части
}
if (...все части простые...) printf("%d\n",a);
```

3. В предыдущем варианте мы отступили от принципа нисходящего проектирования, поскольку сначала требовалось сформулировать условие: проверить, являются ли все части числа простыми, из чего следует, что написанным вчерне процессом прове-

руется условие всеобщности. Для реализации процесса используется стандартный контекст с **break**. Если условие не соблюдается (то есть выполняется обратное), то происходит досрочный выход, тогда «естественное» достижение конца цикла по условию, стоящему в заголовке, говорит о справедливости свойства всеобщности.

```
//-----16-02.cpp
//----- Суперпростое число с n значащими цифрами
void super(int n){
long v,a; int i;
for (v=1,i=0; i<n; i++) v*=10;    // Определение нижней границы
for (a=v/10; a<v; a++){
    if (PR(a)==0) continue;
    for (long ll=10; ll<v; ll*=10){ // ll Пробегаем значения 10,100, 1000 < v
        if (PR(a/ll)==0)          // Проверка старшей части
            break;                // Не простое – досрочный выход
        if (PR(a%ll)==0)          // Проверка младшей части
            break;                // Не простое – досрочный выход
    }
    if (ll==v)                    // Достигли конца – все простые
        printf("super=%ld\n",a);
}
}
```

Сортировка Шелла. Использование стандартных функций, библиотек и известных решений также соответствует принципу модульности. Иногда удобнее модифицировать известный алгоритм, добавив к нему для большей универсальности дополнительные параметры, нежели разрабатывать всю программу «от нуля». Сортировка Шелла (см. раздел 2.5) использует любой стандартный алгоритм сортировки, основанной на обмене («пузырек», вставка погружением), но не во всем массиве, а в группе, начинающейся с элемента **k** с шагом **s**. Часть задачи можно решить, формально заменив в исходном алгоритме шаг 1 на **s** и начальный элемент 0 на **k**.

```
//-----16-03.cpp
//-----Сортировка методом "пузырька"
void sort(int A[], int n){
int i,found;                // Количество сравнений
do { found=0;               // Повторять просмотр...
    for (i=0; i<n-1; i++)
        if (A[i] > A[i+1]) { // Сравнить соседей
            int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
            found++;         // Переставить соседей
        }
    } while(found !=0); }    //...пока есть перестановки
//-----Сортировка методом "пузырька" с шагом s, начиная с k
void sort1(int A[], int n , int k, int s){
int i,found;                // Количество сравнений
do { found=0;               // Повторять просмотр...
    for (i=k; i<n-s; i+=s)
        if (A[i] > A[i+s]) { // Сравнить соседей (через s)
```



```

        int cc = A[i]; A[i]=A[i+s]; A[i+s]=cc;
        found++; // Переставить соседей
    }
} while(found !=0); } //...пока есть перестановки

```

В сортировке Шелла исходный массив разбивается на m частей, в каждую из которых попадают элементы с шагом m , начиная от 0, 1, ..., $m-1$ соответственно, то есть

```

0, m, 2m, 3m, ...
1, m+1, 2m+1, 3m+1, ...
2, m+2, 2m+2, 3m+2, ...

```

Каждая часть сортируется отдельно. Затем выбирается меньший шаг, и алгоритм повторяется. Шаг удобно выбрать равным степеням 2, например: 64, 32, 16, 8, 4, 2, 1. Последняя сортировка выполняется с шагом 1. Несмотря на увеличение числа циклов, суммарное число перестановок будет меньшим. Имея частичную сортировку в виде функции, нужно просто вызвать ее в теле двойного цикла.

```

//-----16-04.cpp
//----- Сортировка Шелла
void shell(int A[], int n){
for (int m=1; m<n; m*=2); // Определение последней степени 2
for (m/=2; m!=0; m/=2) // Цикл с переменным шагом m=32,16,8:
for (int k=0; k<m; k++) // Цикл по группам k=0:m-1
    sort1(A,n,k,m); }

```

«Грязное» программирование. Обработка строки. Функция заменяет в строке последовательность одинаковых символов на константу – счетчик и один такой символ (например, **qwertyaaaaaaaaaatybbbbbbbbb** – **qwerty12aty8bgg**).

«Грязная» программа моделирует основные свойства процесса обработки строки: за один шаг цикла просматривается один неповторяющийся символ или цепочка повторяющихся. Цикл просмотра цепочки является «заглушкой», заменяющей будущий процесс обработки. Инвариант – переменная цикла на каждом шаге должна устанавливаться на начало следующего фрагмента.

```

//-----16-05.cpp
//-----" Грязная" программа - просмотр повторяющихся цепочек
void proc(char c[]){
for (int i=0; c[i]!='\0'; i++) // 1 шаг - 1 символ или 1 цепочка
{
    if (c[i]!=' ' && c[i]==c[i+1])
    { // Заглушка
        putchar('*');
        while (c[i]==c[i+1]) i++;
    }
    else putchar(c[i]);
}
}
void main(){ proc("gfbvege aaaaaaaaaa ffffffff"); }

```

Достоинство такой программы – она может быть проверена и отлажена, хотя и бесполезна. Следующий шаг – замена «заглушки» на требуемый фрагмент. Он включает в себя последовательность действий:

- 1) определение длины последовательности **k** и установку индекса на ее последний символ **j**;
- 2) запись двух цифр счетчика в начало последовательности в символьном виде;
- 3) сохранение в строке одного символа из повторяющихся;
- 4) сдвиг «хвоста»;
- 5) установка индекса **i** на последний символ полученного фрагмента – с целью сохранения инварианта внешнего цикла.

```
//-----16-06.cpp
//----- Свертка цепочек повторяющихся символов
void proc(char c[]){
for (int i=0; c[i]!=0; i++){           // 1 шаг - 1 символ ???
{
if (c[i]!=' ' && c[i]==c[i+1])
{           // старая заглушка
// putchar('*'); while (c[i]==c[i+1]) i++;
// 1 - длина k
// Начало нужно - не трогаем i
// Конец фрагмента - j
for (int j=i, k=1; c[j]==c[j+1]; k++, j++);
// j - на последний из 'aaaaa'
// 2 - k - записать в c[] в виде 2 цифр
// i - сдвинуть так, чтобы он оказался там, где надо
if (k>=10) c[i++]=k/10+'0';
c[i++]=k%10+'0';
// 3 - оставить 1 символ - уже стоим там !!!!
// 4 - сдвинуть хвост - перенос с использованием 2 индексов
int i1;
for(j++, i1=i+1; c[j]!=0; j++, i1++) c[i1]=c[j];
c[i1]=0;
// 5 на конец полученного фрагмента уже стоим там !!!!
// свойство - i - на оставленном символе
// i++ => на следующий фрагмент
}
}
}

void main(){
char cc[]="gfbvege aaaaaaaaaa ffffffff";
proc(cc); puts(cc);}
```

1.7. ЛОГИЧЕСКОЕ И «ИСТОРИЧЕСКОЕ» В ПРОГРАММИРОВАНИИ

Анализируя поведение программы и разрабатывая ее, мы постоянно сталкивались с двумя противоположными взглядами на программу. «Исторический» взгляд состоит в анализе последова-

тельности выполняемых ею действий, траектории ее выполнения. При этом совсем не обязательно рассматривать ее работу с конкретными данными или использовать отладчик для ее трассировки. «Историческими» являются и абстрактные рассуждения примерно такого вида: «если на текущем шаге цикла условие истинно, а на следующем шаге ложно, то...». Наоборот, логический взгляд на программу или на ее отдельный фрагмент основан на проведении логического доказательства, убеждающего в том, что программа и ее фрагмент дают определенный результат при любых значениях входных переменных. Аналогично, если есть несколько стандартных программных контекстов с известными результатами их выполнения, то логический подход к анализу программы связан с их включением в цепочку логических рассуждений (базирующихся как на формальной логике, так и на здравом смысле), выводящих результат ее работы.

Естественно, что программист пользуется и тем и другим. Технология структурного программирования олицетворяет собой логический подход, образная модель – «исторический» взгляд на программу, основанный на представлении процесса ее выполнения.

«ИСТОРИЧЕСКИЙ» И ЛОГИЧЕСКИЙ ВЗГЛЯДЫ НА ЦИКЛ

Наиболее ярко «исторический» и логический взгляды на программу проявляются в проектировании циклов. «Историк» всегда пытается написать цикл для первого шага, а потом вносит изменения для последующих шагов, заканчивая обсуждением последнего шага. Логический подход основывается на проектировании шага цикла «вообще» как элемента повторяющегося процесса. С этой точки зрения приоритеты разработки цикла таковы:

- тело цикла;
- способ перехода к следующему шагу;
- начальное состояние и условия завершения цикла.

Важнее всего то, что цикл повторяется и как он это делает, а когда заканчивается – это уже частности. Если условие завершения сразу сформулировать не удастся, то можно написать «вечный цикл» с позднейшим включением альтернативного выхода.

```
for (int i=0; 1; ){           // 1- истина, повторять пока «истина», т.е. всегда
    ... if (что-то будет) break; ...
}
```

Инвариант цикла. Первое, что необходимо решить при проектировании цикла – выбрать, что является его шагом. Как только

это определено, в цикле появляется условие, которое сохраняется на протяжении всего цикла – **инвариант цикла**. Исходя из него, проектируется шаг цикла. В начале шага предполагается соблюдение этого условия. Шаг должен быть спроектирован так, чтобы по его окончании условие оказалось верным для следующего. Например, при работе с текстовой строкой выбирается инвариант: индекс **i** в массиве указывает на начало очередного слова. Тогда шаг цикла должен перемещать этот индекс от начала текущего к началу следующего слова.

```
//-----17-01.cpp
//----- Цикл пословной обработки : i - начало слова
void F(char c[]){
for (int i=0; c[i]!=' '; i++); // Начало первого слова для первого шага
while(c[i]!=0){               // Шаг цикла слово + цепочка пробелов
    for (;c[i]!=' ' && c[i]!=0; i++) // Обработка слова
        putchar(c[i]);
    for (;c[i]==' '; i++);          // Обработка цепочки пробелов
}
```

Если уж быть более точным, инвариантом цикла является утверждение: индекс указывает на начало очередного слова либо на конец строки.

Другой пример – обработка комментариев. Ограничители комментария представляют собой сочетания двух символов – «/*» или «*/». Если шаг цикла обрабатывает двухсимвольный ограничитель, то он должен корректировать на единицу переменную цикла. В противном случае соблюдается условие: 1 шаг – 1 символ.

```
//-----17-02.cpp
//----- Удаление комментариев из строки
void F(char c[]){
    int i,j,cm; // cm признак нахождения внутри комментария
    for (i=j=cm=0; c[i]!='\0'; i++) {
        if (c[i]=='*' && c[i+1]=='/') { cm--, i++; continue; }
        if (c[i]=='/' && c[i+1]=='*') { cm++, i++; continue; }
        if (cm==0) c[j++] = c[i];
    }
    c[j]=0; }
```

Наиболее показательно применение инварианта в итерационных циклах, в которых результат текущего шага напрямую зависит от результата предыдущего. Например, в алгоритме поиска значения в упорядоченном массиве методом половинного деления (двоичный поиск) инвариант – это интервал (**a**, **b**), на котором находится искомое значение. На каждом шаге цикла результатом является правая или левая половина интервала от предыдущего шага в зависимости от результата сравнения со значением в его середине (см. раздел 2.5 «Сортировка и поиск»).

Плюс-минус метр «от столба». Только после того, как шаг цикла спроектирован «вообще», необходимо поставить условия начала и завершения цикла. В них можно «промахнуться» в пределах одного шага до и после требуемого начального или конечного значения, что можно считать достаточно типичной ошибкой. Поэтому по окончании разработки цикла надо еще раз проверить, где он «стартует» и где «тормозится». Аналогичной проверке должны быть подвержены все альтернативные выходы из цикла.

```
//-----17-03.cpp
//---- Простая вставка
void sort(int in[], int n){
for ( int i=1; i < n; i++) {           // Для очередного i
    int v=in[i];                      // Делай 1 : сохранить очередной
    for (int k=0; k<i; k++)            // Делай 2 : поиск места вставки
        if(in[k]>v) break;            // перед первым, большим v
    for( int j=i-1; j>=k; j--)         // Делай 3: сдвиг на 1 вправо
        in[j+1]=in[j];               // от очередного до найденного
    in[k]=v;                          // Делай 4 : вставка очередного на место
}                                     // первого, большего него
```

В сортировке вставками при просмотре от начала массива (с индексом **k**) очередной элемент **v** вставляется перед первым, большим его самого. Для этого необходимо «освободить место» сдвигом вправо всех элементов в диапазоне от **k** до **i-1**. Стандартный контекст этой операции имеет вид **for(int j=...; j>...; j-) in[j+1]=in[j]**; Границы сдвига устанавливаются исходя из более детального рассмотрения начала и окончания процесса: первым шагом на освободившееся место, занятое текущим **in[i]**, должен быть помещен предыдущий. В соответствии с правилом сдвига **in[j+1]=in[j]** начальное значение **j=i-1** даст нам **in[i-1+1]=in[i-1]**. Последний сдвиг должен быть **in[k+1]=in[k]**, следовательно, **j>=k**.

Многообразие вариантов циклического процесса. Циклический процесс может быть по-разному запрограммирован, если в качестве шага цикла выбрать различные единицы структур обрабатываемых данных. От этого будут зависеть как инварианты циклов, так и наличие и число вложенных циклов. Например, обработка строки может вестись и посимвольно, и пословно (см. раздел 2.4 «Символы. Строки. Текст»).

ЖЕСТКАЯ И АВТОМАТНАЯ ЛОГИКА ПРОГРАММЫ

«Исторический» и логический элементы в программе проявляются не только в том, что ее поведение (последовательность выполнения действий) – это историческая сторона программы, а структура алгоритма, воспроизводящая это поведение, – логиче-

ская сторона. Мы оставили в стороне данные. Традиционное отношение к ним как к объекту обработки алгоритмом не исчерпывает их назначения. Данные в программе могут использоваться также для запоминания «истории ее работы», а это уже имеет отношение к ее логической стороне. Обычная «историческая» связь двух частей алгоритма **А–В** через проверяемые внешние условия ($k==...$) непосредственно отражена в логической структуре программы. Логика программы – последовательность операторов в значительной степени отражает историю ее работы: последовательно проверяются условия ($k==..., m==..., n==...$). Такую связь можно назвать связью через алгоритм.

Внутренние данные программы могут использоваться для запоминания происходивших при работе программы «событий», например, выполнения или невыполнения условий проверки внешних данных. Такие данные могут свидетельствовать о наличии «событий» как прямо (переменные-признаки), так и косвенно, через определенные свои значения (рис. 1.19). Так, проверка свойства всеобщности (см. раздел 1.3) заключается в том, что цикл прекращается либо по обнаружении невыполнения свойства на одном из элементов множества, либо по достижении его конца. Тогда по значению переменной – индекса по завершении цикла можно судить об истории его работы.

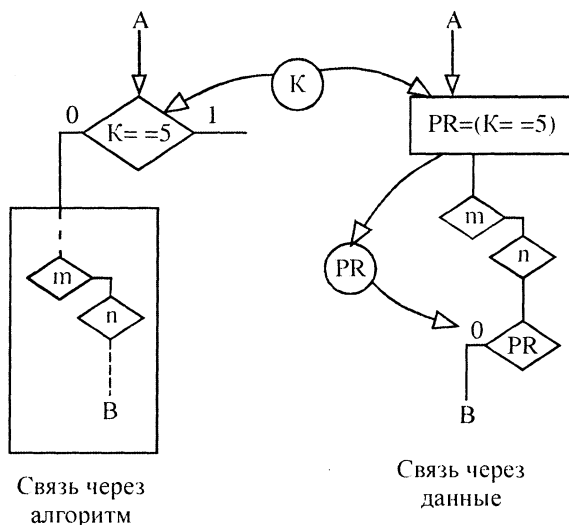


Рис. 1.19

Связь различных частей алгоритма через значения внутренних данных отражается в управляющей логике программы только косвенно. Увидеть ее можно, лишь анализируя «историческую» последовательность выполнения программы и значения переменных. Причем переменным должен быть присвоен «смысл», соответствующий характеру сохраняемых в них результатов. Часто они интерпретируются как различные *состояния программы*, в которые она переходит в зависимости от вида входных данных.

При переносе части логики алгоритма во внутренние переменные состояния значительно сокращается алгоритмическая составляющая программы. Доведя этот процесс до конца, можно получить программу, логика которой определяется ее внутренними данными (состояниями), что соответствует используемой как в математике, так и в прикладном программировании модели **конечного автомата**.

В качестве примера рассмотрим фрагмент, осуществляющий ветвление по комбинациям из трех условий. В обычной «исторической» логике он будет выглядеть так:

```
if (a<0)
    if (b<0)
        if (c<0)      x=5;    // ...1
        else          x=2;    // ...2
    else
        if (c<0) x=7;      // ...3
        else      x=6;    // ...4
    else ... и т.д.
```

Можно использовать переменную состояния, в которую каждое условие, принимающее значение 0/1, войдет со своим весом. Полученную переменную состояния можно обрабатывать более «регулярно», выполнив для нее множественное ветвление через **switch** либо используя как индекс для извлечения данных из массива значений.

```
int s=(a<0)*4 + (b<0)*2 + (c<0);
switch (s){
case 0: x=5; break;
case 1: x=2; break;
case 2: x=7; break;
case 3: x=6; break;
}
int v[]={5,2,7,6,...};
int s1=(a<0)*4 + (b<0)*2 + (c<0);
x=v[s1];
```



2. ПРОГРАММИСТ «НАЧИНАЮЩИЙ»

Содержание этой главы – «классика жанра» в области начального этапа практического программирования. Многие алгоритмы этого раздела были изобретены «еще в каменном веке», когда ограниченные ресурсы компьютера не позволяли развернуться ни в памяти, ни в скорости выполнения алгоритмов. Среди них есть, безусловно, феноменальные решения, позволяющие решить задачу при отсутствии для этого условий. Например, сортировка циклическим слиянием – это «сортировка без сортировки», позволяет упорядочить массив без перестановки его элементов, а только используя операции разделения и соединения (слияния) их последовательностей, находящихся в файлах.

Резонный вопрос начинающего программиста – зачем мне все это надо? Во-первых, эти алгоритмы в сильно концентрированном виде содержат изученный нами в предыдущем разделе «джентльменский набор», в других областях программирования гораздо больше «воды», и они гораздо менее показательны. Во-вторых, эти алгоритмы лежат под толстым слоем программного обеспечения в операционных системах, базах данных и т.д. В-третьих, зачастую изобилие ресурсов является виртуальным, то есть только кажущимся, а реальность далека от совершенства. В качестве примера рассмотрим поведение программы обычной сортировки, если она работает в виртуальной памяти и упорядочивает массив, размерность которого превышает объем физической памяти компьютера. Если в программе имеется внутренний цикл, который пробегает по всему массиву, то в тот момент, когда он достигнет конца массива, первые элементы окажутся «затертыми» (вытесненными) из физической памяти, поэтому следующий шаг цикла начнет все сначала – будет загружать весь массив из файла выгрузки. Учитывая, что диск работает на 2–3 порядка медленнее, мы получим из компьютера «кофемолку», работающую без явных признаков результата. В то же время есть алгоритмы, позволяющие выполнить сортировку по частям с последующим объединением результата и не приводящие к подобным эффектам.

И, наконец, последнее. Несмотря на грандиозный объем программного обеспечения, обработку строк текста по-прежнему приходится вести «своими руками». То же самое относится к особенностям представления текста, которые тут и там «всплывают» при

обработке данных, в основном при переходе от одной среды программирования и от одной операционной системы к другой. В связи с бурным развитием компьютеров, эти анахронизмы всплывают чаще, чем какие-либо иные.

2.1. АРИФМЕТИЧЕСКИЕ ЗАДАЧИ

В учебных задачах результат выполняемых действий сам по себе обычно непривлекателен.

Л. Венгер, А. Венгер. Готов ли ваш ребенок к школе?

Задачи, основанные на свойствах чисел, составляющих их цифр, – излюбленная тематика школьных олимпиад по программированию. Достоинство таких задач в том, что ничто лишнее не мешает созерцать принципы проектирования программ, стандартные программные контексты, да и стыдно сослаться на незнание арифметики.

Свойства делимости. Такие арифметические процедуры, как сокращение дробей, разложение числа на простые множители, определение наименьшего общего кратного, наибольшего общего делителя, поиск простых чисел, основаны на проверке свойств делимости чисел. Для этой цели используется операция получения остатка от деления «%», число делится на другое число, если остаток от деления равен 0. Нелишне напомнить, что все эти свойства определены для целых чисел, которым в Си соответствуют базовые типы данных `int` и `long`.

Работа с цифрами числа. То, что при выводе результата и при написании констант мы наблюдаем число, состоящее из цифр, еще ничего не значит, ибо это есть **внешняя форма представления числа** (см. раздел 2.4). Когда мы используем целую переменную, она представлена в памяти во внутренней (двоичной) форме. То, что с этой формой компьютером выполняются арифметические действия, можно считать «чудом» и не вникать, как он это делает. Отдельные же цифры числа можно получить, используя правила определения значения числа из цифр: вес следующей цифры десятичного числа в 10 раз больше текущей. Тогда остаток от деления числа n на 10 можно интерпретировать как значение младшей цифры числа, частное от деления на 10 – как отбрасывание младшей цифры числа, из чего составляется простой цикл получения цифр числа в обратном порядке. Выражение $s*10$ «дописывает» к числу 0 справа, а $s = s*10 + k$ добавляет к нему очередную цифру k .

Выражение	Интерпретация
$n \% 10$	Младшая цифра числа n
$n = n / 10$	Отбросить младшую цифру n
for ($i=0$; $n!=0$; $i++$, $n/=10$) { ... $n \% 10$... }	Получить цифры числа в обратном порядке
$s = s * 10 + k$	Добавить цифру k к значению числа s справа

Поиск полным перебором. Никогда не следует забывать, что основное достоинство компьютера состоит в возможности «тупого» перебора вариантов. Все арифметические задачи, ребусы, головоломки решаются путем полного перебора всех возможных значений чисел с выделением всех либо первого подходящего варианта. Как правило, такой цикл является внешним в программе поиска. Для поиска первого подходящего проверяемое условие может быть вынесено в заголовок цикла, но только в инверсном виде, поскольку любой цикл имеет в заголовке условие продолжения. Для поиска наименьшего из возможных перебор нужно производить в направлении увеличения проверяемых значений, для поиска наибольшего – в направлении уменьшения.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Счастливые билеты. «Счастливым» называется билет, в котором в шестизначном номере сумма первых трех цифр равна сумме последних трех. Решение строится на основе полного перебора всех шестизначных чисел. Каждое из них следует разложить на цифры, а затем сравнить суммы первых и последних трех. Как видим, решение складывается из стандартных фрагментов, нужно только выложить их в нужной последовательности «сверху вниз».

1. Исходные данные и результат. Функция возвращает целое – количество «счастливых» билетов. Формальных параметров нет. Основа алгоритма – полный перебор возможных билетов, то есть всех шестизначных чисел. Если число «счастливое» – увеличивается счетчик. Для определения свойства «быть счастливым» число необходимо разложить на цифры. Если они будут записаны в массиве, то условие легко записать: сумма первых трех элементов массива равна сумме трех последних.

```
int    happy(){
int    n;           // Количество «счастливых» билетов
long   v;           // Проверяемое шестизначное число
int    B[6];        // Массив значений цифр
for (n=0,v=0; v <= 999999; v++){
```

```

        // Разложить v в массив цифр числа - B
        if (B[0]+B[1]+B[2]==B[3]+B[4]+B[5]) n++;
    }
return n;}

```

2. Цифры числа получаются уже известным нам циклом деления числа на 10 с сохранением остатков в элементах массива (порядок не важен).

```

int    m;           // Номер шага (цифры)
long vv;           // Исходное число
for (vv = v, m=0; m<6; m++){
    B[m] = vv % 10; // Остаток - очередная цифра
    vv = vv / 10;   // Частное становится делимым
}

```

3. Окончательный вариант:

```

//-----21-01.cpp
//-----Счастливые билеты
long happy (){
    int m, B[6]; long v ,vv, n;
    for (n=0,v=0; v <= 999999; v++){
        for (vv = v, m=0; m<6; m++, vv /=10)
            B[m] = vv % 10;
        if (B[0]+B[1]+B[2]==B[3]+B[4]+B[5]) n++;
    }
    return n;}

```

Простые множители. Сформировать в массиве последовательность простых множителей заданного числа, ограниченную значением 0. Простые множители – простые числа, произведение которых дает заданное число, например: $72 = 2 \times 2 \times 2 \times 3 \times 3$.

1. Можно написать первый вариант программы, ничего принципиально не решив. Если предположить, что функция получает массив заданной размерности, который надо заполнить, и существует некоторый повторяющийся процесс, на каждом шаге которого получается очередной множитель, то первый вариант функции будет выглядеть так:

```

void    mnog(int val, int A[], int n) {
int      i;           // Количество множителей
int      m;           // Значение множителя
for (i=0; не кончился массив и есть множители; i++)
{
    // Получить очередной множитель m
    A[i] = m;
}
A[i]=0; }             // Ограничить последовательность

```

2. Получение очередного простого множителя. Простой множитель – минимальное простое число, на которое исходное делится без остатка. Если оно найдено (**m**), то на следующем шаге цикла

его нужно «исключить» из раскладываемого числа, то есть использовать вместо исходного числа **val** частное от деления его на **m**. Таким образом, для перехода к следующему шагу цикла нужно выполнить **val = val / m**. Процесс должен продолжаться пока **val** не обратится в 1.

```
void calc(int val, int A[], int n){
int m,i;
for (i=0; i<n-1 && val !=1; i++){
    // Получить минимальное простое число m, нацело делящее val
    val /= m; A[i] = m;
}
A[i] = 0;}
```

3. Минимальный простой множитель определяется обычным перебором значений, начиная с 2, пока не обнаружится делящееся нацело. Добавив цикл поиска, получим окончательный вариант.

```
//-----21-02.cpp
//-----Простые множители числа
void calc(int val, int A[], int n){
int m,i;
    for (i=0; i<n-1 && val !=1; i++){
        for (m=2; val % m !=0; m++);
        val /= m; A[i] = m;
    }
A[i] = 0;}
```

Простые числа. Сформировать массив простых чисел, не превышающих заданное число. Простое число – число, которое делится нацело только на 1 и на само себя.

1. Исходные данные и результат – формальные параметры функции – аналогичны параметрам в предыдущем примере. Сущность алгоритма состоит в проверке всех чисел от 2 до **val** и сохранении их в массиве, если они простые.

```
void calc(int val, int A[], int n){
int i; // Номер очередного простого числа
int m; // Очередное проверяемое число
for (i=0, m=2; i < n-1 && m < val; m++){
    {
        if (m - простое число)
            A[i++] = m;
    }
}
A[i] = 0;}
```

2. Конкретизируем утверждение, что **m** – простое число. Во-первых, оно не делится ни на одно число в диапазоне от 2 до **m/2** включительно. Во-вторых, что то же самое, оно не делится ни на одно простое число от 2 до **m-1**. Но эти простые числа накоплены предыдущими шагами цикла в массиве **A** от **A[0]** до **A[i-1]** включительно. Таким образом, число простое, если оно удовлетворяет

условию всеобщности: не делится ни на один элемент массива от 0 до $i-1$. Используем стандартный контекст с прерыванием цикла по нарушению проверяемого условия (число делится нацело на элемент массива) и проверяем свойство всеобщности как условие нормального завершения цикла (достижение конца заполненной части массива).

```
int n;
for (n=0; n < i; n++)
    if (m % A[n]==0) break; // Разделилось нацело
if (i==n)
    { ...m - простое число... }
```

3. Окончательный вариант:

```
//-----21-03.cpp
//-----Простые числа
void calc(int val, int A[], int n){
    int i,m,k;
    for (i=0, m=2; i < n-1 && m < val; m++) {
        for (k=0; k < i; k++)
            if (m % A[k]==0) break;
        if (i==k)
            A[i++] = m;
    }
    A[i] = 0;}
```

Несократимые дроби. При моделировании вычислений над несократимыми дробями вычисление общего знаменателя, сокращение дробей и другие действия производятся с использованием свойств делимости чисел. Так, функция умножения дробей для сокращения полученного произведения ищет наибольший общий делитель для числителя и знаменателя.

```
//-----21-04.cpp
//----- Умножение дробей
void sokr(int A[2],int B[2], int C[2]){
    C[0]=A[0]*B[0]; // A[0]-числитель, A[1]-знаменатель
    C[1]=A[1]*B[1];
    for (int n=C[0]; !(C[0]%n==0 && C[1]%n==0); n--);
    C[0]/=n; C[1]/=n;
}
```

Работа с датами. При вычислении дат основную сложность представляет неравномерность числа дней в месяцах. Решение любой задачи «в лоб» состоит в моделировании действия «перейти к следующему дню» с учетом всевозможных корректировок переходов к следующему месяцу и году. Например, функция, добавляющая к дате заданное количество дней, использует цикл, тело которого добавляет один день к текущей дате.

```
//-----21-05.cpp
//----- Добавить к дате заданное количество дней
void add_days(int A[3], int nd){ // A[0]-день, A[1]-месяц, A[2]-год.
static int days[]={ 0,31,28,31,30,31,30,31,31,30,31,30,31};
while(nd--){ // По числу добавленных дней
    A[0]++;
    if (A[0] > days[A[1]]) { // Выход за пределы месяца
        if ((A[1]==2) && (A[0]==29) && (A[2]%4==0))
            continue; // К 29 февраля високосного года
        A[0]=1; A[1]++; // К первому числа следующего месяца
        if (A[1]==13){ // К первому января следующего года
            A[1]=1; A[2]++; }
    }
}}
```

Головоломки. Все головоломки с подбором цифр решаются единообразно. Перебираются все числа в диапазоне поиска, в тело цикла вписываются проверки всех ограничений, которые видны в условии задачи: совпадение цифр, обозначенных буквами, несовпадение цифр, обозначенных разными буквами, появление заданных цифр на заданных позициях. Все, что «просеивается» через эти ограничения, и является решением задачи.

Число 512 обладает замечательным свойством: сумма его цифр в кубе равна самому этому числу. Требуется найти все числа с подобным свойством. Алгоритм поиска основан на полном переборе значений проверяемого числа **a**. В теле цикла подсчитывается сумма его цифр – **s**, а затем проверяется условие **s*s*s==a**, которое и соответствует проверяемому свойству.

```
//-----21-06.cpp
//----- Поиск числа, подобного 512 > (5+1+2)=8, 8 ^ 3= 512
void find(){
int a,n,k,s;
    for (a=10; a<30000; a++){
        for (n=a, s=0; n!=0; n=n/10)
            { k=n%10; s=s+k;}
        if (a==s*s*s) printf("%d^3=%d\n",s,a);
    }
}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

1. Найти в массиве и вывести значение наиболее часто встречающегося элемента.
2. Найти в массиве элемент, наиболее близкий к среднему арифметическому суммы его элементов.
3. Найти наименьшее общее кратное всех элементов массива (то есть число, которое делится на все элементы).
4. Найти наибольший общий делитель всех элементов массива (на который они все делятся без остатка).

5. Получить среднее между минимальным и максимальным значениями элементов массива и относительно этого значения разбить массив на две части (части не сортировать).

6. Задан массив, определить значение k , при котором сумма $|A(1)+A(2)+...+A(k)-A(k+1)-...-A(N)|$ минимальна (то есть минимален модуль разности сумм элементов в правой и левой частях, на которые массив делится этим k).

7. Заданы два упорядоченных по возрастанию массива. Составить из их значений третий, также упорядоченный по возрастанию (слияние).

8. Известно, что 1 января 1999 года – пятница. Для любой заданной даты программа должна выводить день недели.

9. Известно, что 1 января 1999 года – пятница. Программа должна найти все «черные вторники» и «черные пятницы» 1999 года (то есть – 13-е числа).

10. Найти в массиве наибольшее число подряд идущих одинаковых элементов (например, $\{1,5,3,6,6,6,6,3,4,4,5,5,5\} = 5$).

11. Составить алгоритм решения ребуса РАДАР= $(P+A+D)^4$ (различные буквы означают различные цифры, старшая – не 0).

12. Составить алгоритм решения ребуса МУХА+МУХА+МУХА = СЛОН (различные буквы означают различные цифры, старшая – не 0).

13. Составить алгоритм решения ребуса ДРУГ – ГУРД = 2727 (различные буквы означают различные цифры, старшая – не 0).

14. Составить алгоритм решения ребуса $4*ЛОТ + ТОЛ = ЛОТО$ (различные буквы означают различные цифры, старшая – не 0).

15. Несократимая дробь задана числителем и знаменателем – переменными типа **long**. Разработать функцию сложения дробей.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Содержательно сформулировать результат выполнения функции, определить «смысл» отдельных переменных, найти стандартные контексты, их определяющие, написать вызов функции.

Пример выполнения тестового задания

```
//-----21-07.cpp
//-----
int test(int a){
    int n,k,i;
    for (n=a; n!=0; n/=10){
        k=n%10;
        if (k==0) break;           // Цифра - 0
        if (k==1) continue;      // Цифра - 1
    }
}
```

```

    for (i=2; i<k; i++)
        if ( k%i==0) break;           // Цифра не простая
    if (k!=i) break;                   // Цифра не простая
}
if (n==0) return 1;                   // Дошли до конца без break
return 0; }                          // все цифры простые

#include <stdio.h>
void main(){
    printf("test(1357)=%d\n",test(1357));
    printf("test(1457)=%d\n",test(1457)); }

```

Функция возвращает логическое значение, то есть проверяет свойства числа **a**. Внешний цикл выделяет в нем последовательность цифр, очередная цифра хранится в переменной **k**. Внутренний цикл проверяет свойства делимости этой цифры. Причем как после внешнего, так и после внутреннего цикла проверяется условие «естественного» выхода из цикла; похоже проверяются свойства всеобщности или существования. Внутренний цикл, проверяющий цифру **k** на делимость в диапазоне от 2 до **k-1**, определяет, является ли **k** простым. Тогда оба цикла проверяют один из трех возможных вариантов: все цифры числа – простые, все цифры числа – не простые, в числе есть те и другие цифры.

Для точной подгонки результата нужно проследить цепочку операторов **break** и условий, при которых они выполняются. Самый внутренний **break** происходит при обнаружении делителя цифры (**k** – не простое). Следующий **break** происходит, если условие «естественного» выхода не было достигнуто, то есть был предыдущий **break** и выход из внешнего цикла происходит по тому же условию (**k** – не простое). Но последнее условие (**n==0**) является условием «естественного» завершения этого же цикла, то есть проверяется отсутствие предыдущего **break** по не простой цифре. Таким образом, функция проверяет, **все ли цифры числа являются простыми**. Поскольку цифры 0 и 1 внутренним циклом не проверяются, для них сделано исключение.

```

//-----21-08.cpp
//-----1
int F1(int n){
    for ( int i=2; n % i !=0; i++);
    return i; }
//-----2
int F2(int n1, int n2){
    for ( int i=n1; !(n1 % i ==0 && n2 % i ==0); i--);
    return i; }
//-----3
int F3(int n1, int n2){
    int i = n1; if (i < n2) i=n2;
    for (; !(i % n1 ==0 && i % n2 ==0); i++);
}

```



```

return i; }
//-----4
int F4(int a){
for ( int n=2; n<a; n++)
    { if (a%n==0) break; }
if (n==a) return 1;
return 0;}
//-----5
int F5(int a){
for ( int s=0,n=2; n<a; n++)
    { if (a%n==0) s++; }
if (s==0) return 1;
return 0;}
//-----6
int F6(int a, int b){
for ( int n=a; n%a!=0 || n%b!=0; n++);
return n; }
//-----7
int F7(int a, int b){
for ( int n=a; a%n!=0 || b%n!=0; n--);
return n; }
//-----8
int F8(int a){
int n,k,s;
for (n=a, s=0; n!=0; n=n/10)
    { k=n%10; s=s+k;}
return s;}
//-----9
int F9(int a){
int n,k,s;
for (n=a, s=0; n!=0; n=n/10)
    { k=n%10; if (k>s) s=k;}
return s;}
//-----10
int F10(int a){
int n,k,s;
for (n=a, s=0; n!=0; n=n/10)
    { k=n%10; s=s*10+k;}
return s;}
//-----11
void F11(){
int a,n,k,s;
    for (a=10; a<30000; a++){
        for (n=a, s=0; n!=0; n=n/10)
            { k=n%10; s=s+k;}
        if (a==s*s*s) printf("%d\n",a);
    } }
//-----12
void F12(int a, int A[10]){
int i,n;
for (i=0, n=a; n!=0; i++, n=n/10);
for (A[i--]=-1, n=a; n!=0; i--, n=n/10)
    A[i]=n % 10;
}
//-----13
void F13(int v, int A[], int m){
int i,n,a;

```

```

    for (i=0,a=2; a<v && i<m-1 ; a++){
    for (n=2; n<a; n++)
        { if (a%n==0) break; }
    if (n==a) A[i++]=a;
    }
A[i]=0; }
//-----14
void F14(int v, int A[], int m){
int i,n,a,j;
    for (i=0,a=2; a<v && i<m-1 ; a++){
    for (j=0; j<i; j++)
        { if (a%A[j]==0) break; }
    if (j==i) A[i++]=a;
    }
A[i]=0; }
//-----15
void F15(int val, int A[], int n){
int m,i;
    for (i=0; i<n-1 && val !=1; i++){
    for (m=2; val % m !=0; m++);
    val /= m; A[i] = m;
    }
A[i] = 0;}
//-----16
int F16(int c[], int n){
int i,j;
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
            if (c[i]==c[j]) return i;
    return -1; }
//-----17
int F17(int n)
{ int k,m;
for (k=0, m=1; m <= n; k++, m = m * 2);
return k-1; }
//-----18
void F18(int c[], int n)
{ int i,j,k;
for (i=0,j=n-1; i < j; i++,j--)
    { k = c[i]; c[i] = c[j]; c[j] = k; }
}
//-----19
int F19(int c[], int n)
{ int i,j,k1,k2;
    for (i=0; i<n; i++){
    for (j=k1=k2=0; j<n; j++)
        if (c[i] != c[j])
            { if (c[i] < c[j]) k1++; else k2++; }
    if (k1 == k2) return i;
    }
return -1; }
//-----20
int F20(int c[], int n)
{ int i,j,m,s;
    for (s=0, i=0; i < n-1; i++){
    for (j=i+1, m=0; j<n; j++)
        if (c[i]==c[j]) m++;

```

```

    if (m > s) s = m;
}
return s; }
//-----21
int F21(int c[], int n)
{ int i,j,k,m;
for (i=k=m=0; i < n-1; i++)
    if (c[i] < c[i+1]) k++;
    else {
        if (k > m) m=k;
        k=0; }
if (k > m) m=k;
return m; }

```

2.2. ИТЕРАЦИОННЫЕ ЦИКЛЫ И ПРИБЛИЖЕННЫЕ ВЫЧИСЛЕНИЯ

В большинстве циклов действия, производимые в теле цикла, не влияют на параметры его протекания: количество шагов, характеристики шага. В таких циклах параметры заголовка цикла не зависят от значений переменных, вычисляемых в теле цикла, и цикл имеет постоянное количество повторений, например:

```
for (i=0; i<n; i++) { ...A[i]... }
```

Если же поведение программы на некотором шаге цикла может зависеть от результатов выполнения тела цикла на предыдущих шагах либо число повторений цикла зависит от результатов выполнения шага, такие циклы и программируемые ими процессы называются **итерационными**. Наиболее широко они применяются в вычислительной математике, когда для получения численного результата используется итерационный цикл **последовательных приближений** к нему.

Итерационный цикл – цикл, в котором число его повторений и поведение программы на каждом шаге цикла зависят от результатов, полученных на предыдущих шагах.

Если изобразить общую схему итерационного цикла, то в нем обязательно будут переменные, сохраняющие результат предыдущего (**x1**) и еще более ранних (**x2,...**) шагов, а также переменная **x** – результат текущего шага:

```
for (x1=...,x2=...; условие(x1,x2); x2=x1,x1=x)
{ ...x = f(x1,x2);...}
```

Если в итерационном цикле гарантируется выполнение одного шага, то может быть использован цикл **do...while**.

```

x=...; x1=...;           // Начальное значение текущего шага
do {
    x2 = x1; x1 = x;      // Следующий шаг
    x = f(x1,x2);         // Результат текущего шага
} while (условие(x2,x1,x)); // Условие завершения

```

Если использовать результат только текущего шага, который зависит от результата предыдущего, то схему цикла можно упростить.

```
for (x=...; условие(x); ) { ...x = f(x);... }
```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Нахождение корня функции методом половинного деления.

Если математическая функция монотонно возрастает или убывает на заданном интервале **a,b**, имея на его концах противоположные знаки, то корень функции **x** можно найти методом половинного деления интервала. Проще говоря, если кривая на интервале (**a,b**) пересекает ось **X**, то к этой точке пересечения можно приблизиться, последовательно уменьшая этот интервал делением его пополам. Сущность алгоритма состоит в проверке значения функции на середине интервала. После проверки из двух половинок выбирается та, на концах которых функция имеет разные знаки. Процесс продолжается до тех пор, пока интервал не сократится до заданных размеров, определяющих точность результата.

Математическая функция, корень которой ищется, задана внешней функцией (уже в терминах Си) вида **double f(double x)**. Для того, чтобы программа могла работать с произвольной внешней функцией, последняя должна быть передана через указатель (см. раздел 3.3).

```

//-----22-01.cpp
//-----Корень функции по методу середины интервала
double find(double a, double b, double (*pf)(double)){
    double c;           // Середина интервала
    if ( (*pf)(a) * (*pf)(b) > 0) return 0.; // Одинаковые знаки
    for (; b-a > 0.001;){
        c = (a+b) / 2; // Середина интервала
        if ( (*pf)(c) * (*pf)(a) > 0)
            a = c;      // Правая половина интервала
        else b = c;     // Левая половина интервала
    }
    return (a+b)/2; } // Возвратить один из концов интервала

```

В данном примере итерационный характер цикла не очень-то и просматривается. Но положение интервала на новом шаге цикла (правая или левая половина) определяется предыдущим шагом, поэтому итерационность все же присутствует.

Нахождение корня функции методом последовательных приближений. Итерационный характер процесса нахождения корня функции явно присутствует в методе последовательных приближений. Чтобы найти корень функции $f(x)=0$, решается эквивалентное уравнение $x = f(x) + x$. Если для него значение x в правой части считать результатом итерационного цикла на предыдущем шаге (x_1), а значение x в левой части – результатом текущего шага цикла, то такой процесс последовательного приближения к результату можно сделать итерационным.

```
x = x0;
do {
    x1 = x;
    x = f(x1) + x1;
} while( условие(x,x1) );
```

Окончательный вид программы включает еще и проверку качественного характера (сходимости) процесса. Дело в том, что данный метод успешно работает не для всех видов функций $f()$ и начальных значений x_0 . В принципе итерационный процесс может приводить, наоборот, к бесконечному удалению от значения корня. Тогда говорят, что процесс расходится. Для проверки сходимости приходится запоминать разность значений x и x_1 предыдущего шага.

```
//-----22-02.cpp
//---- Корень функции по методу последовательных приближений
double find(double x, double eps , double (*pf)(double)){
// Начальное значение, точность и указатель на внешнюю функцию
double x1,dd;
dd = 100.;
do {    x1 = x;
        x = (*pf )(x1) + x1;
        if (fabs(x1-x) > dd )
            return 0.;                // Процесс расходится
        dd = fabs(x1-x);
    }
    while (dd > eps); return x; }      // Выход - значение корня

void main(){ printf("cos(x)=0 x=%f\n",find(0,0.01,cos)); }
```

Вычисление степенного полинома. При вычислении значения степенного полинома необходимы целые степени аргумента x .

$$y = A_n \cdot x^n + A_{n-1} \cdot x^{n-1} + \dots + A_1 \cdot x + A_0$$

Исключить вычисление степеней в явном виде можно, преобразовав полином к виду, соответствующему итерационному циклу с последовательным накоплением результата.

$$y = ((((((A_n \cdot x + A_{n-1}) \cdot x + A_{n-2}) \cdot x + \dots + A_1) \cdot x + A_0$$

шаг 1 шаг 2 шаг 3 ... шаг n

```
//-----22-03.cpp
```

```
//-----Степенной полином
```

```
double poly(double A[], double x, int n){  
    int i; double y;  
    for (y=A[n], i=n-1; i>=0; i--) y = y * x + A[i];  
    return y; }
```

Вычисление суммы степенного ряда. При вычислении сумм рядов, слагаемые которых включают в себя степени аргумента и факториалы, можно также использовать итерационный цикл. В этом цикле значение очередного слагаемого ряда находится умножением предыдущего слагаемого на некоторый коэффициент, что позволяет избавиться от повторного вычисления степеней и факториалов. Сам коэффициент вычисляется делением выражения для n -го и $n-1$ -го членов суммы ряда.

$$Y = S_0(x) + S_1(x) + S_2(x) + \dots + S_{n-1}(x) + S_n(x) + \dots$$
$$k(x,n) = S_n / S_{n-1}$$

Так, для ряда, вычисляющего $\sin(x)$, коэффициент и функция его вычисления имеют вид:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots + (-1)^n x^{2n+1}/(2n+1)!$$
$$S_0 \quad S_1 \quad S_2 \quad S_3 \quad \dots \quad S_n$$
$$k(x,n) = S_n/S_{n-1} = -x^2/(2n(2n+1))$$

```
//--- Вычисление значения функции sin через степенной ряд
```

```
double sum(double x, double eps){  
    double s, sn; // Сумма и текущее слагаемое ряда  
    int n;  
    for (s=0., sn = x, n=1; fabs(sn) > eps; n++) {  
        s += sn;  
        sn = - sn * x * x / (2.*n * (2.*n + 1));  
    } return s;}
```

```
// Вычисление степенного ряда для x
```

```
// в диапазоне от 0.1 до 1 с шагом 0.1
```

```
void main(){  
    double x;  
    for (x=0.1; x <= 1.; x += 0.1)  
        printf("x=%0.1lf\t sum=%0.4lf\t sin=%0.4lf\n", x, sum(x, 0.0001), sin(x));  
}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Для заданного варианта написать функцию вычисления суммы ряда. Для диапазона значений 0.1...0.9 и изменения аргумента с шагом 0.1 вычислить значения суммы ряда и контрольной функции, к которой он сходится, с точностью до четырех знаков после запятой.

Для вариантов 6–8 (см. ниже) использовать значение **sin** и **cos** на предыдущем шаге для вычисления значений на текущем шаге по формулам:

$$\begin{aligned}\sin(nx) &= \sin((n-1)x) \cos(x) + \cos((n-1)x) \sin(x) \\ \cos(nx) &= \cos((n-1)x) \cos(x) - \sin((n-1)x) \sin(x)\end{aligned}$$

В вариантах 3, 9, 11, **Вн** (числа Бернулли) использовать в виде следующего массива значений для $n = 1 \dots 11$.

$$\frac{1}{6}, \frac{1}{30}, \frac{1}{42}, \frac{1}{30}, \frac{5}{66}, \frac{691}{2730}, \frac{7}{6}, \frac{3617}{510}, \frac{43867}{798}, \frac{174611}{330}, \frac{854513}{138}.$$

Вариант	Ряд	Функция
1	$1 - x^2/2! + \dots + (-1)^n x^{2n}/(2n)!$	$\cos(x)$
2	$z = ((x-1)/(x+1)) \cdot (2/1)z + (2/3)z^3 + \dots + (2/2n-1)z^{2n-1}$	$\ln(x)$
3	$x + \frac{1}{3}x^3 + \frac{2}{15}x^5 + \dots + 2^{2n}(2^{2n}-1)B_n x^{2n-1}/(2n)!$ (ряд с $n = 1$)	$\operatorname{tg}(x)$
4	$x + \frac{1}{2 \cdot 3}x^3 + \dots + \frac{1 \cdot 3 \cdot 5 \dots (2n-1)}{2 \cdot 4 \cdot 6 \dots 2n(2n+1)}x^{2n+1}$	$\arcsin(x)$
5	$1 + x \ln 3 + (x \ln 3)^2/2! + \dots + (x \ln 3)^n/n!$	3^x
6	$1 + \frac{1}{2}x + \frac{1 \cdot 1}{2 \cdot 4}x^2 - \dots + (-1)^n \frac{1 \cdot 1 \cdot 3 \dots (2n-3)}{2 \cdot 4 \cdot 6 \dots 2n}x^n$	$(1+x)^{0.5}$
7	$\sin(x) - \sin(2x)/2 + \dots + (-1)^n \sin(nx)/n$	$x/2$
8	$-\cos(x) + \cos(2x)/2^2 + \dots + (-1)^n \cos(nx)/n^2$	$0.25(x^2 - \pi^2/3)$
9	$\frac{1}{x} - \left(\frac{1}{3}x + \frac{1}{45}x^3 + \frac{2}{945}x^5 + \dots + 2^{2n}B_n x^{2n-1}/(2n)! \right)$	$\operatorname{ctg}(x)$
10	$(x-1)/x + (x-1)^2/2x^2 + (x-1)^3/3x^3 + \dots + (x-1)^n/nx^n$	$\ln(x)$
11	$-x^2/2 - x^4/12 - x^6/45 - \dots - 2^{2n-1}(2^{2n}-1)B_n x^{2n}/n(2n)!$	$\ln(\cos(x))$
12	$x - x^3/3! + x^5/5! - \dots + x^{(2n+1)}/(2n+1)!$	$\operatorname{sh}(x)$
13	$1 + x^2/2! + x^4/4 + \dots + x^{2n}/2n!$	$\operatorname{ch}(x)$
14	$x - x^3/3 + x^5/5 + \dots + (-1)^n x^{(2n-1)}/(2n-1)$	$\operatorname{arctg}(x)$
15	$\pi/2 - 1/x + 1/3x^3 - 1/5x^5 + \dots + (-1)^{(n+1)}/(2n+1)x^{(2n+1)}$	$\operatorname{arctg}(x)$
16	$1 - \frac{1}{2}x + \frac{1 \cdot 3}{2 \cdot 4}x^2 - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6}x^3 + \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8}x^4 - \dots$	$1/(1+x)^{0.5}$
17	$1 - 2x + 3x^2 - 4x^3 + 5x^4 + \dots + (-1)^n(n+1)x^n$	$1/(1+x)^2$

18	$\cos(x) + \cos(3x)/3 + \dots + \cos((2n-1)x)/(2n-1)$	$0.51n(\operatorname{ctg}(x/2))$
19	$\cos(x) - \cos(2x)/2 + \dots + (-1)^{n+1} \cos(nx)/n$	$\ln(2\cos(x/2))$
20	$\sin(x) + \sin(3x)/3^3 + \dots + \sin((2n-1)x)/(2n-1)^3$	$(\pi/8)x(\pi-x)$

ТЕСТОВЫЕ ЗАДАНИЯ

Восстановить в общем виде формулу степенного ряда, вычисляемого в данной функции.

Пример выполнения тестового задания

```
for (s=0, sn = 1 n=2; fabs(sn) > eps; n+=2)
{ s += sn;
  sn= sn * x * x / (n*(n+1)); }
```

Для получения аналитической зависимости необходимо восстановить последовательность значений степеней x и произведений целых коэффициентов, составляющих факториалы либо сокращающихся при переходе от шага к следующему. После получения явно наблюдаемой зависимости необходимо перевести ее в аналитический вид для натурального $m = 1, 2, 3...$

n	sn до	Коэффициент	sn после
2	1	$x^2 / (2 \cdot 3)$	$x^2 \cdot (2 \cdot 3)$
4	$x^2 / (2 \cdot 3)$	$x^2 / (4 \cdot 5)$	$x^4 / (2 \cdot 3 \cdot 4 \cdot 5)$
6	$x^4 / (2 \cdot 3 \cdot 4 \cdot 5)$	$x^2 / (6 \cdot 7)$	$x^6 / (2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7)$
m	$x^{2m} / (2 \cdot m + 1)!$		

В данном примере накапливаются четные степени x и нечетные факториалы (обратите внимание, что n в самой программе меняется через 2, а ряд выражен через натуральное m).

```
//-----22-06.cpp
//-----1
double s,sn; int n;
for (s=0, sn = 1, n=1; fabs(sn) > eps; n++)
{ s += sn;
  sn= - sn * x / n; }
//-----2
for (s=0, sn = x, n=1; fabs(sn) > eps; n++)
{ s += sn;
  sn= - sn * x / n; }
//-----3
for (s=0, sn = x; n=1; fabs(sn) > eps; n+=2)
{ s += sn;
  sn= sn * x * x / (n*(n+1)); }
//-----4
for (s=0, sn = x, n=1; fabs(sn) > eps; n+=2)
```



```

{ s += sn;
sn= sn * x / (n *(n+1) ); }
//-----5
for (s=0, sn = x, n=1; fabs(sn) > eps; n++)
{ s += sn;
sn= sn * x * (2*n) / (2*n-1); }
//-----6
for (s=0, sn = x, n=1; fabs(sn) > eps; n+=2)
{ s += sn;
sn= sn * x * x * n / (n+1); }
//-----7
for (s=0, sn = x, n=1; fabs(sn) > eps; n++)
{ s += sn;
sn= sn * x * x * (2*n-1) / (2*n+1); }
//-----8
for (s=0, sn = x, n=2; fabs(sn) > eps; n+=2)
{ s += sn;
sn= sn * x * x * (n -1) / (n+1); }
//-----9
for (s=0, sn = 1, n=1; fabs(sn) > eps; n++)
{ s += sn;
int nn = 2*n-2; if (nn==0) nn=1;
sn= sn * x * x * nn / (2*n); }
//-----10
for (s=0, sn = 1, n=1; fabs(sn) > eps; n+=2)
{ s += sn;
int nn = n-1; if (nn==0) nn=1;
sn= sn * x * x * nn / (n+1);}

```

2.3. СТРУКТУРЫ ДАННЫХ. ПОСЛЕДОВАТЕЛЬНОСТЬ. СТЕК. ОЧЕРЕДЬ

Хороший Сагиб у Сами и умный,
Только больно дерется стеком.

Н.С. Тихонов. Сами

Структура данных – множество взаимосвязанных переменных. Программа включает в себе единство алгоритма (процедур, функций) и обрабатываемых данных. Единицами описания данных и манипулирования ими в любом языке программирования являются переменные. Формы их представления – типы данных, могут быть и заранее определенными (базовые), и сконструированные в программе (производные). Но так или иначе, переменные – это «непосредственно представленные в языке» данные.

Между переменными в программе существуют неявные, непосредственно не наблюдаемые связи. Они могут заключаться в том, что несколько переменных используются алгоритмом для достижения определенной цели, решения частной задачи, причем значения этих переменных будут взаимозависимы (логические связи).

Связи могут устанавливаться и через память – связыванием переменных через указатели либо включением их одна в другую (физические связи) (рис. 2.1).

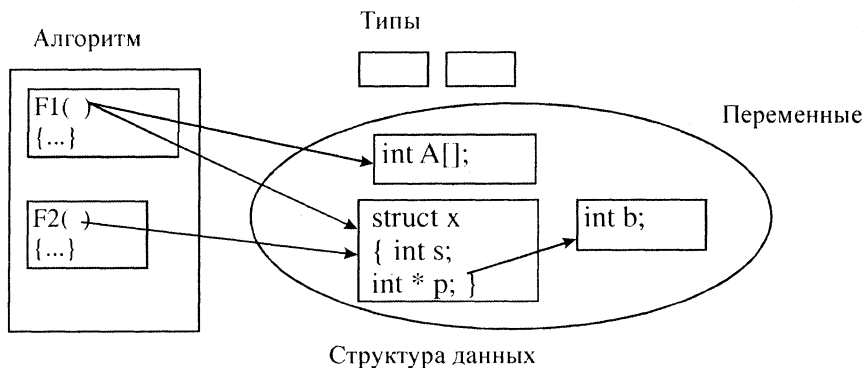


Рис. 2.1

Структура данных – совокупность физически (типы данных) и логически (алгоритм, функции) взаимосвязанных переменных и их значений.

Структура данных – последовательность. Это самая простая иллюстрация различий между переменной и структурой данных. **Последовательностью** называется упорядоченное множество переменных, количество которых может меняться. В идеальном случае последовательность может быть неограниченной, реально же в программе имеются те или иные ограничения на ее длину. Рассмотрим самый простой способ представления последовательности – ее элементы занимают первые n элементов массива (без «дырок»). Чтобы определить текущее количество элементов последовательности, можно поступить двумя способами:

- использовать дополнительную переменную – счетчик числа элементов;
- добавлять каждый раз в качестве обозначения конца последовательности дополнительный элемент с особым значением – признак конца последовательности, например, нулевой **ограничитель последовательности**.

Массив как переменная здесь необходим, но не достаточен для отношения к нему как к структуре данных – последовательности. Для этого нужны еще и правила хранения в нем значений: они мо-

гут определяться и начальным его наполнением, и функциями, которые работают с массивом именно как с последовательностью. У массива, таким образом, возникает дополнительный «смысл», который позволяет по-особому интерпретировать работающие с ним фрагменты.

```
A[0]=0; // Создать пустую последовательность
for(n=0; A[n]!=0; n++); // Найти конец последовательности
for(n=0; A[n]!=0; n++); // Добавить в конец последовательности
A[n]=c; A[n+1]=0;
for (i=0; A[i]!=0; i++); // Включить в последовательность
for (; i>=n; i--) A[i+1]=A[i]; // под заданным номером n
A[n]=c;
for (i=0; A[i]!=0; i++); // Удалить из последовательности
if (n<=i) { // под заданным номером n
for(; A[i]!=0; i++) A[i]=A[i+1]; }
```

Программа, добавляющая элементы в последовательность, должна проверять размерность массива на предмет наличия в нем свободных элементов.

Текстовая строка как последовательность. По определению строка – последовательность символов, ограниченная символом с кодом 0 (конец строки), представляет собой структуру данных, а массив, в котором она находится, является переменной.

Стек. Операции вставки и извлечения элементов из обычной последовательности адресные – они используют номер элемента (индекс). Если ограничить возможности изменения последовательности только ее концами, получим структуры данных, называемые **стеком** и **очередью**.

Стек – последовательность элементов, включение элементов в которую и исключение из которой производится только с одного конца.

Начало последовательности называется дном стека, конец последовательности, в который добавляются элементы и из которых они исключаются, – вершиной стека. Операция добавления нового элемента (запись в стек) имеет общепринятое название **Push** (погрузить), операция исключения – **Pop** (звук выстрела). Операции **Push** и **Pop** безадресные: для их выполнения никакой дополнительной информации о месте размещения элементов не требуется.

Представление стека в массиве. Стек обычно представляется массивом с дополнительной переменной, которая указывает на последний элемент последовательности в вершине стека – указатель стека (рис. 2.2).

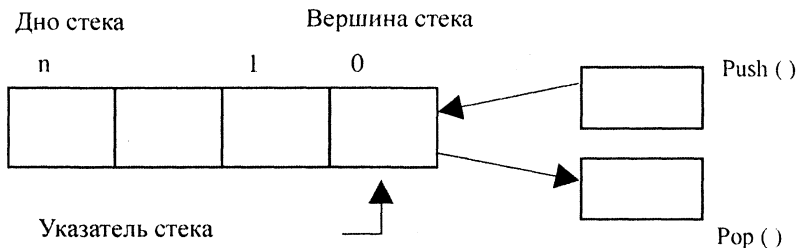


Рис. 2.2

```
//-----23-01.cpp
//-----Основные операции со стеком
#define SIZE 100 // Размерность стека
int Stack[SIZE]; // Массив для размещения стека
int SP; // Указатель стека
void Init(){ SP=-1; } // Стек пуст
void Push(int val) { // Запись в стек
    Stack[ ++SP]=val; } // Запись по указателю стека
int Pop() { // Исключение из стека
    if (SP < 0 ) return(0); // Стек пуст
    return ( Stack[SP--]); // Возвратить элемент по указателю
} // Указатель к предыдущему
```

Использование свойств стека в программировании. Исключительная популярность стека в программировании объясняется тем, что при заданной последовательности записи элементов в стек (например, **А-В-С**) извлечение их происходит в обратном порядке (**С-В-А**). А именно эта последовательность действий соответствует таким понятиям, как вложенность вызовов функций, вложенность определений конструкций языка и т.д. Следовательно, везде, где речь идет о вложенности процессов, структур, определений, механизмом реализации такой вложенности является стек:

- при вызове функции адрес возврата (адрес следующей за вызовом команды) запоминается в стеке, таким образом создается «история» вызовов функций, которую можно восстановить в обратном порядке;

- при синтаксическом анализе вложенных друг в друга конструкций языка трансляторы используют магазинные (стековые) автоматы, стек при этом содержит не до конца проанализированные конструкции языка.

Для способа хранения данных в стеке имеется общепринятый термин – **LIFO** (last in – first out, «последний пришел – первый ушел»).

Другое важное свойство стека – относительная адресация его элементов. На самом деле для элемента, сохраненного в стеке, важно не его абсолютное положение в последовательности, а положение относительно вершины стека или его указателя, которое отражает «историю» его заполнения. Поэтому адресация элементов стека происходит относительно текущего значения указателя стека.

```
//-----23-02.cpp
//-----Работа со стеком по смещению
int Get(int n) { // Получить n-й элемент в стеке
return (Stack[SP-n]); } // относительно указателя стека
```

В архитектуре практически всех компьютеров используется аппаратный стек. Он представляет собой обычную область внутренней (оперативной) памяти компьютера, с которой работает специальный регистр – указатель стека. С его помощью процессор выполняет операции **Push** и **Pop** по сохранению и восстановлению из стека байтов и машинных слов различной размерности. Единственное отличие аппаратного стека от рассмотренной модели – это его расположение буквально «вверх дном», то есть его заполнение от старших адресов к младшим.

Очередь. Объяснять, что такое очередь как способ организации данных, излишне, потому что здесь полностью применим житейский смысл этого понятия.

Очередь – последовательность элементов, включение в которую производится с одного, а исключение из которой – с другого конца.

Для способа хранения данных в очереди есть общепринятый термин – **FIFO** (first in – first out, «первый пришел – первый ушел»).

Простейший способ представления очереди последовательно-стью, размещенной от начала массива, не совсем удобен, поскольку при извлечении из очереди первого элемента все последующие придется постоянно передвигать к началу. Альтернатива: у очереди должно быть два указателя – на ее начало в массиве и на ее конец. По мере постановки элементов в очередь ее конец будет продвигаться к концу массива, то же самое будет происходить с началом при исключении элементов. Выход из создавшегося положения – «зациклить» очередь, то есть считать, что за последним эле-

ментом массива следует опять первый. Подобный способ организации очереди в массиве еще иногда называют циклическим буфером (рис. 2.3).

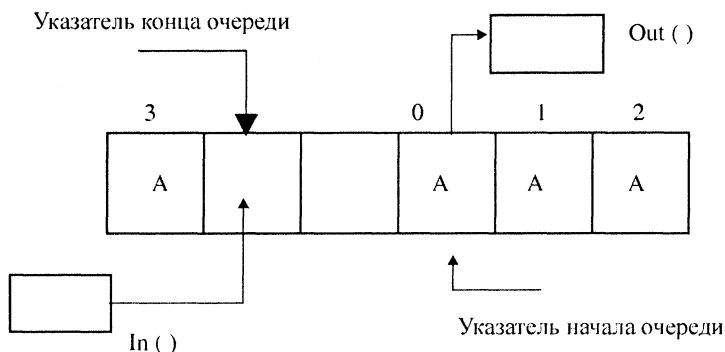


Рис. 2.3

```
//-----23-03.cpp
//-----Основные операции с очередью (циклический буфер )
#define SIZE 100           // Максимальная длина очереди
int QUEUE[SIZE];          // Массив элементов очереди
int fst;                   // Указатели на первый элемент очереди
int lst;                   // Указатель на следующий за последним
void Clear(){ fst = lst = 0; } // Очистить очередь
int In(int val) {          // Поставить в конец очереди
    int next;
    if ((next = (lst+1) % SIZE) == fst)
        return 0;         // Переполнение очереди
    QUEUE[lst] = val;
    lst = next; return 1;}
int Out() {                // Взять из начала очереди
    int val;
    if (fst == lst) return 0; // Очередь пуста
    val = QUEUE[fst++];
    fst %= SIZE;           // По достижении fst==SIZE
    return val; }          // сбрасывается в 0
```

В отличие от стека указатель на конец очереди ссылается не на последний занятый, а на первый свободный элемент массива.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Дайте содержательное определение операциям с последовательностью, стеком и очередью.

```
//-----23-04.cpp
//-----
int sp =-1, LIFO[100];
int lst =0,fst =0,FIFO[100];
```

```

int SEQ[100]={0};
//----- 1
void F1()
{ int c; if (sp < 1) return;
c = LIFO[sp]; LIFO[sp]=LIFO[sp-1]; LIFO[sp-1]=c; }
//----- 2
int F2(int n)
{ int v,i;
if (sp < n) return (0);
v = LIFO[sp-n];
for (i=sp-n; i<sp; i++) LIFO[i]=LIFO[i+1];
sp--; return v;}
//----- 3
void F3(){ LIFO[sp+1]=LIFO[sp]; sp++; }
//----- 4
int F4(int n)
{ int v,i1,i2;
i1 = (fst+n) %100;
v = FIFO[i1];
for (; i1!=lst; i1 = i2){
    i2 = (i1 + 1) % 100;
    FIFO[i1]=FIFO[i2]; }
lst = --lst % 100; return v;}
//----- 5
void F5()
{ int n;
if (fst==lst) return;
n = (lst-1) %100;
FIFO[lst]=FIFO[n];
lst = ++lst % 100;}
//----- 6
void F6(int vv){
for (int i=0; SEQ[i]!=0; i++);
SEQ[i]=vv; SEQ[i+1]=0; }
//----- 7
void F7(int k){
for (int i=0; SEQ[i]!=0; i++);
int c=SEQ[k]; SEQ[k]=SEQ[i-1-k]; SEQ[i-1-k]=c; }

```

2.4. СИМВОЛЫ. СТРОКИ. ТЕКСТ

Особенности обработки текста в Си. Любой язык программирования содержит средства представления и обработки текстовой информации. Другое дело, что обычно программист наряду с символами имеет дело с типом данных (формой представления) – **строкой**, причем особенности ее организации скрыты, а для работы предоставлен стандартный набор функций. В Си, наоборот, форма представления строки открытая, а программист работает с ней «на низком уровне».

Символ текста. Базовый тип данных **char** понимается тройко: как байт – минимальная адресуемая единица представления данных в компьютере; как целое со знаком (в диапазоне $-127...+127$);

как **символ текста**. Этот факт отражает общепринятые стандарты на представление текстовой информации, которые «защиты» и в архитектуре компьютера (клавиатуры, экрана, принтера), и в системных программах (драйверах). Стандартом установлено соответствие между символами и присвоенными им значениями целой переменной (**кодами**). Любое устройство, отображающее символьные данные, при получении кода выводит соответствующий ему символ. Аналогично клавиатура (совместно с драйвером) кодирует нажатие любой клавиши с учетом регистровых и управляющих клавиш в соответствующий ей код.

' '	- 0x20,	'B'	- 0x42,
'*'	- 0x2A,	'Y'	- 0x59,
'0'	- 0x30,	'Z'	- 0x5A,
'1'	- 0x31,	'a'	- 0x61,
'9'	- 0x39,	'b'	- 0x62,
'A'	- 0x41,	'z'	- 0x7A.

Обработка символов. Числовая и символьная интерпретация типа данных **char** позволяет использовать обычные операции для работы с целыми числами для обработки символов текста. Тип данных **char** не имеет никаких ограничений на выполнение операций, допустимых для целых переменных: от операций сравнения и присваивания до арифметических операций и операций с отдельными битами. Но, за исключением редких случаев, знание кодов символов при операциях не требуется. Для представления отдельных символов можно пользоваться символьными (литерными) константами. Транслятор вместо такой константы всегда подставляет код соответствующего символа:

```
char c;
for (c= 'A'; c <= 'Z'; c++) ...
for (c=0x41; c <=0x5A; c++) ...
```

Имеется ряд кодов так называемых неотображаемых символов, которым соответствуют определенные действия при вводе-выводе. Например, символу с кодом **0x0D** («возврат каретки») соответствует перевод курсора в начало строки. Для представления таких символов в программе используются символьные константы, начинающиеся с обратной косой черты.

Константа	Название	Действие
\a	bel	Звуковой сигнал
\b	bs	Курсор на одну позицию назад
\f	ff	Переход к началу (перевод формата)
\n	lf	Переход на одну строку вниз(перевод строки)
\r	cr	Возврат на первую позицию строки

\t	ht	Переход к позиции, кратной 8 (табуляция)
\v	vt	Вертикальная табуляция по строкам
\\'\'\'\'?		Представление символов \, ', ", ?
\nn		Символ с восьмеричным кодом nn
\xnn		Символ с шестнадцатеричным кодом nn
\0		Символ с кодом 0

Некоторые программы и стандартные функции обработки символов и строк (**isdigit**, **isalpha**) используют тот факт, что цифры, прописные и строчные латинские буквы имеют упорядоченные по возрастанию значения кодов:

'0' - '9'	0x30 - 0x39
'A' - 'Z'	0x41 - 0x5A
'a' - 'z'	0x61 - 0x7A

Строка. Строкой называется последовательность символов, ограниченная символом с кодом 0, то есть '\0'. Из ее определения видно, что она является объектом переменной размерности. Местом хранения строки служит массив символов. Суть взаимоотношений строки и массива символов состоит в том, что строка – это структура данных, а массив – это переменная (см. раздел 2.3):

– строка хранится в массиве символов, массив символов может быть инициализирован строкой, а может быть заполнен программно:

```
char A[20] = { 'C','t','p','o','k','a','\0' };
char B[80];
for (int i=0; i<20; i++) B[i] = 'A';
B[20] = '\0';
```

– строка представляет собой последовательность, ограниченную символом '\0', поэтому работать с ней нужно в цикле, ограниченном не размерностью массива, а условием обнаружения символа конца строки:

```
for (i=0; B[i] !='\0'; i++)...
```

– соответствие размерности массива и длины строки транслятором не контролируется, за это несет ответственность программа (программист, ее написавший):

```
char C[20],B[]="Строка слишком длинная для C";
// следить за переполнением массива
// и ограничить строку его размерностью
for (i=0; i<19 && B[i]!='\0'; i++) C[i] = B[i];
C[i]='\0';
```

Строковая константа – последовательность символов, заключенная в двойные кавычки. Допустимо использование неотображаемых символов. Строковая константа автоматически дополняется символом '\0', ею можно инициализировать массив, в том числе такой, размерность которого определяется размерностью строки:

```
char A[80] = "123456\r\n";
char B[] = "aaaaa\033bbbb";
... "Это строка"...
```

Обработка строки. Большинство программ, обрабатывающих строки, используют последовательный просмотр символ за символом – **посимвольный просмотр строки**. Если же в процессе обработки строки предполагается изменение ее содержимого, то проще всего организовать программу в виде посимвольного переписывания входной строки в выходную. Однако этот вариант имеет свои недостатки. При сложной структуре обрабатываемых фрагментов в программе появится много переменных-признаков, фиксирующих те или иные «события» в строке. В этом случае лучше выбрать в качестве шага внешнего цикла обнаружение и обработку основного фрагмента строки, например, слова (**пословный просмотр строки**). Текст такой программы будет в большей степени приближен к формату обрабатываемой строки.

Текстовые файлы. Формат строки, ограниченной символом '\0', используется для представления ее в памяти программы. При чтении строки или последовательности символов из внешнего потока (клавиатура, экран, файл) ограничителем строки является другой символ – '\n' (перевод строки, line feed). Здесь возможны различные «тонкости» при вызове функций, работающих со строками. Например, функция чтения из потока-клавиатуры возвращает строку, ограниченную единственным символом '\0', а функция чтения из потока-файла дополнительно помещает символ '\n', если строка полностью поместилась в отведенный буфер (массив символов).

Функции стандартной библиотеки ввода-вывода обязаны «сглаживать противоречия», связанные с исторически сложившимися формами и анахронизмами в представлении строки в различных устройствах ввода-вывода и операционных системах (текстовый файл, клавиатура, экран) и приводить строки к единому внутреннему формату.

Представление текста. Текст – это упорядоченное множество строк, и наш уровень работы с данными не позволяет предложить для его хранения что-либо иное, кроме двумерного массива символов:

```
char A[20][80];
char B[][40] = { "Строка", "Еще строка", "0000", "abcdef" };
```

Первый индекс двумерного массива соответствует номеру строки, второй – номеру символа в нем:

```
for (int i=0; i<20; i++)
for (int k=0; A[i][k] !='\0'; k++) {...} // Работа с символами i-й строки
```

Внешняя и внутренняя формы представления числа. Текст и числовые данные имеют еще одну точку соприкосновения. Дело в том, что все наблюдаемые нами числовые данные – это совсем не то, с чем имеет дело компьютер. При вводе или выводе целого или вещественного числа мы имеем дело со строкой текста, в которой присутствуют символы, изображающие цифры числа, – внешней формой представления (рис. 2.4).

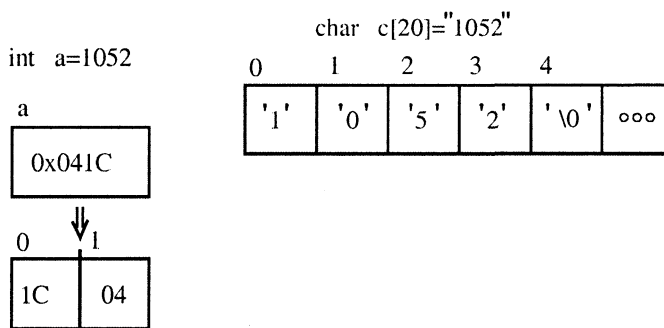


Рис. 2.4

Внутренняя форма представления числа – представление числа в виде целой (int, long) или вещественной переменной.

Внешняя форма представления числа – представление числа в виде строки символов – цифр в заданной системе счисления.

Функции и объекты стандартных потоков ввода-вывода могут, в частности, вводить и выводить целые числа, представленные в десятичной, восьмеричной и шестнадцатеричной системах счисления. При этом происходят преобразования, связанные с переходом от внешней формы представления к внутренней, и наоборот.

Обратите внимание, что о системе счисления имеет смысл говорить только тогда, когда число рассматривается в виде последовательности цифр, то есть во внешней форме представления числа. Внутренняя форма представления числа – двоичная и нас, грубо говоря, не интересует, поскольку компьютер корректно оперирует с ней и без нашего участия.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Обработка символов с учетом особенностей их кодирования. Получить символ десятичной цифры из значения целой переменной, лежащей в диапазоне 0...9:

```
int n; char c; c = n + '0';
```

Получить символ шестнадцатеричной цифры из значения целой переменной, лежащей в диапазоне 0...15:

```
if (n <=9) c = n + '0'; else c = n - 10 + 'A';
```

Получить значение целой переменной из символа десятичной цифры:

```
if (c >='0' && c <='9') n = c - '0';
```

Получить значение целой переменной из шестнадцатеричной цифры:

```
if (c >='0' && c <='9') n = c - '0';  
else  
if (c >='A' && c <='F') c = c - 'A' + 10;
```

Преобразовать строчную латинскую букву в прописную:

```
if (c >='a' && c <='z') c = c - 'a' + 'A';
```

Посимвольная обработка строки. Удаление лишних пробелов. Здесь уместно напомнить одно правило: количество индексов определяет количество независимых перемещений по массивам (степеней свободы). Если для входной строки индекс может изменяться в заголовке цикла посимвольного просмотра (равномерное «движение» по строке), то для выходной строки он меняется только в моменты добавления очередного символа. Кроме того, не нужно забывать «закрывать» выходную строку символом конца строки.

```
//-----24-01.cpp  
//--- Удаление лишних пробелов при посимвольном переписывании  
void nospace(char c1[],char c2[]) {  
    for ( int j=0,i=0;c1[i]!='\0;i++) { // Посимвольный просмотр строки  
        if (c1[i]!=' ') { // Текущий символ не пробел  
            if (i!=0 && c1[i-1]==' ') // Первый в слове -  
                c2[j++]=' '; // добавить пробел  
            c2[j++]=c1[i]; // Перенести символ слова  
        } // в выходную строку  
    }  
    c2[j]='\0'; }
```

Контекст `cl[j++]` имеет вполне определенный смысл: добавить к выходной строке очередной символ и переместиться к следующему.

Поскольку в процессе переписывания размер «уплотненной» части строки всегда меньше исходной, то можно совместить входную и выходную строку в одном массиве (запись нового содержимого будет происходить поверх просмотренного старого).

Посимвольная обработка строки. Поиск слова максимальной длины. Несмотря на ярко выраженный «словный» характер алгоритма, его можно реализовать путем посимвольного просмотра. Достаточно использовать счетчик, который увеличивается на каждый символ слова и сбрасывает его при обнаружении пробела. Дополнительно в момент сбрасывания счетчика фиксируется его максимальное значение, а также индекс начала слова.

```
//-----24-02.cpp
//---- Поиск слова максимальной длины посимвольная обработка
// Функция возвращает индекс начала слова или 1, если нет слов
int find(char s[]) {
    int i,n,lmax,imax;
    for (i=0,n=0,lmax=0,imax=-1; s[i]!='\0'; i++){
        if (s[i]!=' ') n++;           // Символ слова увеличить счетчик
        else {                       // перед сбросом счетчика
            if (n > lmax) { lmax=n; imax=i-n; }
            n=0;                     // Фиксация максимального значения
        }                             // То же самое для последнего слова
    }
    if (n > lmax) { lmax=n; imax=i-n; }
    return imax; }
```

Пословная обработка текста. Поиск слова максимальной длины. В этой версии программы циклов больше, то есть имеются «архитектурные излишества», зато структура программы отражает в полной мере сущность алгоритма.

```
//-----24-03.cpp
//---- Поиск слова максимальной длины пословная обработка
int find(char in[]){
    int i=0, k, m, b;
    b=-1; m=0;
    while (in[i]!='\0') {             // Цикл пословного просмотра строки
        while (in[i]==' ') i++;        // Пропуск пробелов перед словом
        for (k=0; in[i]!=' ' && in[i]!='\0'; i++,k++); // Подсчет длины слова
        if (k>m){                      // Контекст выбора максимума
            m=k; b=i-k; }              // Одновременно запоминается
    }                                  // индекс начала
    return b; }
```

Здесь можно проиллюстрировать еще один принцип разработки программ: после ее написания для произвольной «усредненной»

ситуации необходимо проверить ее «на крайности». В данном случае, при отсутствии в строке слов (строка состоит из пробелов или пуста), установленное начальное значение $b = -1$ будет возвращено в качестве результата (что и задумывалось при установке значения -1 как недопустимого).

Сортировка слов в строке (выбором). При помощи функции поиска можно упорядочить слова по длине. Данный пример – хорошая иллюстрация сущности сортировки выбором, приведенной в разделе 2.5 для обычных массивов: из входного множества объектов (последовательности) выбирается минимальный (максимальный) и переносится в выходное. Наглядность программы состоит в том, что найденное слово удаляется из входной строки «забиванием» его пробелами.

```
//-----24-04.cpp
//---- Сортировка слов в строке в порядке убывания (выбором)
void sort(char in[], char out[])
{ int i=0,k;
  while((k=find(in))!=-1) { // Получить индекс очередного слова
    for (; in[k]!=' ' && in[k]!=0; i++,k++) {
      out[i]=in[k]; in[k]=' '; // Переписать с затиранием
    }
    out[i++]=' '; // После слова добавить пробел
  }
  out[i]=0;}
```

Ввод целого числа. Преобразования при вводе и выводе чисел начинаются с перехода от символа-цифры к значению целой переменной, соответствующему этой цифре, и наоборот:

```
char c; int n;
n = c - '0';
c = n + '0';
```

Ввод целого числа сопровождается его преобразованием из внешней формы – последовательности цифр – к внутренней – целой переменной, которая «интегрирует» цифры в одно значение с учетом их веса (что зависит, кроме всего прочего, и от системы счисления, в которой представлено вводимое число). В преобразовании используется тот факт, что при добавлении к числу очередной цифры справа старое значение увеличивается в 10 раз и к нему – увеличенному – добавляется значение новой цифры, например:

```
Число:      '123'  '1234'
Значение:    123   1234 = 123 * 10 + 4
```

Тогда в основу алгоритма может быть положен цикл просмотра всех цифр числа слева направо, в котором значение числа на теку-

щем шаге цикла получается умножением на 10 результата предыдущего цикла и добавлением значения очередной цифры:

```
n = n * 10 + c[i] - '0';
```

```
//-----24-05.cpp
//---- Ввод десятичного целого числа
int StringToInt(char c[]){
    int n,i;
    for (i=0; !(c[i]>='0' && c[i]<='9'); i++)
        if (c[i]=='\0') return 0; // Поиск первой цифры
    for (n=0; c[i]>='0' && c[i]<='9'; i++) // Накопление целого
        n = n * 10 + c[i] - '0'; // "Цифра за цифрой"
    return n; }
```

Вывод целого числа. В преобразовании используется тот факт, что значение младшей цифры целого числа **n** равно остатку от деления его на 10, вторая цифра – остатку от деления на 10 частного **n/10** и т.д. В основу алгоритма положен цикл, в котором на каждом шаге получается значение очередной цифры справа как остаток от деления числа на 10, а само число уменьшается в 10 раз. Поскольку цифры получаются в обратном порядке (по-арабски), массив символов также необходимо заполнять от конца к началу. Для этого нужно либо вычислить заранее количество цифр, либо заполнить лишние позиции слева нулями или пробелами.

```
//-----24-06.cpp
//---- Вывод целого десятичного числа
void IntToString(char c[], int n)
{ int nn,k;
  for (nn=n, k=0; nn!=0; k++, nn/=10); // Подсчет количества цифр числа
  c[k] = '\0'; // Конец строки
  for (k--; k >=0; k--, n /= 10) // Получение цифр числа
      c[k] = n % 10 + '0'; // в обратном порядке
}
```

Сравнение строк. При работе со строками часто возникает необходимость их сравнения в алфавитном порядке. Простейший способ состоит в сравнении кодов символов, что при наличии последовательного кодирования цифр и латинских букв дает гарантию их алфавитного упорядочения (цифры, прописные латинские, строчные латинские). Так, например, работает стандартная функция **strcmp**.

```
//-----24-07.cpp
//---- Сравнение строк по значениям кодов
int my_strcmp(unsigned char s1[], unsigned char s2[]) {
    for ( int n=0; s1[n]!='\0' && s2[n]!='\0'; n++)
        if (s1[n] != s2[n]) break;
    if (s1[n] == s2[n]) return 0;
    if (s1[n] < s2[n]) return -1;
    return 1; }
```

Обратите внимание на то, что массивы символов указаны как беззнаковые. В противном случае коды с весом более 0x80 (символы кириллицы) будут иметь отрицательные значения и располагаться в алфавите «раньше» латинских, имеющих положительные значения кодов. Чтобы установить свой порядок следования символов в алфавите, символы расставляют в порядке убывания их «весов» и используют порядковый номер символа в последовательности в качестве характеристики его «веса».

```
//-----24-08.cpp
//---- Сравнение строк с заданными "весами" символов
int Carry(char c){
    static char ORD[] = "AaБбВвГгДдЕе1234567890";
    if (c=='\0') return 0;
    for ( int n=0; ORD[n]!='\0'; n++)
        if (ORD[n]==c) return n;
    return n+1; }
int my_strncmp(char s1[],char s2[]){
    int n; char c1,c2;
    for (n=0; (c1=Carry(s1[n]))!='\0' &&(c2=Carry(s2[n]))!='\0'; n++)
        if (c1 != c2) break;
    if (c1 == c2) return 0;
    if (c1 < c2) return -1;
    return 1; }
```

Выделение вложенных фрагментов. Этот пример включает в себя практически все перечисленные выше приемы работы со строкой: поиск символов с запоминанием их позиций, исключение фрагментов, преобразование числа из внутренней формы во внешнюю. Сложность задачи обязывает использовать принцип модульного проектирования. Требуется переписать из входной строки вложенные друг в друга фрагменты последовательно один за другим, оставляя при исключении фрагмента на его месте уникальный номер.

Пример:

a{b{c}b}a{d{e{g}e}d}a => {c}{b1b}{g}{e3e}{d4d}a2a5a

Задачу будем решать по частям. Несомненно, нам потребуется функция, которая ищет открывающуюся скобку для самого внутреннего вложенного фрагмента. Имея ее, можно организовать уже известное нам переписывание и «выкусывание». Основная идея алгоритма поиска состоит в использовании переменной-счетчика, которая увеличивает свое значение на 1 на каждую из открывающихся скобок и уменьшается на 1 на каждую из закрывающихся. При этом фиксируются максимальное значение счетчика и позиция элемента, где это происходит.


```
//-----24-09.cpp
//--- возвращается индекс скобки " {" для пары с максимальной глубиной
int find(char c[]){
    int i;                // Индекс в строке
    int k;                // Счетчик вложенности
    int max;              // Максимум вложенности
    int b;                // Индекс максимальной " {"
    for (i=0, max=0, b=-1; c[i]!=0; i++){
        if (c[i]== '}' ) k--;
        if (c[i]== '{' ) {
            k++;
            if (k>max) { max=k; b=i; }}
    }
    if (k!=0) return 0;    // Защита " от дурака" , нет парных скобок
    return b; }
```

Другой вариант: функция ищет первую внутреннюю пару скобок. Запоминается позиция открывающейся скобки, при обнаружении закрывающейся скобки возвращается индекс последней открывающейся. Заметим, что его также можно использовать, просто последовательность извлечения фрагментов будет другая.

```
//-----24-10.cpp
//--- возвращается индекс скобки " {" для первой самой внутренней пары
int find(char c[]){
    int i;                // Индекс в строке
    int b;                // Индекс максимальной " {"
    for (i=0, b=-1; c[i]!=0; i++){
        if (c[i]== '}' ) return b;
        if (c[i]== '{' ) b=i;
    }
    return b; }
```

Идея основного алгоритма заключается в последовательной нумерации «выкусываемых» из входной строки фрагментов, при этом на место каждого помещается его номер – значение счетчика, которое для этого переводится во внешнюю форму представления.

```
//-----24-11.cpp
//----- Копирование вложенных фрагментов с " выкусыванием"
void copy(char c1[], char c2[]){
    int i=0;              // Индекс в выходной строке
    int k;                // Индекс найденного фрагмента
    int n;                // Запоминание начала фрагмента
    int m;                // Счетчик фрагментов
    for (m=1; (k=find(c1))!=-1; m++){ // Пока есть фрагменты
        for (n=k; c1[k]!='}' ; k++, i++) c2[i]=c1[k]; // Переписать фрагмент
        c2[i++] = c1[k++]; // и его "}"
        if (m/10!=0) c1[n++] = m/10 + '0' ; // На его место две цифры
        c1[n++] = m%10 + '0' ; // номера во внешней форме
        for (;c1[k]!=0; k++, n++) c1[n]=c1[k];
        c1[n]=0; } // Сдвинуть " хвост" к началу
    for (k=0; c1[k]!=0; k++, i++) c2[i]=c1[k]; // Перенести остаток
    c2[i]=0; } // входной строки
```

Практический совет – избегать сложных вычислений над индексами. Лучше всего для каждого фрагмента строки заводить свой индекс и перемещать их независимо друг от друга в нужные моменты. Что, например, сделано выше при «уплотнении» строки – индекс **k** после переписывания найденного фрагмента «останавливается» на начале «хвоста» строки, который переносится под индекс **n** – начало удаляемого фрагмента. Причем записываемые цифры номера смещают это начало на один или два символа. Таким образом фрагмент заменяется во входной строке на его номер (рис. 2.5).

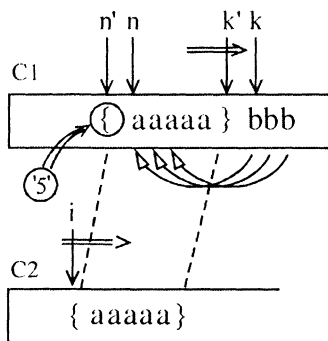


Рис. 2.5

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

1. Выполнить сортировку символов в строке. Порядок возрастания «весов» символов задать таблицей вида **char ORD[] = "AaBbVvГгДдЕе1234567890"**; Символы, не попавшие в таблицу, размещаются в конце отсортированной строки.

2. В строке, содержащей последовательность слов, найти конец предложения, обозначенный символом «точка». В следующем слове первую строчную букву заменить на прописную.

3. В строке найти все числа в десятичной системе счисления, сформировать новую строку, в которой заменить их соответствующим представлением в шестнадцатеричной системе.

4. Заменить в строке принятое в Си обозначение символа с заданным кодом (например, **\101**) на сам символ (в данном случае – **A**).

5. Переписать в выходную строку слова из входной строки в порядке возрастания их длины.

6. Преобразовать строку, содержащую выражение на Си с операциями (**=,==,!=, a+=, a-=**), в строку, содержащую эти же операции с синтаксисом языка Паскаль (**:=,=#, a=a+, a=a-**).

7. Удалить из строки комментарии вида **"/* ... */"**. Игнорировать вложенные комментарии.

8. Заменить в строке символьные константы вида **'A'** на соответствующие шестнадцатеричные (т.е. **'A'** на **0x41**).

9. Заменить в строке последовательность одинаковых символов (не пробелов) на десятичное число, соответствующее их количеству.

ву, и сам символ (т.е. «**abcdaaaaa xyznnnnnnnn**» на «**abcd5a xyz7n**»).

10. Найти в строке два одинаковых фрагмента (не включающих в себя пробелы) длиной более 5 символов и вернуть индекс начала первого из них (т.е. для «**aaaaaabcdefgxxxxxxbcdefgwwwww**» вернуть **n=6** – индекс начала «**bcdefg**»).

11. Оставить в строке фрагменты, симметричные центральному символу, длиной более 5 символов (например, «**dcbabcd**»), остальные символы заменить на пробелы.

12. Найти во входной строке самую внутреннюю пару скобок {...} и переписать в выходную строку содержащиеся между ними символы. Во входной строке фрагмент удаляется.

13. Заменить в строке все целые числа соответствующим повторением следующего за ними символа (например, «**abc5xacb15y**» – «**abcxxxxxacbyyyyyyyyyyyyyyy**»).

14. «Перевернуть» в строке все слова (например, «**Жили были дед и баба**» – «**илиЖ илыб дед и абаб**»).

15. Функция переписывает строку. Если она находит в строке число, то вместо него переписывает в выходную строку соответствующее по счету слово из входной строки (например, «**aaa bb1bb cc2cc**» – «**aaa bbaaabb cccb1bbcc**»).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Содержательно определите действие, производимое над строкой. Напишите вызов функции (входные неизменяемые строки могут быть представлены фактическими параметрами – строковыми константами).

Пример оформления тестового задания

```
//-----24-12.cpp
int F(char c[]){
    for (int i=0,ns=0; c[i]!=0; i++)
        if (c[i]!=' ' && (c[i+1]==' ' || c[i+1]==0)) ns++;
    return ns;}
#include <stdio.h>
void main(){ printf("words=%d\n",F("aaaa bbb ccc dddd"));} // Выведет - 4
```

Функция работает со строкой (поскольку в качестве параметра получает массив символов), которую просматривает до обнаружения символа конца строки. Переменная **ns** является счетчиком. Условие, выполнение которого увеличивает счетчик, – текущий символ не является пробелом, а следующий – пробел либо конец строки. Это условие обнаруживает конец слова. Таким образом, про-

грамма подсчитывает в строке количество слов, разделенных пробелами.

//-----24-13.cpp

```
//----- 1
void F1(char c[])
{ int i,j;
  for (i=0; c[i] !='\0'; i++);
  for (j=0,i--; i>j; i--,j++)
    { char s; s=c[i]; c[i]=c[j]; c[j]=s; }
}
//----- 2
int F2(char s)
{ if (s >='0' && s <='9') return s - '0';
  else return -1; }
//----- 3
void F3(char c[]){
  for ( int i=0; c[i] !='\0'; i++)
    if (c[i] >='a' && c[i] <='z')
      c[i] += 'A' - 'a'; }
//----- 4
int F4(char c[])
{ int i,old,nw;
  for (i=0, old=0, nw=0; c[i] !='\0'; i++) {
    if (c[i]==' ') old = 0;
    else { if (old==0) nw++; old=1; }
    if (c[i]=='\0') break;
  }
  return nw; }
//----- 5
void F5(char c[]){
  for ( int i=0, j=0; c[i] !='\0'; i++)
    if (c[i] !=' ') c[j++] = c[i];
  c[j] = '\0'; }
//----- 6
void F6(char c[], int nn)
{ int k,mm;
  for (mm=nn, k=0; mm !=0; mm /=10 ,k++);
  for (c[k--]='\0'; k>=0; k--)
    { c[k]= nn % 10 + '0'; nn /=10; }
}
//----- 7
int F7(char c[])
{ int i,s;
  for (i=0; c[i] !='\0'; i++)
    if (c[i] >='0' && c[i]<='7') break;
  for (s=0; c[i] >='0' && c[i] <='7'; i++)
    s = s * 8 + c[i] - '0';
  return s; }
//----- 8
int F8(char c[])
{ int n,k,ns;
  for (n=0,ns=0; c[n] !='\0'; n++) {
    for (k=0; n-k >=0 && c[n+k] !='\0'; k++)
      if (c[n-k] != c[n+k]) break;
    if (k >=3) ns++;
  }
  return ns; }
```

```

//----- 9
int F9(char c1[],char c2[])
{ int i,j;
  for (i=0; c1[i] !='\0'; i++) {
    for (j=0; c2[j] !='\0'; j++)
      if (c1[i+j] != c2[j]) break;
    if (c2[j] =='\0') return i;
  }
return -1;}
//----- 10
char F10(char c[])
{ char m,z='?'; int n,s,i;
  for (s=0,m='A'; m <='Z'; m++) {
    for (n=0, i=0; c[i] !='\0'; i++)
      if (c[i]==m) n++;
      if (n > s) { z=m; s=n; }
  }
return z; }
//----- 11
void F11(char c[], double x)
{ int i; x:=(int)x;
  for (c[0]='.', x -= (int)x, i=1; i<6; i++) {
    x *= 10.; c[i]=(int)x + '0'; x -= (int)x; }
c[i]='\0'; }
//----- 12
int F12(char c[]){
  for (int i=0; c[i]!=0; i++){
    if (c[i]==' ') continue;
    for (int j=i+1; c[j]==c[i]; j++);
    for (; c[j]!=0; j++){
      for (int k=0; i+k<j && c[i+k]==c[j+k]; k++);
      if (k>=4) return i;
    } return -1; }
//----- 13
void F13(char c[])
{ int i,j,cm;
  for (i=j=cm=0; c[i] !='\0'; i++) {
    if (c[i]=='*' && c[ i+1]=='/') { cm--, i++; continue; }
    if (c[i]=='/' && c[ i+1]=='*') { cm++, i++; continue; }
    if (cm==0) c[j++ ] = c[i];
  }
c[j]=0; }

```

2.5. СОРТИРОВКА И ПОИСК

Далее он расставил всех присутствующих по этому кругу (строго как попало).

Л. Кэрролл. Алиса в Стране Чудес

Простейшая сортировка. Если попросить не знающих содержание этого раздела написать функцию, выполняющую упорядочение данных в массиве, то 90 процентов напишут примерно так:

//-----25-01.cpp

//----- Дилетантская сортировка

```
void sort(int A[], int n){  
    for (int i=0; i<n; i++)  
        for (int j=i; j<n; j++)  
            if (A[i]>A[j]){  
                int c=A[i]; A[i]=A[j]; A[j]=c;  
            }  
}
```

В основе лежит логика «здравого смысла». Необходимо переставлять элементы массива, если они нарушают порядок, количество таких перестановок должно соответствовать количеству возможных пар элементов, а это дает цикл в цикле. Принцип сравнения «каждый с каждым» приводит к тому, что для каждого i -го элемента необходимо просмотреть все последующие за ним (второй цикл начинается с $j=i$). И наконец, программа отражает справедливую убежденность большинства, что *за один цикл просмотра упорядочить массив нельзя*.

Первый парадокс: несмотря на явное наличие обмена, эта сортировка относится к группе сортировок выбором.

Линейный поиск. Для начала зададимся жизненно важным вопросом: а зачем вообще нужна сортировка? Ответ простой: если данные не упорядочены, то найти что-либо, нас интересующее, можно только последовательным перебором всех элементов. Для обычного массива фрагмент программы, определяющий, имеет ли один из его элементов заданное значение, выглядит так:

```
for (i=0; i<n; i++)  
    if (A[i]==B) break;  
if (i != n) ...найден...
```

То, что мы получаем в данном фрагменте только факт наличия элемента массива с данным значением, не играет никакой роли для понимания сущности поиска данных. В реальных программах «элементами массива» являются, конечно, не простые переменные, а более сложные образования (например, структурированные переменные). Та часть элемента данных, которая идентифицирует его и используется для поиска, называется **ключом**. Остальная часть несет в себе содержательную информацию, которая извлекается и используется из найденного элемента данных.

Ключ – часть элемента данных, которая используется для его идентификации и поиска среди множества других таких элементов.

Приведенный фрагмент программы обеспечивает в неупорядоченном массиве **последовательный**, или **линейный**, поиск, а среднее количество просмотренных элементов для массива размерности N будет равно $N/2$.

Проверка упорядоченности. Функция проверки упорядоченности массива служит живой иллюстрацией теоремы: массив упорядочен, если упорядочена любая пара соседних элементов.

```
//-----25-02.cpp
//---- Проверка упорядоченности массива
int is_sorted(int a[], int n){
for (int i=0; i<n-1; i++)
    if (a[i]>a[i+1]) return 0;
return 1;}
```

Двоичный поиск в упорядоченных данных. Если элементы данных упорядочены, то найти интересующий нас можно значительно быстрее. Алгоритм **двоичного**, или **бинарного**, поиска основан на делении пополам текущего интервала поиска. В основе его лежит тот факт, что при однократном сравнении искомого элемента и некоторого элемента массива мы можем определить, справа или слева от текущего следует искать. Проще всего выбирать элемент на середине интервала, в котором производится поиск. Тогда получим такой алгоритм:

- искомый интервал поиска делится пополам, и по значению элемента массива в точке деления определяется, в какой части следует искать значение на следующем шаге цикла;
- для выбранного интервала поиск повторяется;
- при «сжатии» интервала в 0 поиск прекращается;
- в качестве начального интервала выбирается весь массив.

```
//-----25-03.cpp
//-----Двоичный поиск в упорядоченном массиве
int binary(int c[], int n, int val){// Возвращает индекс найденного
int a,b,m;                        // Левая, правая границы и
for(a=0,b=n-1; a <= b; ) {        // середина
    m = (a + b)/2;                // Середина интервала
    if (c[m] == val)              // Значение найдено
        return m;                // вернуть индекс найденного
    if (c[m] > val)
        b = m-1;                  // Выбрать левую половину
    else
        a = m+1;                  // Выбрать правую половину
return -1; }                      // Значение не найдено
```

Оценим количество сравнений, которые необходимо для поиска требуемого значения. Так как после первого сравнения интервал

уменьшается в 2, после второго – в 4 раза и т.д., то количество сравнений будет не больше соответствующей степени 2, дающей размерность массива n , или $2^s = n$, тогда $s = \log_2(n)$.

Для массива из 1000 элементов их будет 10, из 1 000 000 – 20. Именно ради этого и существуют многочисленные алгоритмы сортировки. С небольшими изменениями данный алгоритм может использоваться для определения **места включения** нового значения в упорядоченный массив. Для этого необходимо ограничить деление интервала до получения единственного элемента ($a=b$), после чего дополнительно проверить, куда следует производить включение.

```
//-----25-04.cpp
//-----Двоичный поиск места включения в упорядоченном массиве
int find(int c[], int n, int val){
    int a,b,m; // Левая, правая границы и
    for(a=0,b=n-1; a < b; ) { // середина
        m = (a + b)/2; // Середина интервала
        if (c[m] == val) // Значение найдено -
            return m; // вернуть индекс
        if (c[m] > val)
            b = m-1; // Выбрать левую половину
        else
            a = m+1; // Выбрать правую половину
    } // Выход по a==b
    if (val > c[a]) return a+1; // Включить на следующую
    return a; } // или на текущую позицию
```

Трудоёмкость алгоритмов. Для сравнения свойств алгоритмов важно не то, сколько конкретно времени они выполняются на данных известного объема, а как они поведут себя при увеличении этого объема в 10, 100, 1000 раз и т.д., то есть тенденция увеличения времени обработки, а оно в свою очередь зависит от количества базовых операций над элементами данных – выборов, сравнений, перестановок. С этой целью введено понятие **трудоёмкости**.

Трудоёмкость – зависимость числа базовых операций алгоритма от размерности входных данных.

Трудоёмкость показывает не абсолютные затраты времени в секундах или минутах, что зависит от конкретных особенностей компьютера, а в какой зависимости растет время выполнения программы при увеличении объемов обрабатываемых данных. Оценим трудоёмкости известных нам алгоритмов (рис. 2.6):

– трудоёмкость линейного поиска – $N/2$ – линейная зависимость;

– трудоемкость двоичного поиска – зависимость логарифмическая $\log_2 N$;

– для сортировки обычно используется цикл в цикле. Отсюда видно, что трудоемкость даже самой плохой сортировки не может быть больше $N \times N$. – зависимость квадратичная. За счет оптимизации она может быть снижена до $N \times \log(N)$;

– алгоритмы рекурсивного поиска, основанные на полном переборе вариантов (см. раздел 3.4), имеют обычно показательную зависимость трудоемкости от размерности входных данных (m^N).

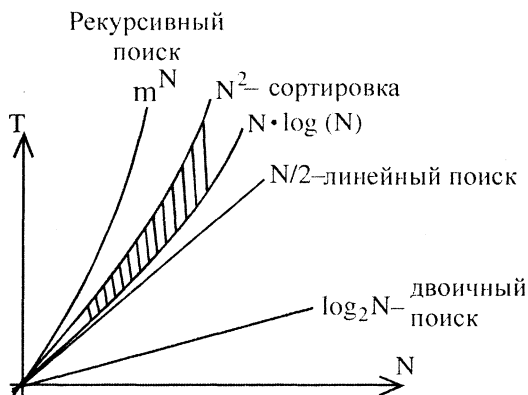


Рис. 2.6

Классификация сортировок. Алгоритмы сортировки можно классифицировать по нескольким признакам.

Вид сортировки по размещению элементов: внутренняя – в памяти, внешняя – в файле данных.

Вид сортировки по виду структуры данных, содержащей сортируемые элементы: сортировка массивов, массивов указателей, списков и других структур данных.

Основная идея алгоритма. В основе многообразия сортировок лежит многообразие идей. Здесь нужно сразу же отделить «зерна от плевел»: идею алгоритма от вариантов ее технической реализации, которых может быть несколько, а также от улучшений основного метода. Кроме того, применительно к разным структурам данных один и тот же алгоритм сортировки будет выглядеть по-разному. Еще более запутывает вопрос использование одной и той же идеи на основной и второстепенной ролях. Например, обмен значений соседей, положенный в основу обменных сортировок,

используется в сортировке вставками, именуемой «сортировка погружением». Попробуем навести здесь порядок.

Прежде всего, выделим сортировки, в которых в процессе работы создается **упорядоченная часть** – размер ее увеличивается на 1 за каждый шаг внешнего цикла. Сюда относятся две группы сортировок:

сортировка вставками: очередной элемент помещается по месту своего расположения в выходную последовательность (массив);

сортировка выбором: выбирается очередной минимальный элемент и помещается в конец последовательности.

Две другие группы используют разделения на части, но по различным принципам и с различной целью:

сортировка разделением: последовательность (массив) разделяется на две частично упорядоченные части по принципу «больше-меньше», которые затем могут быть отсортированы независимо (в том числе тем же самым алгоритмом);

сортировка слиянием: последовательность регулярно распределяется в несколько независимых частей, которые затем объединяются (слияние).

Сортировки этих групп отличаются от «банальных сортировок» тем, что процесс упорядочения в них в явном виде не просматривается (сортировка без сортировки).

Отдельная группа **обменных сортировок** с многочисленными оптимизациями основана на идее регулярного обмена соседних элементов.

Особняком стоит **сортировка подсчетом**. В ней определяется количество элементов, больших или меньших данного, определяется его местоположение в выходном массиве.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Сортировка выбором. На каждом шаге сортировки из последовательности выбирается минимальный элемент и переносится в конец выходной последовательности. Далее вступают в силу детали процесса, но характерным остается наличие двух независимых частей – неупорядоченной (оставшихся элементов) и упорядоченной. При исключении выбранного элемента из массива на его место может быть записано «очень большое число», исключающее его повторный выбор. Выбранный элемент может удаляться путем сдвига оставшейся части, минимальный элемент может меняться местами с «очередным». Трудоемкость алгоритма – $n^2/2$.

Следующий пример – один из многочисленных вариантов «мирного сосуществования» упорядоченной и неупорядоченной частей в одном массиве. Упорядоченная часть находится слева, и ее размерность соответствует числу выполненных шагов внешнего цикла. Неупорядоченная часть расположена справа, поэтому поиск минимума с запоминанием индекса минимального элемента происходит в интервале от *i* до конца массива.

```
//-----25-05.cpp
//---- Сортировка выбором
void sort(int in[], int n){
for ( int i=0; i < n-1; i++){           // Для очередного i
    for ( int j=i+1, k=i; j<n; j++)      // k - индекс минимального
        if (in[j] < in[k]) k=j;         // в диапазоне i..n-1
    int c=in[k]; in[k] = in[i]; in[i] = c; // Три стакана для очередного
}                                         // и минимального
```

В сортировке выбором контекст выбора минимального элемента обычно заметен «невооруженным глазом». Но в следующем варианте он совмещен с процессом обмена и потому не виден: минимальный элемент сразу же перемещается на очередную позицию.

```
//-----25-06.cpp
//---- " Законспирированная" сортировка выбором
void sort(int in[], int n){
for ( int i=0; i < n-1; i++)             // Для очередного i
for ( int j=i+1, k=i; j<n; j++)          // Для всех оставшихся
    if (in[j] < in[i]) {                 // в диапазоне i..n-1
        int c=in[i]; in[i] = in[j]; in[j] = c; // сразу же менять с очередным
    }                                     // Выбор совмещен с обменом
```

Сортировка вставками. Основная идея алгоритма: имеется упорядоченная часть, в которую очередной элемент помещается так, что упорядоченность сохраняется (включение с сохранением порядка). Технические детали: можно проводить линейный поиск от начала упорядоченной части до первого, больше данного, с конца – до первого, меньше данного (трудоемкость алгоритма по операциям сравнения – $nxn/4$), использовать двоичный поиск места в упорядоченной части (трудоемкость алгоритма – $nx\log(n)$). Сама процедура вставки включает в себя перемещение элементов массива (не учтенное в приведенной трудоемкости). В следующем примере последовательность действий по вставке очередного элемента в упорядоченную часть «разложена по полочкам» в виде последовательности четырех действий, связанных переменными.

```
//-----25-07.cpp
//---- Простая вставка
void sort(int in[], int n){
for ( int i=1; i < n; i++) {             // Для очередного i
    int v=in[i];                         // Делай 1 : сохранить очередной
    for (int k=0; k<i; k++)              // Делай 2 : поиск места вставки
        if(in[k]>v) break;               // перед первым, большим v
```

```

for(int j=i-1; j>=k; j--)    // Делай 3: сдвиг на 1 вправо
in[j+1]=in[j];              // от очередного до найденного
in[k]=v;                    // Делай 4 : вставка очередного на место
                             // первого, большего него
}

```

В сортировке выбором нет характерных программных контекстов, «ответственных» за вставку: характер программы определяется циклом поиска места вставки, который корректно работает только на упорядоченных данных. Таким образом, получается замкнутый круг для логического анализа, разрываемый только доказательством методом математической индукции: вставка на i -м шаге выполняется корректно в упорядоченных данных, подготовленных аналогичным $i-1$ -м шагом, и т.д. до 0.

Вставка погружением. Очередной элемент «погружается» путем ряда обменов с предыдущим до требуемой позиции в уже упорядоченную часть массива, пока «не достигнет дна» либо пока не встретит элемент, меньше себя. Наличие контекста «трех стаканов» делает его подозрительно похожим на обменную сортировку, но это не так.

```

//-----25-08.cpp
//----- Вставка погружением, подозрительно похожая на обмен
void sort(int in[],int n) {
for ( int i=1; i<n; i++)    // Пока не достигли " дна" или меньшего себя
    for ( int k=i; k !=0 && in[k] < in[k-1]; k--){
        int c=in[k]; in[k]=in[k-1]; in[k-1]=c;
    }
}

```

Сортировка Шелла. Существенными в сортировках вставками являются затраты на обмены или сдвиги элементов. Для их уменьшения желательно сначала производить погружение с большим шагом, сразу определяя элемент «по месту», а затем делать точную «подгонку». Так поступает **сортировка Шелла**: исходный массив разбивается на m частей, в каждую из которых попадают элементы с шагом m , начиная от 0, 1, ..., $m-1$ соответственно, то есть

```

0 , m , 2m , 3m , ...
1 , m+1, 2m+1, 3m+1, ...
2 , m+2, 2m+2, 3m+2, ...

```

Каждая часть сортируется отдельно с использованием алгоритма вставок или обмена. Затем выбирается меньший шаг, и алгоритм повторяется. Шаг удобно выбрать равным степеням 2, например, 64, 32, 16, 8, 4, 2, 1. Последняя сортировка выполняется с шагом 1. Несмотря на увеличение числа циклов, суммарное число перестановок будет меньшим. Принцип сортировки Шелла можно применить и во всех обменных сортировках.

Замечание. Сортировка Шелла требует четырех вложенных циклов: по шагу сортировки (по уменьшающимся степеням 2 –

$m=64, 32, 16 \dots$), по группам (по индексу первого элемента в диапазоне $k=0\dots m-1$), а затем два цикла обычной сортировки погружением для элементов группы, начинающейся с k с шагом m . Для двух последних циклов нужно взять базовый алгоритм, заменив шаг 1 на m и поменяв границы сортировки.

Обменная сортировка «пузырьком». Обзор вариантов обменной сортировки начнем с горячо любимой автором (с методической точки зрения), но с наименее эффективной простой сортировки обменом, или сортировки **методом «пузырька»**. Суть ее заключается в следующем: производятся попарное сравнение соседних элементов 0–1, 1–2 ... и перестановка, если пара расположена не в порядке возрастания. Просмотр повторяется до тех пор, пока при пробегании массива от начала до конца перестановок больше не будет.

```
//-----25-09.cpp
//-----Сортировка методом "пузырька"
void sort(int A[], int n){
    int i, found;
    do {
        found = 0;
        for (i=0; i<n-1; i++)
            if (A[i] > A[i+1]) {
                int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
                found++;
            }
    } while(found !=0); }
```

// Количество сравнений
// Повторять просмотр...
// Сравнить соседей
// Переставить соседей
//.пока есть перестановки

Оценить трудоемкость алгоритма можно через среднее количество сравнений, которое равно $(n \times n - n)/2$.

Обменные сортировки имеют ряд особенностей. Прежде всего, они чувствительны к степени исходной упорядоченности массива. Полностью упорядоченный массив будет просмотрен ими один раз, в то время как выбор или вставка будут «изображать бурную деятельность». Кроме того, основное свойство, на котором основана их оптимизация, непосредственно не наблюдаемо в тексте программы: ему не соответствует никакой программный контекст, и оно выводится из наблюдения за последовательным выполнением ряда шагов цикла: элемент с большим значением «захватывается» рядом последовательных обменов и «всплывает» к концу массива, пока не встретит элемент, больше себя. С этим последним процесс продолжается.

Шейкер-сортировка учитывает тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные, например:

шаг n	5	7	10	9	8	12	14
	5	7	*****		8	12	14
последняя перестановка	5	7	9	*****		12	14
	5	7	9	8	10	12	14
шаг n+1					-----		
					упорядоченная часть		

Это свойство так же не очевидно, как и предыдущее, то есть не наблюдается непосредственно в программных контекстах. Но исходя из него, просмотр имеет смысл делать не до конца массива, а до последней перестановки, выполненной на предыдущем просмотре. Для этой цели в программе обменной сортировки необходимо запоминать индекс переставляемой пары, который по завершении внутреннего цикла просмотра и будет индексом последней перестановки. Кроме того, необходима переменная – граница упорядоченной части, которая должна при переходе к следующему шагу получать значение пресловутого индекса последней перестановки. Условие окончания – граница сместится к началу массива.

```
//-----25-10.cpp
//----- Однонаправленная Шейкер-сортировка
void sort(int A[], int n){
    int i,b,b1;           // b граница отсортированной части
    for (b=n-1; b!=0; b=b1) { // Пока граница не сместится к правому краю
        b1=0;              // b1 место последней перестановки
        for (i=0; i<b; i++) // Просмотр массива
            if (A[i] > A[i+1]) { // Перестановка с запоминанием места
                int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
                b1=i;
            }
    }
}
```

Если же просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то получим классическую Шейкер-сортировку.

Сортировка подсчетом. Особняком стоящая сортировка, требующая обязательного выходного массива, поскольку элементы в нем размещаются не подряд. Идея алгоритма: число элементов, меньше текущего, определяет его позицию (индекс) в выходном массиве. Наличие переменной-счетчика и использование его в качестве индекса в выходном массиве являются хорошо заметными программными контекстами. Трудоемкость алгоритма – $n \times n/2$.

```
//-----25-11.cpp
//----- Сортировка подсчетом (неполная)
void sort(int in[],int out[],int n)
{ int i,j ,cnt;
  for (i=0; i< n; i++) {
    for ( cnt=0,j=0; j<n; j++)
        if (in[j] < in[i]) cnt++;
    out[cnt]=in[i];
  }
}
```

// Счетчик элементов, больших текущего
// Определяет его место в выходном массиве

Этот фрагмент некорректно работает, если в массиве имеются равные элементы. Объясните поведение программы в такой ситуации и предложите решение проблемы.

Сортировки рекурсивным разделением. Сортировки разделяют массив на две части относительно некоторого значения, называемого **медианой**. Медианой может быть выбрано любое «среднее» значение, например, среднее арифметическое. Сами части не упорядочены, но обладают таким свойством, что элементы в левой части меньше медианы, а в правой – больше. Благодаря такому свойству эти части можно сортировать независимо друг от друга. Для этого нужно вызвать ту же самую функцию сортировки, но уже по отношению не к массиву, а к его частям. Функции, вызывающие сами себя, называются рекурсивными и рассмотрены в разделе 3.4. Рекурсивный вызов продолжается до тех пор, пока очередная часть массива не станет содержать единственный элемент:

```
//---- Схема сортировки рекурсивным разделением
void sort(int in[], int a, int b){
    int i;
    if (a>=b) return;
        // Разделить массив в интервале a..b
        // на две части a..i-1 и i..b
        // относительно значения v по принципу <v, >=v
    sort(in,a,i-1);  sort(in,i,b);}
```

Технический момент: разделение лучше всего производить в отдельном массиве (пример разделения приведен в разделе 1.2), после чего разделенные части перенести обратно. Кроме того, нужно следить, чтобы разделяемые части содержали хотя бы один элемент.

«Быстрая» сортировка умудряется произвести разделение в одном массиве с использованием оригинального алгоритма на основе обмена. Сравнение элементов производится с концов массива (**i=a, j=b**) к середине (**i++** или **j--**), причем «укорочение» происходит только с одной из сторон. После каждой перестановки меняется тот конец, с которого выполняется «укорочение». В результате этого массив разделяется на две части относительно значения первого элемента **in[a]**, который и становится медианой.

```
//-----25-13.cpp
//-----"Быстрая" сортировка
void sort(int in[], int a, int b){
    int i,j,mode;
    if (a>=b) return;                // Размер части =0
    for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++){
        if (in[i] > in[j]){           // Перестановка концевой пары
            int c = in[i]; in[i] = in[j]; in[j]=c;
```

```

        mode = -mode;           // со сменой сокращаемого конца
    }
    sort(in,a,i-1); sort(in,i+1,b);}

```

Очевидно, что медиана делит массив на две неравные части. Алгоритм разделения можно выполнить итерационно, применяя его к той части массива, которая содержит его середину (по аналогии с двоичным поиском). Тогда в каждом шаге итерации медиана будет сдвигаться к середине массива.

Сортировка слиянием. Алгоритм слияния упорядоченных последовательностей рассмотрен в разделе 1.2. На практике слияние эффективно при работе с данными большого объема в последовательных файлах, где принцип слияния последовательно читаемых данных «без заглядывания вперед» выглядит естественно.

Простое однократное слияние базируется на других алгоритмах сортировки. Массив разбивается на n частей, каждая из них сортируется независимо, а затем отсортированные части объединяются слиянием. Реально такое слияние используется, если массив целиком не помещается в памяти. В данной простой модели одномерный массив разделяется на 10 частей – используется двумерный массив из 10 строк по 10 элементов. Затем каждая строка сортируется отдельно. Алгоритм слияния использует стандартные контексты: выбирается строка, в которой первый элемент минимальный (минимальный из очередных), он-то и «сливается» в выходную последовательность. Исключение его производится сдвигом содержимого строки к началу, причем в конец добавляется «очень большое число», играющее роль «затычки» при окончании этой последовательности.

```

//-----25-14.cpp
//----- Простое однократное слияние
void sort(int a[], int n); // Любая сортировка одномерного массива
#define N 4                // Количество массивов
void big_sort(int A[], int n){
    int B[N][10];
    int i,j,m = n/N;        // Размерность массивов
    for (i=0; i<n; i++) B[i/m][i%m]=A[i]; // Распределение
    for (i=0; i<N; i++) sort(B[i],10);    // Сортировка частей
    for (i=0; i<n; i++){           // Слияние
        for ( int k=0, j=0; j<N; j++) // Индекс строки с минимальным
            if (B[j][0] < B[k][0]) k=j; // B[k][0]
        A[i] = B[k][0];             // Слияние элемента
        for (j=1; j<m; j++) B[k][j-1]=B[k][j]; // Сдвиг сливаемой строки
        B[k][m-1]=10000;           // Запись ограничителя
    }
}

```

Циклическое слияние. Оригинальный алгоритм «сортировки без сортировки» базируется на том факте, что при слиянии двух

упорядоченных последовательностей длиной s длина результирующей – в 2 раза больше. Главный цикл включает в себя разделение последовательности на 2 части и их обратное слияние в одну. Первоначально они неупорядочены, тем не менее, можно считать, что в них имеются группы упорядоченных элементов длиной $s=1$. Каждое слияние увеличивает размер группы вдвое, то есть размер группы меняется $s=2, 4, 8, \dots$. Поэтому «собака зарыта» в способе слияния: оно не может выйти за пределы очередной группы, пока обе сливаемые группы не закончились. Это значит, переход к следующей паре осуществляется «скачком» (рис. 2.7).

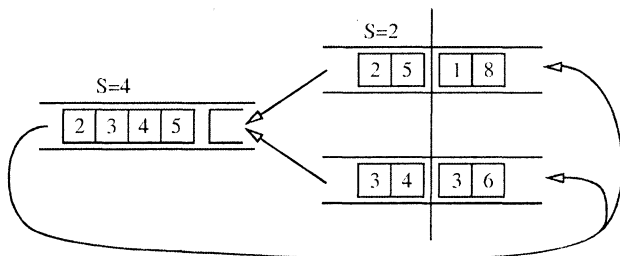


Рис. 2.7

В приведенной программе для простоты размерность массива должна быть равна степени 2, чтобы группы были всегда полными. Внешний цикл организован формально: переменная s принимает значения степени 2. В теле цикла сначала производится разделение массива на две части, а затем – их слияние. Для успешного проектирования слияния важно правильно выбрать индексы с учетом независимости и относительности «движений» по отдельным массивам. Поэтому их здесь целых четыре на три массива. Индекс i в выходном массиве увеличивается в заголовке цикла. Это значит, что за один шаг цикла один элемент из входных последовательностей переносится в выходную. Движение по группам разложено на две составляющие: k – общий индекс начала обеих групп, а $i1, i2$ – относительные индексы внутри групп. Здесь же отрабатывается «скачок» к следующей паре групп: при условии, что обе группы закончились ($i1==s \ \&\& \ i2==s$), обнуляются относительные индексы в группах, а индекс начала увеличивается на длину группы. В процессе слияния отрабатываются четыре возможные ситуации: завершение первой или второй группы и выбор минимального из пары очередных элементов групп – в противном случае.

```
//-----25-15.cpp
//----- Циклическое двухпутевое слияние ( n равно степени 2)
void sort(int A[], int n){
    int B1[100],B2[100];
    int i,i1,i2,s,a1,a2,a,k;
    for (s=1; s!=n; s*=2){ // Размер группы кратен 2
        for (i=0; i<n/2; i++){ // Разделить пополам
            { B1[i]=A[i]; B2[i]=A[i+n/2]; }
        }
        i1=i2=0;
        for (i=0,k=0; i<n; i++){ // Слияние с переходом " скачком "
            if (i1==s && i2==s) // при достижении границ
                k+=s,i1=0,i2=0; // обеих групп
            if (i1==s) A[i]=B2[k+i2++];
            else // 4 условия слияния по окончании
                if (i2==s) A[i]=B1[k+i1++];
            else // групп и по сравнению
                if (B1[k+i1 ] < B2[k+i2 ]) A[i]=B1[k+i1++];
            else A[i]=B2[k+i2++];
        }
    }
}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Алгоритм сортировки реализовать в виде функции, возвращающей в качестве результата характеристику трудоемкости алгоритма (например, количество сравнений). Если имеется базовый алгоритм сортировки (для Шелла – «пузырек», для Шейкер – «пузырек», для вставки с двоичным поиском – погружение), то аналогично оформить базовый алгоритм и сравнить эффективность.

1. Сортировка вставками. Место помещения очередного элемента в отсортированную часть определить с помощью двоичного поиска. Двоичный поиск оформить в виде отдельной функции.

2. Сортировка Шелла. Частичную сортировку с заданным шагом, начиная с заданного элемента, оформить в виде функции. Алгоритм частичной сортировки – вставка погружением.

3. Сортировка разделением. Способ разделения: вычислить среднее арифметическое всех элементов массива и относительно этого значения разбить массив на две части (с использованием вспомогательных массивов).

4. Шейкер-сортировка. Движение в прямом и обратном направлениях реализовать в виде одного цикла, используя параметр – направление движения (+1/–1) и меняя местами нижнюю и верхнюю границы просмотра.

5. «Быстрая» сортировка с итерационным циклом вычисления медианы. Для заданного интервала массива, в котором производится разделение, найти медиану обычным способом. Затем выбрать ту часть интервала между границей и медианой, где находится середина исходного интервала, и процесс повторить.

6. Сортировка циклическим слиянием с использованием одного выходного и двух входных массивов. Для упрощения алгоритма и разграничения сливаемых групп в последовательности в качестве разделителя добавить «очень большое значение» (**MAXINT**).

7. Сортировка разделением. Медиана – среднее между минимальным и максимальным значениями элементов массива. Относительно этого значения разбить массив на две части (с использованием вспомогательных массивов).

8. Простое однократное слияние. Разделить массив на n частей и отсортировать их произвольным методом. Отсортированный массив получить однократным слиянием упорядоченных частей. Для извлечения очередных элементов из упорядоченных массивов использовать массив из n индексов (по одному на каждый массив).

9. Сортировка подсчетом. Выходной массив заполнить значениями «-1». Затем для каждого элемента определить его место в выходном массиве подсчетом количества элементов, строго меньших, чем данный. Естественно, что все одинаковые элементы попадают на одну позицию, за которой следует ряд значений «-1». После этого оставшиеся в выходном массиве позиции со значением «-1» заполнить копией предыдущего значения.

10. Сортировка выбором. Выбрать минимальный элемент в массиве и запомнить его. Затем удалить, а все последующие за ним элементы сдвинуть на один влево. Сам элемент занести на освободившуюся последнюю позицию.

11. Сортировка вставками. Извлечь из массива очередной элемент. Затем от начала массива найти первый элемент, больший, чем данный. Все элементы, от найденного до очередного сдвинуть на один вправо, и на освободившееся место поместить очередной элемент. (Поиск места включения от начала упорядоченной части.)

12. Сортировка выбором. Выбрать минимальный элемент в массиве, перенести в выходной массив на очередную позицию и заменить во входном на «очень большое значение» (**MAXINT**).

13. Сортировка Шелла. Частичную сортировку с заданным шагом, начиная с заданного элемента, оформить в виде функции. Алгоритм частичной сортировки – обменная (методом «пузырька»).

14. Сортировка выбором. Выбрать минимальный элемент в массиве, перенести в выходной массив на очередную позицию. Во входном массиве все элементы от следующего за текущим до конца сдвинуть на один влево.

15. Сортировка «хитрая». Из массива однократным просмотром выбрать последовательность элементов, находящихся в порядке

возрастания, перенести в выходной массив и заменить во входном на «-1». Затем оставшиеся элементы включить в полученную упорядоченную последовательность методом погружения.

16. Оптимизированный двоичный поиск. В процессе деления выбрать не середину интервала, а значение, вычисленное из предположения о линейном возрастании значений элементов массива в текущем интервале поиска. Сравнить эффективности разработанного и базового алгоритмов на массивах с резко неравномерным возрастанием значений (например, 1, 2, 2, 3, 4, 25).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

По тексту программы определите алгоритм сортировки, «смысл» отдельных переменных и назначение циклов.

```
//-----25-16.cpp
//----- 1
void F1(int in[],int n)
{ int i,j,k,c;
  for (i=1; i<n; i++){
    for (k=i; k !=0; k--){
      if (in[k] > in[k-1]) break;
      c=in[k]; in[k]=in[k-1]; in[k-1]=c;
    }
  }
}

//----- 2
void F2(int in[],int out[],int n)
{ int i,j ,cnt;
  for (i=0; i< n; i++) {
    for ( cnt=0,j=0; j<n; j++)
      if (in[j] > in[i]) cnt++;
    else
      if (in[j]==in[i] && j>i) cnt++;
    out[cnt]=in[i];
  } }

//----- 3
void F3(int in[],int n)
{ int a,b,dd,i,lasta,lastb,swap;
  for (a=lasta=0, b=lastb=n, dd=1; a < b; dd = !dd, a=lasta, b=lastb){
    if (dd){
      for (i=a,lastb=a; i<b; i++){
        if (in[i] > in[i+1]){
          lastb = i;
          swap = in[i]; in[i]=in[i+1]; in[i+1]=swap;
        }
      }
    }
    else {
      for (i=b,lasta=b; i>a; i--){
        if (in[i-1] > in[i]){
          lasta = i;
          swap = in[i]; in[i]=in[i-1]; in[i-1]=swap;
        }
      }
    }
  }
}
```

```

//----- 4
int find(int out[],int n, int val);
// Двоичный или линейный поиск расположения значения val
// в массиве out[n]
void F4(int in[], int n){
int i,j,k;
for (i=1; i<n; i++){
    { int c; c = in[i]; k = find(in,i,c);
      for (j=i; j!=k; j--) in[j] = in[j-1];
        in[k] = c; }
}
//----- 5
void F5(int in[], int n){
int i,j,c,k;
for (i=0; i < n-1; i++){
    for (j=i+1,c=in[i],k=i; j<n; j++){
        if (in[j] > c) { c = in[j]; k=j; }
        in[k] = in[i]; in[i] = c;
    }
}
//-----6
void F6(int A[], int n){
int i,found;
do { found =0;
    for (i=0; i<n-1; i++){
        if (A[i] > A[i+1])
            { int cc; cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
              found++;
            }
    } while(found !=0); }
//-----7
void sort(int a[], int n); //Любая сортировка одномерного массива
#define MAXINT 1000
int A[100], B[10][10];
void F7(){
int i,j;
for (i=0; i<100; i++) B[i/10][i%10]=A[i];
for (i=0; i<10; i++) sort(B[i],10);
for (i=0; i<100; i++){
    int k;
    for (k=0, j=0; j<10; j++){
        if (B[j][0] < B[k][0]) k=j;
        A[i] = B[k][0];
        for (j=1; j<10; j++) B[k][j-1]=B[ k ][ j];
        B[k][9]=MAXINT;
    }
}
//-----8
void F8(int in[], int a, int b){
int i,j,mode;
if (a >=b) return;
for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++){
    if (in[i] > in[j]){
        int c = in[i]; in[i] = in[j]; in[j]=c;
        mode = -mode;
    }
}
F8(in,a,i-1); F8(in,i+1,b); }
//-----9
void F9(int A[], int n){

```

```

int i,b,b1;
for (b=n-1; b!=0; b=b1) {
    b1=0;
    for (i=0; i<b; i++)
        if (A[i] > A[i+1]) {
            int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
            b1=i;
        }
    }
}
//-----10
void F10(int A[], int B1[], int B2[], int n){
    int i,i1,i2,s,a1,a2,a,k;
    for (s=1; s!=n; s*=2){
        for (i=0; i<n/2; i++)
            { B1[i]=A[i]; B2[i]=A[i+n/2]; }
        i1=i2=0; a1=a2=MAXINT;
        for (i=0,k=0; i<n; i++)
            {
                if (a1==MAXINT && a2==MAXINT && i1==s && i2==s)
                    k+=s,i1=0,i2=0;
                if (a1==MAXINT && i1!=s) a1=B1[k+i1],i1++;
                if (a2==MAXINT && i2!=s) a2=B2[k+i2],i2++;
                if (a1<a2)a=a1,a1=MAXINT;
                else a=a2,a2=MAXINT;
                A[i]=a;
            }
    }
}

```

2.6. УКАЗАТЕЛИ

Указатель как средство доступа к данным. Передавать данные между программами, данные от одной части программы к другой (например, от вызывающей функции к вызываемой) можно двумя способами:

- создавать в каждой точке программы (например, на входе функции) копию тех данных, которые необходимо обрабатывать;
- передавать информацию о том, где в памяти расположены данные. Такая информация, естественно, более компактна, чем сами данные, и ее условно можно назвать указателем. Получаем «дилетантское» определение указателя:

Указатель – переменная, содержащая информацию о расположении в памяти другой переменной.

Термин «указатель» по сути соответствует более широко трактуемому в информатике термину «ссылка». Ссылка – это данные, обеспечивающие доступ к другим данным (как правило, размещенным в другом месте). Ссылка всегда более компактна, чем адресуемые ею данные, она позволяет обращаться к ним из разных мест, обеспечивает множественный доступ и разделение (рис. 2.8).

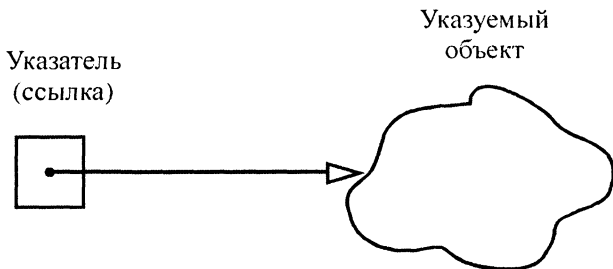


Рис. 2.8

Указатель как элемент архитектуры компьютера. Указатели занимают особое место среди типов данных, потому что они проецируют на язык программирования ряд важных принципов организации обработки данных в компьютере. Понятие указателя связано с такими понятиями компьютерной архитектуры, как адрес, косвенная адресация, организация внутренней (оперативной) памяти. От них мы и будем отталкиваться. **Внутренняя память (оперативная память)** компьютера представляет собой упорядоченную последовательность байтов или машинных слов (ячеек памяти), проще говоря – массив. Номер слова памяти, через который оно доступно как из команд компьютера, так и во всех других случаях, называется **адресом**. Если в команде непосредственно содержится адрес памяти, то такой доступ к этому слову памяти называется **прямой адресацией**.

Возможен также случай, когда машинное слово содержит адрес другого машинного слова. Тогда доступ к данным во втором машинном слове через первое называется **косвенной адресацией**. Команды косвенной адресации имеются в любом компьютере и являются основой любого регулярного процесса обработки данных. То же самое можно сказать о языке программирования. Даже если в нем отсутствуют указатели как таковые, работа с массивами базируется на аналогичных способах адресации данных (рис. 2.9).

В языках программирования имя переменной ассоциируется с адресом области памяти, в которой транслятор размещает ее в процессе трансляции программы. Все операции над обычными переменными преобразуются в команды с прямой адресацией к соответствующим словам памяти.

Указатель – переменная, содержимым которой является адрес другой переменной.

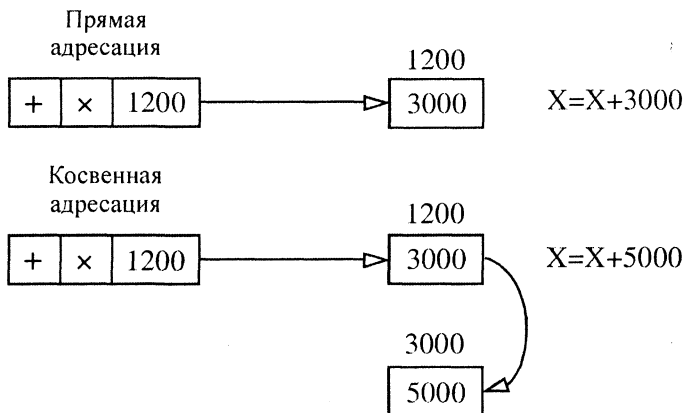


Рис. 2.9

Определение указателя и работа с ним. Соответственно, основная операция для указателя – это косвенное обращение по нему к той переменной, адрес которой он содержит. В Си имеется специальная операция – **"**"**, которую называют **косвенным обращением по указателю**. В более широком смысле ее следует понимать как переход от переменной-указателя к той переменной (объекту), на которую он ссылается. В дальнейшем будем пользоваться такими терминами:

- указатель, который содержит адрес переменной, **ссылается** на эту переменную или **назначен** на нее;
- переменная, адрес которой содержится в указателе, называется **указуемой** переменной.

Последовательность действий при работе с указателем включает три шага.

1. Определение указуемых переменных и переменной-указателя. Для переменной-указателя самым существенным здесь является определение ее типа данных.

```
int a,x; // Обычные целые переменные
int *p;  // Переменная - указатель на другую целую переменную
```

В определении указателя присутствует та же самая операция косвенного обращения по указателю. В соответствии с принципами определения типа переменной (см. раздел 2.8) эту фразу следует понимать так: переменная **p** при косвенном обращении к ней дает переменную типа **int**. То есть свойство ее – быть указателем, определяется в контексте возможного применения к ней операции **"**"**. Обратите внимание, что в определении присутствует **указуе-**

мый тип данных. Это значит, что указатель может ссылаться не на любые переменные, а только на переменные заданного типа, то есть указатель в Си **типизирован**.

2. Связывание указателя с указуемой переменной. Значением указателя является адрес другой переменной. Следующим шагом указатель должен быть настроен, или **назначен**, на переменную, на которую он будет ссылаться (рис. 2.10).

`p = &a;` // Указатель содержит адрес переменной `a`

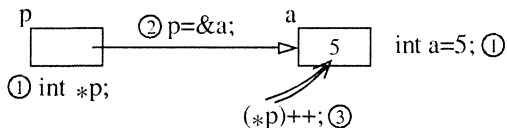


Рис. 2.10

Операция **&** понимается буквально как адрес переменной, стоящей справа. В более широкой интерпретации она «превращает» объект в указатель на него (или производит переход от объекта к указателю на него) и является в этом смысле прямой противоположностью операции *****, которая «превращает» указатель в указуемый объект. То же самое касается типов данных. Если переменная **a** имеет тип **int**, то выражение **&a** имеет тип – указатель на **int** или **int*** (рис. 2.11).

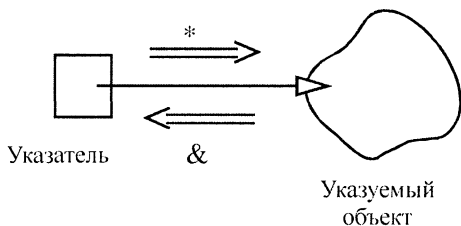


Рис. 2.11

3. И наконец, в любом выражении косвенное обращение по указателю интерпретируется как переход от него к указуемой переменной с выполнением над ней всех далее перечисленных в выражении операций.

<code>*p=100;</code>	// Эквивалентно <code>a=100</code>
<code>x = x + *p;</code>	// Эквивалентно <code>x=x+a</code>
<code>(*p)++;</code>	// Эквивалентно <code>a++</code>

Указатель как «степень свободы» программы. Указатель дает «степень свободы» или универсальности любому алгоритму обработки данных.

Действительно, если некоторый фрагмент программы получает данные непосредственно в переменной, то он может обрабатывать ее и только ее. Если же данные он получает через указатель, то обработка данных (указуемых переменных) может производиться в любой области памяти компьютера (или программы). При этом сам фрагмент может и «не знать», какие данные он обрабатывает, если значение самого указателя передано программе извне (рис. 2.12).

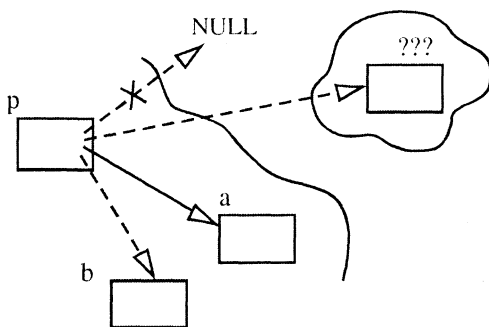


Рис. 2.12

Указатель и память. В Си принята расширенная интерпретация указателя, позволяющая через указатель работать с массивами и с памятью компьютера на низком (архитектурном) уровне без каких-либо ограничений со стороны транслятора. Эта «свобода самовыражения» обеспечивается одной дополнительной операцией **адресной арифметики**.

Любой указатель в Си ссылается на неограниченную в обе стороны область памяти (массив), заполненную переменными указанного типа с индексацией элементов относительно текущего положения указателя.

Адресная арифметика. Операция **указатель+целое**, которая называется операцией **адресной арифметики**, интерпретируется следующим образом (рис. 2.13):

– любой указатель потенциально ссылается на неограниченную в обе стороны область памяти, заполненную переменными указанного типа;

– переменные в области нумеруются от текущей указуемой переменной, которая получает относительный номер 0. Переменные в направлении возрастания адресов памяти нумеруются положительными значениями (1, 2, 3...), в направлении убывания – отрицательными (–1, –2...);

– результатом операции **указатель+i** является адрес i-й переменной (значение указателя на i-ю переменную) в этой области относительно текущего положения указателя.

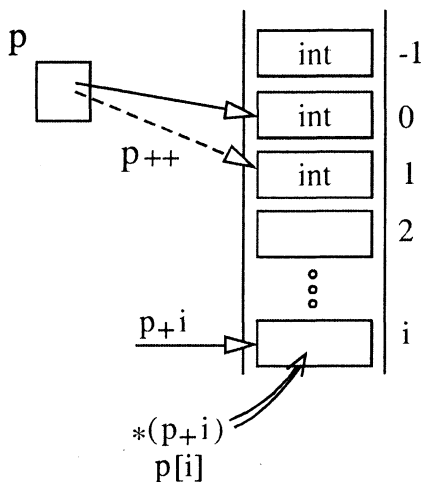


Рис. 2.13

Выражение	Смысл
*p	Значение указуемой переменной
p+i	Указатель на i-ю переменную после указуемой
p-i	Указатель на i-ю переменную перед указуемой
*(p+i)	Значение i-й переменной после указуемой
p[i]	Значение i-й переменной после указуемой
p++	Переместить указатель на следующую переменную
p--	Переместить указатель на предыдущую переменную
p+=i	Переместить указатель на i переменных вперед
p-=i	Переместить указатель на i переменных назад
*p++	Получить значение указуемой переменной и переместить указатель к следующей
*(--p)	Переместить указатель к переменной, предшествующей указуемой, и получить ее значение
p+i	Указатель на свободную память вслед за указуемой переменной

В операциях адресной арифметики транслятором автоматически учитывается размер указываемых переменных, то есть $+i$ понимается не как смещение на i байтов, слов и прочее, а как смещение на i указываемых переменных. Другая важная особенность: при перемещении указателя нумерация переменных в памяти остается относительной и всегда производится от текущей указываемой переменной.

Указатели и массивы. Нетрудно заметить, что указатель в Си имеет много общего с массивом. Наоборот, труднее сформулировать, чем они отличаются друг от друга. Действительно, разница лежит не в принципе работы с указываемыми переменными, а в способе назначения указателя и массива на ту память, с которой они работают. Образно говоря, указателю соответствует массив, «не привязанный» к конкретной памяти, а массиву соответствует указатель, постоянно назначенный на выделенную транслятором область памяти. Это положение вещей поддерживается еще одним правилом: имя массива во всех выражениях воспринимается как указатель на его начало, то есть имя массива A эквивалентно выражению $\&A[0]$ и имеет тип «указатель на тип данных элементов массива». Таким образом, различие между указателем и массивом аналогично различию между переменной и константой. Указатель – это ссылочная переменная, а имя массива – ссылочная константа, привязанная к конкретному адресу памяти.

Массив – память + привязанная к ней адресная константа, **указатель** – «свободно перемещающийся по памяти» массив.

Массив	Указатель	Различия и сходства
<code>int A[20]</code>	<code>int *p</code>	
<code>A</code>	<code>p</code>	Оба интерпретируются как указатели и оба имеют тип <code>int*</code>
<code>---</code>	<code>p=&A[4]</code>	Указатель требует настройки «на память»
<code>A[i]</code> <code>&A[i]</code> <code>A+i</code> <code>*(A+i)</code>	<code>p[i]</code> <code>&p[i]</code> <code>p+i</code> <code>*(p+i)</code>	Работа с областью памяти как с обычным массивом, так и через указатель полностью идентична, вплоть до синтаксиса
<code>----</code>	<code>p++</code> <code>*p++</code> <code>p+=i</code>	Указатель может перемещаться по памяти относительно своего текущего положения

Границы памяти, адресуемой указателем. Если любой указатель ссылается на неограниченную область памяти, то возникают резонные вопросы: где границы этой памяти, кто и как их опреде-

ляет, кто и как контролирует нарушение этих границ указателем. Ответ на него неутешителен для начинающего программиста: транслятор принципиально исключает такой контроль как при трансляции программы, так и при ее выполнении. Он не помещает в генерируемый программный код каких-либо дополнительных команд, которые могли бы это сделать. И дело здесь прежде всего в самой концепции языка Си: не включать в программный код ничего, не предусмотренного самой программой, и не вносить ограничений в возможности работы с данными. Следовательно, ответственность ложится целиком на работающую программу (точнее, на программиста, который ее написал).

На что ссылается указатель? Синтаксис языка в операциях с указателями не позволяет различить в конкретной точке программы, что подразумевается под этим указателем: указатель на отдельную переменную, массив (начало, середину, конец...), какова размерность массива и т.д. Все эти вопросы целиком находятся в ведении работающей программы. Все же даже поверхностный взгляд на программу позволяет сказать, с чем же работает указатель – с отдельной переменной или массивом:

- наличие операции инкремента или индексации говорит о работе указателя с памятью (массивом);

- использование исключительно операции косвенного обращения по указателю свидетельствует о работе с отдельной переменной.

Указатели как формальные параметры. В Си предусмотрен единый способ передачи параметров в функцию – **передача по значению (by value)**. Формальные параметры представляют собой аналог собственных локальных переменных функции, которым в момент вызова присваиваются значения фактических параметров. Формальные параметры, представляя собой копии, могут как угодно изменяться – это не затрагивает соответствующих фактических параметров. Если же требуется обратное, то формальный параметр должен быть определен как указатель, фактический параметр должен быть явно передан в виде указателя на ту переменную, изменения которой производятся в функции.

```
void inc(int *p)
{ (*p)++; }           // Аналог вызова: pi = &a
void main()
{ int a;
  inc(&a); }           // *(pi)++ эквивалентно a++
```

В Си тоже имеется одно такое исключение: формальный параметр – массив – передается в виде неявного указателя на его начало, то есть **по ссылке**.

```

int      sum(int A[],int n)          // Исходная программа
{ int    s,i;
  for (i=s=0; i<n; i++) s+= A[i];
  return s;}
int      sum(int *p, int n)         // Эквивалент с указателем
{ int    s,i;
  for (i=s=0; i<n; i++) s+= p[i];
  return s; }

int      x,B[10]={1,4,3,6,3,7,2,5,23,6};
void main()
{ x = sum(B,10); }                // Аналог вызова: p = B, n = 10

```

В вызове фигурирует идентификатор массива, который интерпретируется как указатель на начало. Поэтому типы формального и фактического параметров совпадают. Совпадают также оба варианта функций вплоть до генерируемого кода.

Указатель – результат функции. Функция в качестве результата может возвращать указатель. Формальная схема функции обязательно включает в себя:

- определение типа результата в заголовке функции как указателя. Это обеспечивается добавлением пресловутой "*" перед именем функции – **int *F(...;**

- оператор **return** возвращает объект (переменную или выражение), являющийся по своей природе (типу данных) указателем. Для этого можно использовать локальную переменную – указатель.

Содержательная сторона проблемы состоит в том, что функция либо *выбирает* один из известных ей объектов (переменных), либо *создает* их в процессе своего выполнения (динамические переменные), возвращая в том и другом случае указатель на него. Для выбора у нее не так уж много возможностей. Это могут быть:

- глобальные переменные программы;
- формальные параметры, если они являются массивами, указателями или ссылками, то есть «за ними стоят» другие переменные.

Функция не может вернуть указатель на локальную переменную или на формальный параметр-значение, поскольку они разрушаются при выходе из функции. Это приводит к не обнаруживаемой ошибке времени выполнения.

Пример: функция возвращает указатель на минимальный элемент массива. Массив передается как формальный параметр.

```

//-----26-01.cpp
//----- Результат функции - указатель на минимальный элемент
int *min(int A[], int n){
  int *pmin, i;          // Рабочий указатель, содержащий результат
  for (i=1, pmin=A; i<n; i++)
    if (A[i] < *pmin) pmin = &A[i];
  return(pmin); }        // В операторе return - значение указателя

```

```
void main() {
    int B[5]={3,6,1,7,2};
    printf("min=%d\n",*min(B,5)); }
```

Прежде всего обратим внимание на синтаксис. Заголовок функции написан таким образом, как будто имя функции является указателем на **int**. Этим способом и обозначается, что ее результат – указатель. Оператор **return** возвращает значение переменной-указателя **pmin**, то есть адрес. Вообще в нем может стоять любое выражение, значение которого является указателем, например:

```
return &A[k];
return pmin+i;
return A+k;
```

Указатель – результат функции – может ссылаться не только на отдельную переменную, но и на массив. В этом смысле он не отличается ничем от других указателей.

Ссылка как неявный указатель. Во многих языках программирования указатель присутствует, но в завуалированном виде в форме ссылки.

Ссылка – неявный указатель, имеющий синтаксис указуемого объекта (синоним).

Под ссылкой понимается объект (переменная), который существует не сам по себе, а как форма отображения на другой объект (переменную). В этом смысле для ссылки больше всего подходит термин **синоним**. В отличие от явного указателя обращение к ссылке имеет тот же самый синтаксис, что и обращение к объекту-прототипу.

```
int a=5;           // Переменная – прототип
int &b=a;          // Переменная b – ссылка на переменную a
b++;              // Операция над b есть операция над прототипом a
```

Наиболее употребительно в Си, а в других языках – единственное возможное использование ссылки как формального параметра функции. Это означает, что при вызове функции формальный параметр создается как **переменная-ссылка**, то есть *отображается* на соответствующий фактический параметр. Различия двух способов передачи:

- при передаче по значению формальный параметр является копией фактического и может быть изменен независимо от значения оригинала – фактического параметра. Это **входной** параметр;

– при передаче по ссылке формальный параметр отображается на фактический, и его изменение сопровождается изменением фактического параметра-прототипа. Такой параметр может быть как **входным**, так и **выходным**.

Формальный параметр-ссылка совпадает с формальным параметром-значением **по форме** (синтаксису использования), а с указателем - **по содержанию** (механизму реализации) (рис. 2.14).

```
//-----
// Формальный параметр - значение
void inc(int vv){ vv++; }
// Передается значение - копия nn
void main(){ int nn=5; inc(nn); } // nn=5
//-----
// Формальный параметр - указатель
void inc(int *pv) { (*pv)++; }
// Передается указатель - адрес nn
void main(){ int nn=5; inc(&nn); } // nn=6
//-----
// Формальный параметр - ссылка
void inc (int &vv) { vv++; }
// Передается указатель - синоним nn
void main(){ int nn=5; inc(nn); } // nn=6
```

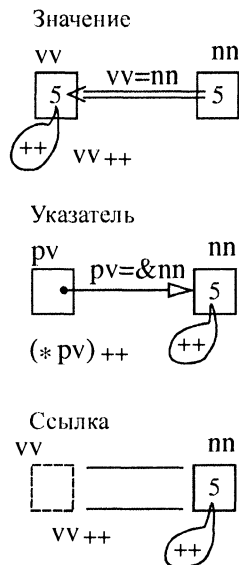


Рис. 2.14

В Си возможна также передача ссылки в качестве результата функции. Ее следует понимать как отображение (синоним) на переменную, которая возвращается оператором **return**. Требования к объекту – источнику ссылки, на который она отображается, еще более строгие – это либо глобальная переменная, либо формальный параметр функции, передаваемый в нее по ссылке или по указателю. При обращении к результату функции – ссылке производится действие с переменной-прототипом. Более подробно все нюансы и примеры будут рассмотрены в разделе 4.2.

```
//-----26-03.cpp
//----- Функция возвращает ссылку на минимальный элемент массива
int &ref_min(int A[], int n){
    for (int i=0,k=0; i<n; i++)
        if (A[i]<A[k]) k=i;
    return A[k];}
void main(){
    int B[5]={4,8,2,6,4};
    ref_min(B,5)++;
    for (int i=0; i<5; i++) printf("%d ",B[i]);
}
```


Здесь «ссылка на ссылке ссылкой погоняет». Формальный параметр **A** – массив, который передается по ссылке и при вызове отображается на **B**. Функция, возвращает ссылку на минимальный элемент **A[k]**, тем самым отображает свой результат на минимальный элемент массива. Кому надоело «играть в прятки» с транслятором, может посмотреть программный эквивалент с использованием обычных указателей (рис. 2.15).

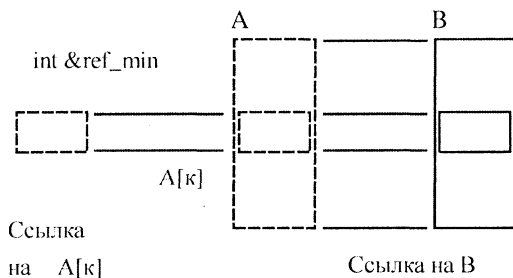


Рис. 2.15

```
//-----26-04.cpp
//---- Функция возвращает указатель на минимальный элемент массива
int *ptr_min(int *p, int n){
    int *pmin;
    for (pmin=p; n>0; p++,n--){
        if (*p < *pmin) pmin=p;
    }
    return pmin;}
void main(){
    int B[5]={4,8,2,6,4};
    (*ptr_min(B,5))++;
    for (int i=0; i<5; i++) printf("%d ",B[i]); }
```

Операции над указателями. В процессе определения указателей мы рассмотрели основные операции над ними:

- операция присваивания указателей одного типа. Назначение указателю адреса переменной **p=&a** есть один из вариантов такой операции;
- операция косвенного обращения по указателю;
- операция адресной арифметики «указатель+целое» и все производные от нее.

Кроме того, имеется еще ряд операций, понимание которых не выходит за рамки «здорового смысла» понятия указателя.

Сравнение указателей на равенство. Равенство указателей однозначно понимается как совпадение адресов, то есть назначение их на одну и ту же область памяти (переменную).

Пустой указатель (NULL-указатель). Среди множества адресов выделяется такой, который не может быть использован для размещения данных в правильно работающей программе. Это значение адреса называется **NULL-указателем**, или «пустым» указателем. Считается, что указатель с таким значением не корректный (указывает «в никуда»). Обычно такое значение определяется в стандартной библиотеке ввода-вывода в виде **#define NULL 0**.

Значение **NULL** может быть присвоено любому указателю. Если указатель по логике работы программы может иметь такое значение, то перед косвенным обращением по нему его нужно проверять на достоверность:

```
int      *p,a;
if (...) p=NULL; else p=&a; ...
if (p !=NULL) *p = 5;    ...
```

Сравнение указателей на «больше-меньше»: при сравнении указателей сравниваются соответствующие адреса как беззнаковые переменные. Если оба указателя ссылаются на элементы одного и того же массива, то соотношение «больше-меньше» следует понимать как «ближе-дальше» к началу массива:

```
//-----26-05.cpp
//--- Симметричная перестановка символов строки
void F(char *p){
  for (char *q=p; *q!=0; q++){
    for (q--; q>p; p++, q--)           // Пока p левее q
      { char c; c=*p; *p=*q; *q=c; } // 3 стакана под указателями
  }
```

Разность значений указателей. В случае, когда указатели ссылаются на один и тот же массив, их разность понимается как «расстояние между ними», выраженное в количестве указуемых переменных.

Преобразование типа указателя. Отдельная операция преобразования типа, связанная с изменением типа указуемых элементов при сохранении значения указателя (адреса), используется при работе с памятью на низком (архитектурном) уровне и рассмотрена подробно в разделе 3.1.

Преобразование целое-указатель: в конечном счете адрес, который представляет собой значение указателя, является обычным машинным словом определенной размерности, чему в Си соответствует целая переменная. Поэтому в Си преобразования типа «указатель-целое» и «целое-указатель» понимаются как получение адреса памяти в виде целого числа и преобразование целого числа в адрес памяти, то есть как работа с реальными адресами памяти

компьютера. Такие операции являются машинно-зависимыми, поскольку требуют знания некоторых особенностей:

- системы преобразования адресов компьютера, размерностей используемых указателей (**int** или **long**);
- распределения памяти транслятором и операционной системой;
- архитектуры компьютера, связанной с размещением в памяти специальных областей (например, видеопамять экрана).

Естественно, что программа, использующая такие знания, не является переносимой (мобильной) и работает только в рамках определенного транслятора, операционной системы или компьютерной архитектуры.

Указатель типа `void*`. Если фрагмент программы «не должен знать» или не имеет достаточной информации о структуре данных в адресуемой области памяти, если указатель во время работы программы ссылается на данные различных типов, то используется указатель на неопределенный (пустой) тип **`void`**. Указатель понимается как адрес памяти, с неопределенной организацией и неизвестной размерностью указуемой переменной. Его можно присваивать, передавать в качестве параметра и результата функции, менять тип указателя, но операции косвенного обращения и адресной арифметики с ним недопустимы.

```
extern void *malloc(int);  
int *p = (int*)malloc(sizeof(int)*20);
```

Функция **`malloc`** возвращает адрес зарезервированной области динамической памяти в виде указателя **`void*`**. Это означает, что функцией выделяется память как таковая, безотносительно к размещаемым в ней переменным. Тип указателя **`void*`** явно преобразуется в требуемый тип **`int*`** для работы с этой областью как с массивом целых переменных.

```
extern int fread(void *, int, int, FILE *);  
int A[20];  
fread((void*)A, sizeof(int), 20, fd);
```

Функция **`fread`** выполняет чтение из двоичного файла **`n`** записей длиной по **`m`** байтов, при этом структура записи для функции неизвестна. Поэтому начальный адрес области памяти передается формальным параметром типа **`void*`**. При подстановке фактического параметра **`A`** типа **`int*`** производится явное преобразование его к типу **`void*`**.

Преобразование типа указателя **`void*`** к любому другому типу указателя соответствует «смене точки зрения» программы на адре-

суемую память от «данные вообще» к «конкретные данные», и наоборот (подробнее о преобразовании типа указателя см. раздел 3.1).

Указатели и многомерные массивы. Двумерный массив реализован как «массив массивов» – одномерный массив с количеством элементов, соответствующих первому индексу, причем каждый элемент представляет собой массив элементов базового типа с количеством, соответствующим второму индексу. Например, `charA[20][80]` определяет массив из 20 массивов по 80 символов в каждом и никак иначе.

Идентификатор массива без скобок интерпретируется как адрес нулевого элемента нулевой строки или указатель на базовый тип данных. В нашем примере идентификатору `A` будет соответствовать выражение `&A[0][0]` с типом `char*`.

Имя двумерного массива с единственным индексом интерпретируется как начальный адрес соответствующего внутреннего одномерного массива; `A[i]` понимается как `&A[i][0]`, то есть начальный адрес *i*-го массива символов.

Указатель на массив. Поскольку любой указатель может ссылаться на массив, термин «указатель на массив» для Си – то же самое, что «масло масляное». Тем не менее, он имеет смысл, если речь идет об указателе на область памяти, содержащей двумерный массив (матрицу), а адресуемой единицей является одномерный массив (строка).

Для работы с многомерными массивами вводятся особые указатели – указатели на массивы. Они представляют собой обычные указатели, адресуемым элементом которых является не базовый тип, а массив элементов этого типа:

```
char    (*p)[80];
```

Круглые скобки имеют здесь принципиальное значение. В контексте определения `p` – это переменная, при косвенном обращении к которой получается массив символов, то есть `p` является указателем на память, заполненную массивами символов по 80 в каждом. При отсутствии скобок имел бы место массив указателей на строки. Следовательно, указатель на массив может быть настроен и может перемещаться по двумерному массиву.

Типичные ошибки при работе с указателями. Основная ошибка, которая периодически возникает даже у опытных программистов, – указатель ассоциируется с адресуемой им памятью. Память – это прежде всего ресурс, а указатель – ссылка на него. Отметим наиболее грубые ошибки:

– неинициализированный указатель. После определения указатель ссылается «в никуда», тем не менее программист работает через него с переменной или массивом, записывая данные по случайным адресам;

– несколько указателей, ссылающихся на общий массив. В этом случае мы имеем дело с одним массивом, а не с несколькими. Если программа работает с несколькими массивами, то они должны либо создаваться динамически, либо браться из двумерного массива;

– выход указателя за границы памяти. Например, конец строки отмечается символом '\0', начало же формально соответствует начальному положению указателя. Если при работе со строкой требуется возвращение на ее начало, то начальный указатель необходимо запоминать либо дополнительно отсчитывать символы.

Строки, массивы символов и указатели `char*`. Среди возможных интерпретаций указателя `char*` – указатель на отдельный символ, на байт, массив байтов, массив целых (размерности 1 байт), можно выделить – указатель на строку: массив, содержащий последовательность символов, ограниченную символом '\0'. Цикл работы со строкой с использованием указателя обычно включает линейное перемещение указателя с проверкой на символ конца строки под указателем.

```
int      strlen(char *p){ // Возвращает длину строки, заданной
int      n;              // указателем на строку char*
for (n=0; *p != '\0'; p++,n++);
return n;}
void      strcat(char *p, char *q){// Объединяет строки,
while (*p !='\0') p++;          // заданные указателями
for (; *q !='\0'; *p++ = *q++);
*p = '\0';}
```

При просмотре массива операции индексирования с линейно изменяющимся индексом (`p[i]` и `i++`) заменены аналогичным линейным перемещением указателя – `*p++`, или `*p, p++`.

Строковая константа в любом контексте программы – это указатель на создаваемый транслятором массив символов, инициализированный этой строкой. Трансляция строковой константы включает в себя:

– создание массива символов с размерностью, достаточной для размещения строки;

– инициализацию (заполнение) массива символами строки, дополненной символом '\0';

– включение в контекст программы, где присутствует строковая константа, указателя на созданный массив символов. В программе ей соответствует тип **char*** – указатель на строку.

```
char *q = "ABCD";           // Программа
char *q;                     // Эквивалент
char A[5] = {'A','B','C','D','\0'};
q = A;
```

Указатель на строку, массив символов, строковая константа.

Имя массива символов, строковая константа и указатель на строку имеют в языке один и тот же тип **char***, поэтому могут использоваться в одном и том же контексте, например, в качестве фактических параметров функций:

```
extern int strcmp(char *, char*);
char *p, A[20];
strcmp(A, "1234");
strcmp(p, A+2);
```

Результат функции – указатель на строку. Функция, возвращающая указатель, может «отметить» им место в строке с интересующими вызывающую программу свойствами. При отсутствии найденного элемента возвращается **NULL**.

Индексация или перемещение указателя. При работе с массивом через указатель всегда существует альтернатива: использовать индексацию при «неподвижном» указателе либо перемещать указатель с помощью операций **p++** или присваивания указателя. Рекомендации – соображения удобства. Единственное исключение: если перемещение по массиву складывается из двух составляющих, то избежать суммирования индексов, а также периодических присваиваний указателей можно сочетанием перемещения указателя и индексации.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Поиск всех вхождений подстроки в строке. Функция получает указатель на начало строки, продвигает его к началу обнаруженного фрагмента и возвращает в качестве результата. Внешний цикл, таким образом, предполагает простое перемещение указателя **p** по строке. В теле цикла для каждого текущего положения указателя **p** проверяется на наличие подстроки. Для этого потребуется индексация относительно текущего положения указателя **p[i]**, тем более что аналогичная индексация используется и во второй строке для попарного сравнения символов.

```
//-----26-06.cpp
//---- Поиск в строке заданного фрагмента
char *find(char *p, char *q){ // Попарное сравнение
for (; *p!='\0'; p++){ // до обнаружения расхождения
    for ( int i=0 ; q[i]!='\0' && q[i]==p[i]; i++);
    if ( q[i] == '\0') return p; // Конец подстроки - успех
    } // иначе продолжить поиск
return NULL;}
```

Для обнаружения всех фрагментов достаточно передавать каждому последующему вызову функции поиска указатель на часть строки, непосредственно следующей за найденным фрагментом.

```
//-----26-07.cpp
//---- Поиск всех вхождений фрагмента в строке
void main()
{ char c[80]="find first abc and next abc and last abc", *q="abc", *s;
for (s=find(c,q); s!=NULL; s=find(s+strlen(q),q)) puts(s);
}
```

В результате получим итерационный цикл, в котором в первый раз функция вызывается с указателем на начало строки, а при повторении цикла – с указателем на первый символ за фрагментом, найденным на текущем шаге – **s+strlen(q)**.

Сортировка слов в строке (выбором). Повторим еще один пример из раздела 2.5, используя технику перемещения указателей по строке.

```
//-----26-08.cpp
//---- Поиск слова максимальной длины - посимвольная обработка
// Функция возвращает указатель начала слова
// или NULL, если нет слов
char *find(char *s) {
int n,lmax; char *pmax;
for (n=0,lmax=0,pmax=NULL; *s!=0;s++){
    if ( *s!=' ') n++; // Символ слова - увеличить счетчик
    else { // Перед сбросом счетчика -
        if (n > lmax) { lmax=n; pmax=s-n; }
        n=0; // фиксация максимального значения
    }
}
if (n > lmax) pmax=s-n; // То же самое для последнего слова
return pmax; }
```

Указатель на начало очередного слова устанавливается перемещением текущего указателя **s**, который в момент запоминания ссылается на первый символ после слова, назад на число символов **n**, равное длине слова.

```
//-----26-09.cpp
//---- Сортировка слов в строке в порядке убывания (выбором)
void sort(char *in, char *out)
{ char * q;
while((q=find(in))!= NULL) { // Получить индекс очередного слова
    for (; *q!=' ' && *q!=0; ) {
```

```

        *out += *q; *q ++=' '; // Переписать с затирианием
    }
    *out++=' '; // После слова добавить пробел
}
*out=0;}

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Вариант задания реализовать в виде функции, использующей для работы со строкой только указатели и операции вида ***p++**, **p++** и т.д. Если функция возвращает строку или ее фрагмент, то это также необходимо сделать через указатель.

1. Функция находит минимальный элемент массива и возвращает указатель на него. С использованием этой функции реализовать сортировку выбором.

2. Шейкер-сортировка использует указатели на правую и левую границы отсортированного массива и сравнивает указатели.

3. Функция находит в строке пары одинаковых фрагментов и возвращает указатель на первый. С помощью функции найти все пары одинаковых фрагментов.

4. Функция находит в строке пары инвертированных фрагментов (например, «**123apг**» и «**гpa321**») и возвращает указатель на первый. С помощью функции найти все пары.

5. Функция производит двоичный поиск места размещения нового элемента в упорядоченном массиве и возвращает указатель на место включения нового элемента. С помощью функции реализовать сортировку вставками.

6. Функция находит в строке десятичные константы и заменяет их на шестнадцатеричные с тем же значением, например, «**aaaaa258xxx**» на «**aaaaa0x102xxx**».

7. Функция находит в строке символьные константы и заменяет их на десятичные коды, например, «**aaa'б'xxx**» на «**aaa54xxx**».

8. Функция находит в строке самое длинное слово и возвращает указатель на него. С ее помощью реализовать размещение слов в выходной строке в порядке убывания их длины.

9. Функция находит в строке самое первое (по алфавиту) слово. С ее помощью реализовать размещение слов в выходной строке в алфавитном порядке.

10. Функция находит в строке симметричный фрагмент вида «**abcdcba**» длиной 7 и более символов (не содержащий пробелов) и возвращает указатель на его начало и длину. С использованием функции «вычеркнуть» все симметричные фрагменты из строки.

11. «Быстрая» сортировка (разделением) с использованием указателей на правую и левую границы массива, текущих указателей на правый и левый элемент и операции сравнения указателей.

12. Сортировка выбором символов в строке. Использовать указатели на текущий и минимальный символы.

13. Найти в строке последовательности, состоящие из одного повторяющегося символа, и заменить его на число символов и один символ (например, «аааааа» – «5а»).

14. Функция создает копию строки и «переворачивает» в строке все слова (например: «Жили были дед и баба» – «илиЖ илиб дед и абаб»).

Примечание: функция, производящая поиск некоторого фрагмента переменной размерности, может вернуть эту размерность по ссылке.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите значения переменных после вызова функции.

```
//-----26-10.cpp
//----- 1
int inc1( int vv) { vv++; return vv; }
void main1(){ int a,b=5; a=inc1(b); }
//----- 2
int inc2( int &vv) { vv++; return vv; }
void main2(){ int a,b=5; a=inc2(b); }
//----- 3
int inc3( int &vv) { vv++; return vv; }
void main3(){ int a,b=5; a=inc3(++b); }
//----- 4
int &inc4( int &vv) { vv++; return vv; }
void main4(){ int a,b=5; a=inc4(b); }
//----- 5
int inc5(int &x) { x++; return x+1; }
void main5 ()
{ int x,y,z,t; x = 5; y = inc5(x); z = inc5(t=inc5(x)); }
//----- 6
int &inc6(int &x){ x++; return x; }
void main6 ()
{ int x,y,z; x = 5; y = inc6(x); z = inc6(inc6(x)); }
//----- 7
int *inc7(int &x) { x++; return &x; }
void main7 ()
{ int x,y,z; x = 5; y = * inc7(x); z = * inc7(*inc7(x)); }
//----- 8
int inc8(int x) { x++; return x+1; }
void main8 () { int x,y,z; x = 5; y=inc8(x);
z = inc8(inc8(x)); }
```

ПРОГРАММНЫЕ ЗАГОТОВКИ И ТЕСТОВЫЕ ЗАДАНИЯ

Определите, используется ли указатель для доступа к отдельной переменной или к массиву. Напишите вызов функции с соответствующими фактическими параметрами – адресами переменных или именами массивов.

Пример оформления тестового задания

```
//-----26-11.cpp
//-----
void F(int *p, int *q, int n){
    for (*q = 0; n > 0; n--){
        *q = *q + *p++; }
    void main(){
        int x,A[5]={1,3,7,1,2};
        F(A,&x,5); printf("x=%d\n",x); } // Выведет 14
```

Формальный параметр **p** используется в контексте ***p++**, что означает работу с последовательностью переменных, то есть с массивом. Число повторений цикла определяется параметром **n**, соответствующим размерности массива. Указатель **q** используется для косвенного обращения через него к отдельной переменной. Поэтому при вызове функции фактическими параметрами являются: имя массива – указатель на начало; адрес переменной – указатель на нее; константа – размерность массива, передаваемая по значению.

```
//-----26-12.cpp
//----- 1
void F1(int *p1, int *p2)
{ int c; c = *p1; *p1 = *p2; *p2 = c; }
//----- 2
void F2(int *p, int *q, int n){
    for (*q = *p; n > 0; n--, p++)
        if (*p > *q) *q = *p; }
//----- 3
int *F3(int *p, int n)
{ int *q;
    for (q = p; n > 0; n--, p++)
        if (*p > *q) q = p;
    return q; }
//----- 4
void F4(char *p)
{ char *q;
    for (q=p; *q !='\0'; q++);
    for (q--; p < q; p++, q--)
        { char c; c = *p; *p = *q; *q = c; }}
//----- 5
int F5(char *p)
{ int n;
    for (n=0; *p !='\0'; p++, n++);
    return n; }
```

```

//----- 6
char *F6(char *p, char *q){
for (; *p != '\0'; p++){
    for ( int j=0; q[j] != '\0' && p[j] == q[j]; j++);
    if (q[j] == '\0') return p;
}
return NULL;}
//----- 7
void F7(char *p, char *q){
for (; *p != '\0'; p++);
for (; *q != '\0'; *p++ = *q++);
*p = '\0'; }
//----- 8
int F8(char *p)
{ int n;
if (*p == '\0') return (0);
if (*p != ' ') n=1; else n=0;
for (p++; *p != '\0'; p++)
    if (p[0] != ' ' && p[-1] == ' ') n++;
return n; }
//----- 9
void F9(char *p)
{ char *q; int n;
for (n=0, q=p; *p != '\0'; p++){
    if (*p != ' ')
        { n=0; *q++ = *p; }
    else { n++; if (n==1) *q++ = *p; }
} *q=0; }
//----- 10
void F10(char *p)
{ char *q; int cm;
for (q=p, cm=0; *p != '\0'; p++) {
    if (p[0] == '*' && p[1] == '/') { cm--, p++; continue; }
    if (p[0] == '/' && p[1] == '*') { cm++, p++; continue; }
    if (cm==0) *q++ = *p;
}
*q=0; }

```

ГОЛОВОЛОМКИ, ЗАГАДКИ

Определите значения указанных ниже переменных.

```

char c1 = "ABCD"[3];
char c2 = ("12345" + 2)[1];
for (char *q = "12345"; *q != '\0'; q++);
char c3 = * (--q);

```

Объясните машинно-зависимый (архитектурный) смысл выражения.

```
*(int*)0x1000=5;
```

Найдите ошибки в функциях.

```

char *F1(){ char cc='Á'; return &cc; }
int *F2(int a){ a++; return &a; }

```

2.7. СТРУКТУРИРОВАННЫЕ ТИПЫ

Структурированный тип. Структурированная переменная (или просто структура) в некотором смысле является прямой противоположностью массиву. Так, если массив представляет собой упорядоченное множество переменных одного типа, последовательно размещенных в памяти, то структура – аналогичное множество, состоящее из переменных разных типов.

```
struct man {           // Имя структуры
    char   name[10];   // Элементы структуры
    int    dd,mm,yy;
    char   *address;
}             // Определение структурированных переменных
A, B, X[10];
```

Составляющие структуру переменные имеют имена, по которым они идентифицируются в ней. Их называют **элементами структуры**, или **полями**, и они имеют синтаксис определения обычных переменных. Использоваться где-либо еще, кроме как в составе структурированной переменной, они не могут. В данном примере структура состоит из массива 10 символов **name**, целых переменных **dd**, **mm** и **yy** и указателя на строку **address**. После определения элементов структуры следует список структурированных переменных. Каждая из них имеет внутреннюю организацию описанной структуры, то есть полный набор перечисленных элементов. Имя структурированной переменной идентифицирует всю структуру в целом, имена элементов – составные ее части. В данном случае мы имеем переменные **A**, **B** и массив **X** из 10 структурированных переменных (рис. 2.16).

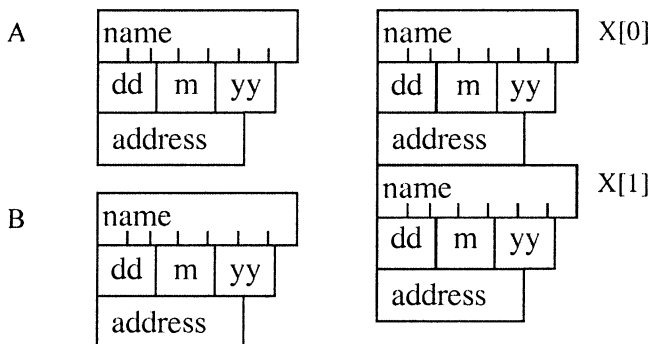


Рис. 2.16

Другое важное свойство структуры – это наличие у нее имени. Имя характеризует структуру как тип данных (форму представления данных) и может использоваться в программе аналогично именам базовых типов для определения переменных, массивов, указателей, спецификации формальных параметров и результата функции, порождения новых типов данных.

```
man C,D[20],*p;
man *create()    { ... }
void f(man *q)   { ... }
```

Структурированный тип определяется сам по себе, то есть без конкретных структурированных переменных.

```
struct man {
    char   name[10];
    int    dd,mm,yy;
    char   *address;
};
```

При определении глобальной (внешней) структурированной переменной или массива таких переменных они могут быть инициализированы списками значений элементов, заключенных в фигурные скобки и перечисленных через запятую.

```
man A = { "Петров",1,10,1969,"Морская-12" };
man X[10] =
    { { "Смирнов",12,12,1977,"Дачная-13" },
      { "Иванов",21,03,1945,"Северная-21" },
      { ..... }
    };
```

Способ работы со структурированной переменной вытекает из ее аналогии с массивом. Точно так же, как нельзя выполнить операцию над всем массивом, но можно – над отдельным его элементом, структуру можно обрабатывать, выделяя отдельные ее элементы. Для этой цели существует операция «.» (точка), аналогичная операции [] в массиве. В структурированной переменной она выделяет элемент с заданным именем.

```
A.name      // Элемент name структурированной переменной A
B.dd        // Элемент dd структурированной переменной B
```

Если элемент структуры – не простая переменная, а массив или указатель, то для него применимы соответствующие ему операции ([],* и адресная арифметика):

```
A.name[i]   // i-й элемент массива name, который является
             // элементом структурированной переменной A
*B.address   // Косвенное обращение по указателю address,
             // который является элементом структурированной
             // переменной B
```

```
B.address[j]    // Индексация по указателю address,
                // который является элементом структурированной
                // переменной B
```

Единственным технологическим отличием от массива является то, что структурированные переменные можно присваивать друг другу, передавать в качестве формальных параметров и возвращать как результат функции по значению, а не только через указатель. При этом происходит копирование всей структурированной переменной «байт в байт».

```
void FF(man X){ ...}
void main(){
man A,B[10],*p;
A=B[4]; // Прямое присваивание структур
p=&A;   // Присваивание через косвенное обращение по указателю
B[0]=*p; // B[0]=A
FF(A);  // Присваивание при передаче по значению X=A
```

Указатель на структуру. Операция «->». То, что указатели на структурированные переменные имеют широкое распространение, подтверждается наличием в Си специальной операции «->» (стрелка, минус-больше), которая понимается как выделение элемента в структурированной переменной, адресуемой указателем (рис. 2.17). То есть операндами здесь являются указатель на структуру и элемент структуры. Операция имеет полный аналог в виде сочетания операций «*» и «.»:

```
man *p,A;      p = &A;
p->mm          // эквивалентно (*p).mm
```

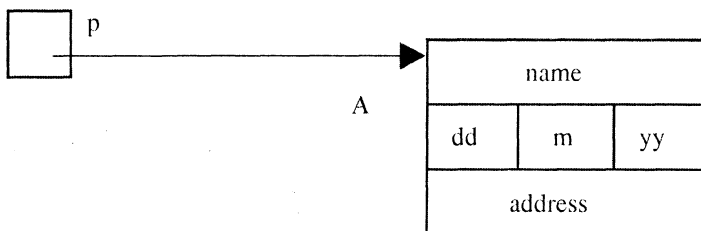


Рис. 2.17

Структура – формальный параметр и результат функции. В отличие от массива, передаваемого только по ссылке (либо доступного функции через указатель), структура может быть передана и возвращена функции всеми возможными способами: по значению, по ссылке и через указатель. Поскольку Си – это язык, приближенный к архитектуре компьютера, программисту известны

механизмы передачи параметров, и он может сравнить затраты времени и памяти в различных вариантах, особенно если размер структурированной переменной достаточно велик:

– при передаче указателя или ссылки на структуру и возвращении их в качестве результата в стек помещается адрес структуры (с размерностью целой переменной). Сама структурированная переменная доступна через указатель (ссылку) «по записи»:

```
struct man{ ...int dd,mm,yy,...};
void proc(man *p){
    p->dd++;           // Для доступа к структуре через указатель
}                     // используется операция ->
void proc1(man &B){    // Структура-прототип через ссылку
    B.dd++;           // доступна «по записи»
}
void main(){
    man A={...,12,5,2001,...};
    proc(&A); proc1(A); }
```

– при передаче формального параметра – структуры по значению в стек помещается копия структуры – фактического параметра, которая может занимать в нем «довольно много места», а копирование – «довольно много времени»:

```
struct man{ ...int dd,mm,yy,...};
void proc(man B){      // Копия структуры – фактического параметра
    cout << B.dd;      // читается, а при изменении не влияет
    B.dd++;            // на оригинал (A.dd не меняется)
}
void main(){
    man A={...,12,5,2001,...};
    proc(A);           // Эквивалент B=A }
```

– при возвращении в качестве результата указателя или ссылки передается адрес структуры (с размерностью целой переменной), для которого не требуется «много места» (передается обычно в регистрах процессора). О характере указуемой переменной уже упоминалось (см. раздел 2.6). Она не может быть локальной переменной или формальным параметром-значением. Она может быть динамической переменной, создаваемой функцией (см. раздел 3.2). Это может быть указатель на глобальные переменные либо на переменные, переданные на вход функции через указатель (ссылку);

– при возвращении структуры по значению в вызывающей функции транслятор создает временную структурированную переменную, а вызываемая функция получает указатель на эту переменную. При выполнении операции **return** происходит копирование возвращаемой переменной во временную переменную через указатель, что может занимать «довольно много времени», а временная переменная – «довольно много места»:

```

struct man{ ...int dd,mm,yy;...}; // Эквивалент программы
man proc( man X){               // man proc(man *out, man X){
X.dd++;                          // X.dd++;
return X;                        // *out = X;
}
void main(){                     // man tmp;
man A={...,12,5,2001,...};       // X=A; out = &tmp;
printf("%d\n",proc(A).dd); }     // Выполнить тело proc
}                                 // Вывод tmp.dd

```

Иерархия типов данных и функций. Иерархия типов данных – определение одного типа данных через другой (в частном случае – вложенность одного в другой) – задает естественный вид связей функций в программе. Последовательность их вызовов будет соответствовать переходу от переменной внешнего типа данных к составляющему (вложенному в нее) типу. Формальные параметры этих функций (точнее, их типы) отражают цепочку вложенных определений типов (в примере: символ – строка – структура – массив структур), при вызове очередной функции в фактическом параметре производится извлечение **составляющего типа данных** (см. раздел 2.8):

```

// Иерархия типов данных и функций
struct man{ ...char name[30];...};
void proc_str(char c[]){         // Уровень 1 – обработка строки
... c[i] ...                     // Уровень 0 – базовый тип данных
}
void proc_man(man *p){           // Уровень 2 – обработка структуры
... proc_str(p->name) ...
}                                 // Уровень 3 – обработка массива структур
void proc_people(man A[], int n){
for (int i=0; i<n; i++)
... proc_man(&A[i]) ...
}
man B[10];                       // Уровень 4 – main
void main(){ ... proc_people(B,10) ... }

```

Объединения. Объединение представляет собой структурированную переменную с несколько иным способом размещения элементов в памяти. Если в структуре (как и в массиве) элементы расположены последовательно друг за другом, то в объединении – «параллельно». То есть для их размещения выделяется одна общая память, в которой они перекрывают друг друга и имеют общий адрес. Размерность ее определяется максимальной размерностью элемента объединения. Синтаксис объединения полностью совпадает с синтаксисом структуры, только ключевое слово **struct** заменяется на **union**.

Назначение объединения заключается не в экономии памяти, как может показаться на первый взгляд. На самом деле оно являет-

ся одним из инструментов управления памятью на принципах, принятых в Си. В разделе 3.1 мы увидим, как использование указателей различных типов позволяет реализовать эти принципы. Здесь же, не вдаваясь в подробности, отметим одно важное свойство: если записать в один элемент объединения некоторое значение, то через другой элемент это же содержимое памяти можно прочесть уже в иной форме представления (как переменную другого типа). Естественно, что при таком манипулировании внутренним представлением данных необходимо знать их форматы и размерность (см. раздел 3.9).

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Представление таблицы в виде массива структур. Первый в нашей практике пример достаточно большой программы (по крайней мере не вмещающейся в рамки этой книги, чтобы приводить ее полностью) демонстрирует торжество принципа модульности в иерархических данных: множество мелких функций, каждая из которых делает на своем уровне ограниченное, но законченное действие, вызывая функции для работы с данными других уровней. Иначе, все «поплывет».

Первым шагом определяются типы данных и переменные: речь идет о таблице, представленной в массиве структур. Сразу же надо принять решение: каким образом будет отрабатываться переменная размерность таблицы и как будут считаться в ней строки. Примем сложное, но эффективное. Для того чтобы при освобождении строк таблицы всякий раз не перемещать элементы массива (уплотнение), введем в каждый из них признак «занятости».

```
//-----27-02.cpp
#define N 100
struct man{
char name[40];
double money;
int dd,mm,yy;
int busy;
} BS[N];           // Статический массив структур - таблица
//----- Очистка таблицы
void clear( man *p, int n) { for (int i=0; i<n; p++,i++) p->busy=0; }
```

Для извлечения строки таблицы (записи) по ее последовательному (логическому) номеру, а не по индексу, который совершенно не важен для пользователя, необходимо отсчитать заданное количество элементов массива, пропуская пустые.

```
//-----27-03.cpp
man *getnum( man *p, int n, int num){
for (int i=0; i<n; p++,i++)
    if (p->busy==1 && num--==0) return p;
return NULL;}
```

Иногда все-таки для упрощения некоторых операций, например, сортировки, потребуется уплотнить массив. Эти функции используют в своей работе присваивание структур.

```
//-----27-04.cpp
void ord( man *p, int n){
man *q=p; // Указатель на уплотненную часть
for (int i=0; i<n; p++,i++)
    if (p->busy==1) *q++ = *p; // Перезапись структур
}
void sort( man *p, int n){
int k;
ord(p,k); // Предварительно уплотнить
do {
    k=0; // Сортировка до первого незанятого
    for (int i=1; i<n && p[i].busy==1; i++)
        if (p[i].dd > p[i-1].dd){
            man x=p[i]; p[i]=p[i-1]; p[i-1]=x; k++; }
    } while(k);}
```

Для ввода новой записи в конец последовательности необходимо взять следующую за последней занятой. Если же последняя занятая находится в конце массива, то массив нужно попытаться уплотнить и повторить операцию.

```
//-----27-05.cpp
man *getfree( man *p, int n){
for (int k=-1,i=0; i<n; i++)
    if (p[i].busy==1) k=i; // Индекс последней занятой
if (k<n) return p+k+1; // Адресная арифметика man* + int
ord(p,n); // Уплотнить
for (k=-1,i=0; i<n; i++) // Повторить
    if (p[i].busy==1) k=i; // Индекс последней занятой
if (k<n) return p+k+1; // Адресная арифметика man* + int
return NULL; } // Все заняты
```

Группа функций, работающих с отдельной структурированной переменной, получает ее через указатель. Проверка его на **NULL**, как это будет видно ниже, используется для включения функций в цепочку вызовов, предусматривающих отрицательный результат выбора (например, несуществующий номер записи).

```
//-----27-06.cpp
void get( man *p){
if (p==NULL) return;
printf(" Name:"); gets(p->name);
p-> busy=1;}
```

```
//-----
void put(man *p){
if (p==NULL) return;
if (p->busy==0) return;
printf("Name:%s\n",p->name);}
```

Для компактного представления основной функции в отдельные модули выносятся даже мелочи типа ввода номера строки и подтверждения выхода.

```
//-----27-07.cpp
int num() { int n; printf("Номер:"); scanf("%d",&n); return n; }
int exit() {
char value; printf("Вы уверены?");
value=getch();
if(value=='Y' || value=='y') return 1;
return 0; }
```

Основная функция представляет собой «вечный цикл», в котором запрашивается очередное действие и выполняется через вызовы необходимых функций.

```
//-----27-08.cpp
void main() {
man *s; int i;
clear(BS,N);
while(1) {
printf("0- get, 1-show 2-del 3-edit 4-sort:");
switch(getch()){
case '0': get(getfree(BS,N)); break;
case '1': for (i=0; i<N; i++) put(&BS[i]); getch(); break;
case '2': s=getnum( BS,N,num());
if (s!=NULL) s->busy=0; break;
case '3': s=getnum( BS,N,num());
if (s!=NULL) {put(s); getch(); } break;
case 4 : sort(BS,N); break;
case 'e': if( exit())return;break;
default : get(getfree(BS,N));break;
}}}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Определить структурированный тип и набор функций для работы с таблицей, реализованной в массиве структур. Выбрать способ организации массива: с отметкой свободных элементов специальным значением поля либо с перемещением их к концу массива (уплотнение данных). Функции должны работать с массивом структур или с отдельной структурой через указатели, а также при необходимости возвращать указатель на структуру. В перечень функций входят:

- «очистка» структурированных переменных;
- поиск свободной структурированной переменной;

- ввод элементов (полей) структуры с клавиатуры;
- вывод элементов (полей) структуры с клавиатуры;
- поиск в массиве структуры с минимальным значением заданного поля;
- сортировка массива структур в порядке возрастания заданного поля;
- поиск в массиве структур элемента с заданным значением поля или с наиболее близким к нему по значению;
- удаление заданного элемента;
- изменение (редактирование) заданного элемента;
- сохранение содержимого массива структур в текстовом файле и загрузка из текстового файла;
- вычисление с проверкой и использованием всех элементов массива по заданному условию и формуле (например, общая сумма на всех счетах) – дается индивидуально.

Перечень полей структурированной переменной:

1. Фамилия И.О., дата рождения, адрес.
2. Фамилия И.О., номер счета, сумма на счете, дата последнего изменения.
3. Номер страницы, номер строки, текст изменения строки, дата изменения.
4. Название экзамена, дата экзамена, фамилия преподавателя, количество оценок, оценки.
5. Фамилия И.О., номер зачетной книжки, факультет, группа,
6. Фамилия И.О., номер читательского билета, название книги, срок возврата.
7. Наименование товара, цена, количество, процент торговой надбавки.
8. Номер рейса, пункт назначения, время вылета, дата вылета, стоимость билета.
9. Фамилия И.О., количество оценок, оценки, средний балл.
10. Фамилия И.О., дата поступления, дата отчисления.
11. Регистрационный номер автомобиля, марка, пробег.
12. Фамилия И.О., количество переговоров (для каждого – дата и продолжительность).
13. Номер телефона, дата разговора, продолжительность, код города.
14. Номер поезда, пункт назначения, дни следования, время прибытия, время стоянки.
15. Название кинофильма, сеанс, стоимость билета, количество зрителей.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определить значения переменных после выполнения действий, а также содержимое формируемых элементов структуры:

```
//-----27-09.cpp
//-----
struct man {
    char name[20];
    int dd,mm,yy;
    char *zodiak;
} A= { "Иванов",1,10,1969,"Весы" }, B[10], *p;
//----- 1
void F1() {
    char c; int i;
    for (i=0; i<10; i++) B[i].zodiak = "abcdefghij" + i;
    c = B[1].zodiak[2]; }
//----- 2
void F2() {
    char c; int i,j;
    for (i=0; i<10; i++) {
        for (j=0; j<10; j++) {
            B[i].name[j] = 'a' + i + j;
            B[i].name[j]='\0'; }
    c = B[1].name[2]; }
//----- 3
void F3() {
    int i,n,s;
    for (i=0; i<10; i++) B[i].dd = i;
    for ( s=0,p=B, n=5; n!=0; n--, p++)
        s += p->dd; }
//----- 4
void F4() {
    char c; int i;
    for (i=0; i<10; i++) B[i].zodiak = A.zodiak + i % 4;
    c = B[5].zodiak[2]; }
//----- 5
void F5() {
    int i,n; char *p;
    for (i=0; i<10; i++) B[i].zodiak = "abcdefghij" + i;
    for (n=0, p=B[6].zodiak; *p !='\0'; p++, n++); }
```

Определить значения переменных после выполнения действий над статическими данными.

```
//-----27-10.cpp
//----- 1
struct man1 {
    char name[20];
    int dd,mm,yy;
    char *zodiak;
    man1 *next; }
A1= {"Петров",1,10,1969,"Весы",NULL },
B1= {"Сидоров",8,9,1958,"Дева",&A1 },
*p1 = &B1;
void F1() { char c1,c2,c3,c4;
```

```

c1 = A1.name[2]; c2 = B1.zodiak[3];
c3 = p1->name[3]; c4 = p1->next->zodiak[1]; }
//----- 2
struct man2 {
char name[20];
char *zodiak;
man2 *next;
    } C2[3] = {
    {"Петров", "Весы", NULL },
    {"Сидоров", "Дева", &C2[0] },
    {"Иванов", "Козерог", &C2[1] }
};
void F2() { char c1,c2,c3,c4;
c1 = C2[0].name[2];    c2 = C2[1].zodiak[3];
c3 = C2[2].next->name[3];
c4 = C2[2].next->next->zodiak[1]; }
//----- 3
struct tree3 {
int vv; tree3 *l,*r;
    } A3 = { 1,NULL,NULL },
B3 = { 2,NULL,NULL },
C3 = { 3, &A3, &B3 },
D3 = { 4, &C3, NULL },
*p3 = &D3;
void F3() { int i1,i2,i3,i4;
i1 = A3.vv; i2 = D3.l->vv; i3 = p3->l->r->vv; i4 = p3->vv; }
//----- 4
struct tree4 {
int vv;
tree4 *l,*r;
} F[4] =
    {{ 1,NULL,NULL },
    { 2,NULL,NULL },
    { 3, &F[0], &F[1] },
    { 4, &F[2], NULL }};
void F4() { int i1,i2,i3,i4;
i1 = F[0].vv; i2 = F[3].l->vv;
i3 = F[3].l->r->vv; i4 = F[2].r->vv; }
//----- 5
struct list5 {
int vv;
list5 *pred,*next;
};
extern list5 C5,B5,A5;
list5 A5 = { 1, &C5, &B5 }, B5 = { 2, &A5, &C5 },
C5 = { 3, &B5, &A5 }, *p5 = &A5;
void F5() { int i1,i2,i3,i4;
i1 = A5.next->vv; i2 = p5->next->next->vv;
i3 = A5.pred->next->vv; i4 = p5->pred->pred->pred->vv; }
//----- 6
char *p6[] = { "Иванов", "Петров", "Сидоров", NULL };
void F6() { char c1,c2,c3,c4;
c1 = *p6[0]; c2 = *(p6[1]+2);
c3 = p6[2][3]; c4 = (*(p6+2))[1]; }
//----- 7
struct dat7 { int dd,mm,yy; }
aa = { 17,7,1977 },
bb = { 22,7,1982 };

```

```

struct man7 {
char name[20];
dat7 *pd;
dat7 dd;
char *zodiak; }
    A7= {"Петров", &aa, { 1,10,1969 }, "Весы" },
    B7= {"Сидоров", &bb, { 8,9,1958 }, "Дева" },
*p7 = &B7;
void F7() { int i1,i2,i3,i4;
i1 = A7.dd.mm; i2 = A7.pd->yy;
i3 = p7->dd.dd; i4 = p7->pd->yy; }
//----- 8
struct dat8 { int dd,mm,yy; };
struct man8 {
char name[20];
dat8 dd[3];
    } A8[2] = {
        {"Петров", {{1,10,1969},{8,8,1988},{3,2,1978}}},
        {"Иванов",{{8,12,1958},{12,3,1976},{3,12,1967}}}
};
void F8() { int i1,i2;
i1 = A8[0].dd[0].mm; i2 = A8[1].dd[2].dd; }
//----- 9
struct man9 {
char name[20];
char *zodiak;
man9 *next;
    } A9= {"Петров","Весы",NULL },
B9= {"Сидоров","Дева",&A9 },
*p9[4] = { &B9, &A9, &A9, &B9 };
void F9() { char c1,c2,c3,c4;
c1 = p9[0]->name[2]; c2 = p9[2]->zodiak[3];
c3 = p9[3]->next->name[3]; c4 = p9[0]->next->zodiak[1]; }

```

2.8. ТИПЫ ДАННЫХ, ПЕРЕМЕННЫЕ, ФУНКЦИИ

Конечно, мама, чтобы не ударить лицом в грязь перед врачами, сама начала изучать язык, на котором пишутся лекарства. Для этого она собрала все рецепты, склеила их в книжечку, и получил учебник.

*Е. Чеповецкий. Непоседа,
Мякиш и Нетак*

Настало время свести воедино все интуитивно используемые понятия, касающиеся не только Си, но и большинства других языков программирования, — понятия, которые образуют установившийся стандарт нижнего уровня организации программы — типы данных, функции и переменные. Язык Си имеет здесь свою специфику. Во-первых, он жестко типизирован с привязкой при трансляции к каждому объекту (переменной или функции) раз и навсегда.

гда заданного типа данных. Во-вторых, способ определения этого типа довольно специфичен: он задается неявно, в контексте (окружении) тех операций, которые можно выполнить над объектом. Это создает дополнительную путаницу у начинающих: они зачастую путают синтаксис использования переменной в выражении и синтаксис ее определения, путают определение с объявлением, поскольку в том и другом случаях применяются одни и те же операции, единый синтаксис. Этот раздел рекомендуется для проверки того, насколько ваши сложившиеся воззрения на язык программирования соответствуют здравому смыслу и действительности. И наконец, изучение последующих разделов немыслимо без свободного оперирования понятиями и терминами.

ОБЩЕСИСТЕМНЫЕ ТЕРМИНЫ

<i>Программа</i> = данные (переменные) + алгоритм (функции).

Физический – реальный, имеющий место на аппаратном уровне, «на самом деле». Например, физический порядок размещения переменных в памяти – реальная последовательность их размещения.

Логический – создаваемый программными средствами, но имеющий под собой полный физический эквивалент. Например, логический порядок следования элементов, данных в структуре данных, – особый порядок, создаваемый программными средствами, обычно определяемый порядком обхода управляющей части структуры данных.

Виртуальный – кажущийся, создаваемый программными средствами, но не имеющий под собой физического эквивалента (или имеющий частично).

Статический – неизменный на стадии выполнения программы, следовательно, определяемый в процессе ее трансляции (или загрузки).

Динамический – изменяемый во время выполнения программы.

Определение (переменной, функции) – фрагмент программы, в котором дается описание объекта и его свойств и который приводит к трансляции объекта в его внутреннее представление в программе.

Объявление – информация транслятору о наличии объекта (и его свойствах), находящегося в недоступной на данный момент части программы.

Тип данных – форма представления данных, которая характеризуется способом организации данных в памяти, множеством допустимых значений и набором операций.

Тип данных – «идея» переменных определенного вида, заложенная в транслятор.

Сама **переменная** – это не что иное, как область памяти программы, в которой размещены данные в соответствующей форме представления, то есть определенного типа. Поэтому любая переменная в языке имеет раз и навсегда заданный тип. Область памяти всегда ассоциируется в трансляторе с именем переменной, поэтому можно дать более строгое определение:

Переменная – именованная область памяти программы, в которой размещены данные с определенной формой представления (типом).

Переменная = тип данных + память (имя) + значение (инициализация).

Инициализация – присваивание переменным во время трансляции начальных значений, которые сохраняются во внутреннем представлении программы и устанавливаются при загрузке программы в память перед началом ее работы.

Неявно (по умолчанию) – вариант действия, производимого транслятором при отсутствии упоминаний о нем в тексте программы.

ТИПЫ ДАННЫХ И ПЕРЕМЕННЫЕ

Базовые типы данных (БТД) – формы представления данных, заложенные в язык программирования «от рождения».

Базовые типы данных в Си – совпадают со стандартными формами представления данных в компьютере.

Производные типы данных (ПТД) – формы представления данных, конструируемые в программе из уже известных (базовых и определенных ранее) типов данных.

Виды производных типов данных в Си – массив, структура, указатель, функция.

Иерархия и конструирование типов данных. В Си используется общепринятый принцип иерархического конструирования типов данных. Имеется набор базовых типов данных, операции над которыми включены в язык программирования. Производные типы

данных конструируются в программе из уже известных, в том числе базовых, типов данных. Понятно, что в языке программирования отсутствуют операции для работы с производным типом данных в целом. Но для каждого способа его определения существует операция **выделения составляющего типа данных**.

Операция выделения составляющего типа данных – операция, выполнение которой над переменной производного типа данных приводит к извлечению составляющего ее типа данных. Или же производится переход к объекту того типа данных, на основании которого она определена:

- для массива – операция «**[]**» – извлечение элемента массива, переход от массива к его элементу;

- для структуры – операция «**«.**»

– извлечение элемента структуры, переход от структурированной переменной к ее элементу;

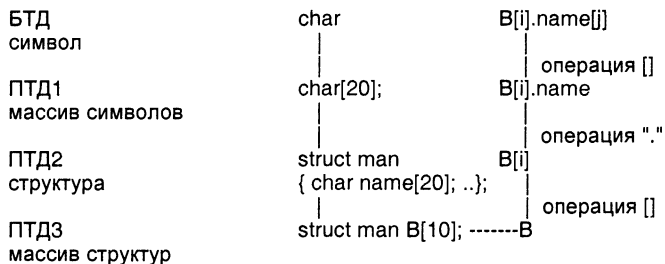
- для указателя – операция «*****» – косвенное обращение по указателю, разыменование указателя, переход от указателя к указываемому объекту. Сюда же относится операция **&** – переход от объекта к указателю на него;

- для функции – операция «**()**» – вызов функции, переход от функции к ее результату.

Пример иерархии типов данных и ее использования при работе с переменными. Прежде всего в программе создается цепочка определений производных типов данных: базовый тип данных используется для определения производного, который в свою очередь используется для определения другого производного типа данных и т.д. Затем определяется переменная, которая относится к одному из типов данных в этой цепочке. Под нее выделяется область памяти, которая получает общее имя. К этому имени могут быть применены операции выделения составляющих типов данных, они осуществляют переход к внутренним компонентам, составляющим переменную. Операции эти должны применяться в обратном порядке по отношению к последовательности определения типов данных. Типы полученных выражений также повторяют в обратном порядке эту последовательность.

Базовый тип **char** (БТД) используется для создания производного типа – массив из 20 символов (ПТД1). Тип данных – структура (ПТД2) использует массив символов в качестве одного из составляющих ее элементов. Последний тип данных – массив из 10 структур (ПТД3) порождает переменную **В** соответствующего типа. Затем все происходит в обратном порядке. Операции «**[]**», «**«.**» и **[]** последовательно выделяют в переменной **В** **i**-ю структуру, элемент структуры **name** и **j**-й символ в этом элементе.

```
struct man B[20];
char c;
c = B[i].name[j];
```



Если внимательно посмотреть на схему, то можно заметить, что в программе в явном виде упоминаются только два типа данных – базовый **char** и структура **struct man**. Остальные два типа – массив символов и массив структур – отсутствуют. Эти типы данных создаются «по ходу дела», в процессе определения переменной **B** и элемента структуры **name**.

Размерность типа данных. Любой тип данных в Си предполагает фиксированную размерность памяти создаваемых переменных. Эта размерность, выраженная в байтах, возвращается операцией **sizeof**, примененной по отношению к типу данных или к любой переменной этого типа.

«Источники» типов данных в Си:

- определение структурированного типа (**struct**) и класса (**class**);
- контекстное определение типа данных переменных;
- абстрактный тип данных;
- спецификатор **typedef**.

Определение структурированного типа. Первая часть определения структурированной переменной представляет собой определение структурированного типа. Оно задает способ построения этого типа данных из уже известных (типы данных элементов структуры). Имя структурированного типа данных (**man**) обладает всеми синтаксическими свойствами базового типа данных, то есть используется наряду с ними во всех определениях и объявлениях.

```
struct man {           // man - Имя структуры, имя типа данных
    char   name[20];   // Элементы структуры
    int    dd,mm,yy;
    char   *address;
} A, B, X[10];        // Определение структурированных переменных
```

Контекстное определение типа переменной – способ неявного определения типа данных переменной посредством включения ее в окружение (контекст) операций выделения составляющего типа данных (*,[],()), выполнение которых в соответствии с заданными приоритетами и скобками приводит к получению типа данных, стоящего в левой части определения.

Способ расшифровки контекста. Контекстное определение типа понимается следующим образом. Если взять переменную некоего неизвестного пока типа данных и выполнить над ней последовательность операций выделения составляющих типов данных, то в результате получится переменная того типа данных, который указан в левой части определения. При этом должны соблюдаться приоритеты выполнения операций, а для их изменения использоваться круглые скобки. Полученная последовательность выполнения операций дает обратную последовательность определений типов.

Использование контекстного способа определения типа объекта:

- определение и объявление переменных;
- формальные параметры функций;
- результат функции;
- определение элементов структуры (**struct**);
- определение абстрактного типа данных;
- определение типа данных (спецификатор **typedef**).

Примеры расшифровки контекста

```
int    *p;
```

Переменная, при косвенном обращении к которой получается целое, – указатель на целое.

```
char    *p[];
```

Переменная, которая является массивом, при косвенном обращении к элементу которого получаем указатель на символ (строку), – массив указателей на символы (строки).

```
char    (*p)[][80];
```

Переменная, при косвенном обращении к которой получается двумерный массив, состоящий из массивов по 80 символов, – указатель на двумерный массив строк по 80 символов в строке.

```
int      (*p)();
```

Переменная, при косвенном обращении к которой получается вызов функции, возвращающей в качестве результата целое, – указатель на функцию, возвращающую целое.

```
int      (*p[10])();
```

Переменная, которая является массивом, при косвенном обращении к элементу которого получается вызов функции, возвращающей целое, – массив указателей на функции, возвращающие целое.

```
char      *(*(*p)())();
```

Переменная, при косвенном обращении к которой получается вызов функции, при косвенном обращении к ее результату получается вызов функции, которая в качестве результата возвращает переменную, при косвенном обращении к которой получается символ, – указатель на функцию, возвращающую в качестве результата указатель на функцию, возвращающую указатель на строку.

Абстрактный тип данных. Используется в тех случаях, когда требуется обозначить некоторый тип данных как таковой, без привязки к конкретной переменной. Синтаксис абстрактного типа данных: берется контекстное определение переменной такого же типа, в котором само имя переменной отсутствует. Используется:

- в операции **sizeof**;
- в операторе создания динамических переменных **new**;
- в операции явного преобразования типа данных;
- при объявлении формальных параметров внешней функции с использованием прототипа.

Например, при резервировании памяти функцией нижнего уровня **malloc** для создания массива из 20 указателей необходимо знать размерность указателя **char***.

```
malloc(20*sizeof(char*))
```

Определение типа данных (спецификатор typedef). Спецификатор **typedef** позволяет в явном виде определить производный тип данных и использовать его имя в программе как обозначение этого типа, аналогично базовым (**int**, **char**...). В этом смысле он похож на определение структуры, в котором имя структуры (со служебным словом **struct**) становится идентификатором структурированного типа данных. Спецификатор **typedef** позволяет сделать то же самое для любого типа данных. Спецификатор **typedef** имеет синтаксис контекстного определения типа данных, в котором вместо имени переменной присутствует имя вводимого типа данных.

```
typedef char *PSTR;    // PSTR - имя производного типа данных
PSTR p,q[20],*pp;
```

Тип данных **PSTR** определяется в контексте как указатель на символ (строку). Переменная **p** типа **PSTR**, массив из 20 переменных типа **PSTR** и указатель типа **PSTR** представляют собой указатель на строку, массив указателей на строку и указатель на указатель на строку соответственно.

· ФУНКЦИЯ КАК ТИП ДАННЫХ

Определение функции состоит из двух частей: заголовка, создающего «интерфейс» функции к внешнему миру, и тела функции, реализующего заложенный в нее алгоритм с использованием внутренних локальных данных.

Заголовок включает в себя имя функции, по которому она идентифицируется и вызывается, списка формальных параметров в скобках и тип ее результата, который она возвращает.

```
// Заголовок: тип результата имя(параметр 1, параметр 2)
int      sum(int A[], int n)
//----- Тело функции (блок)
{ int    s,i;           // Локальные (автоматические) переменные блока
  for (i=s=0; i<n; i++) // Последовательность операторов блока
    s +=A[i];
  return s ; }          // Значение результата в return
```

Формальные параметры – собственные переменные функции, которым при ее вызове ставятся в соответствие (копируются, отображаются) фактические параметры. Синтаксис формальных параметров является синтаксисом **определения переменных** (контекстное определение типа).

Результат функции – временная переменная, которая возвращается функцией и используется как операнд в той части выражения, где был произведен ее вызов. Тип результата задан в заголовке функции тем же способом, что и для обычных переменных. Применяется синтаксис контекстного определения, в котором имя функции выступает в роли переменной-результата:

```
int      sum(...)        // Результат - целая переменная
char     *FF(...)         // Результат - указатель на символ
```

Значение переменной-результата устанавливается в операторе **return**, который производит это действие наряду с завершением выполнения функции и выходом из нее. После **return** может стоять любое выражение, значение которого и становится результатом

функции. Результат может иметь любой тип, кроме массива или функции.

Вызов функции – выполнение тела функции в той части выражения, где встречается имя функции со списком фактических параметров.

```
void main(){  
int ss, x, B[10]={ 1,6,3,4,5,2,56,3,22,3 };  
ss = x + sum(B,10); }  
// Вызов функции: ss = x + результат sum(фактические параметры)  
}
```

Фактические параметры – переменные, константы или выражения, значения которых ставятся в соответствие (отображаются, присваиваются) формальным параметрам. Фактические параметры имеют синтаксис **выражений (объектов программы)**.

Результат функции – **void**. Имеется специальный пустой тип результата – **void**, который обозначает, что функция не возвращает никакого результата. Оператор **return** в такой функции также не содержит никакого выражения, а результат не используется. Вызов такой функции важен выполняемыми внутри действиями.

```
void Call_me(){ puts("I am called"); ... return; }  
void main() { Call_me(); } // Просто вызов
```

Тело функции представляет собой блок, последовательность операторов, заключенную в фигурные скобки.

Локальные переменные – собственные переменные функции, используемые только алгоритмом в теле функции. В Си носят название **автоматических переменных** (см. ниже: «Модульное программирование»).

Глобальные переменные – переменные, определенные вне тел функций и одновременно доступные всем. В Си носят название **внешних переменных** (см. ниже: «Модульное программирование»).

Способы передачи параметров. Существуют два общепринятых способа установления соответствия между формальными и фактическим параметрами, способы передачи параметров **по значению** и **по ссылке**.

Передача параметра по значению осуществляется **копированием** значения фактического параметра в формальный, то есть присваиванием формальному параметру значения фактического. В Си параметры всех типов, за исключением массивов, неявно передаются по значению:

- формальные параметры являются собственными переменными функции;
- при вызове функции присваиваются значения фактических параметров формальным (копирование первых во вторые);
- при изменении формальных параметров значения соответствующих им фактических параметров не меняются.

Передача параметра по ссылке осуществляется **отображением** формального параметра в фактический. Массивы в Си всегда передаются по ссылке:

- формальные параметры существуют как синонимы фактических;
- при изменении формальных параметров значения соответствующих им фактических параметров меняются.

В Си существует также способ передачи параметров с использованием явной **ссылки** (см. раздел 2.6).

```
int sum(int s[], int n){           // Массив отображается, размерность копируется
for (int i=0; i<n; i++) z += s[i];
return z; }
```

```
int      c[10] = {1,6,4,7,3,56,43,7,55,33};
void main()
{ int nn; nn = sum(c,10); }
```

Функция main. В программе должна присутствовать функция, которая автоматически вызывается при загрузке программы в память и при ее выполнении. Более никаких особенностей, кроме указанной, эта функция не имеет.

Функция как тип данных. По правилам определения производных типов данных круглые скобки после имени объекта рассматриваются как примененная к нему операция вызова функции. С этой точки зрения функция является производным типом данных по отношению к своему результату, а операция вызова функции выделяет составляющий тип данных – результат из типа данных – функции (см. «Указатель на функцию», раздел 3.3).

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Модульное программирование – разработка программы в виде группы файлов исходного текста, их независимая трансляция в объектные модули и окончательная сборка в программный файл.

Модуль – файл Си-программы, транслируемый независимо от других файлов (модулей). Не путать с модулем в технологии структурного программирования.

Объектный модуль – файл данных, содержащий оттранслированные во внутреннее представление собственные функции и переменные, а также информацию об обращении к внешним данным и функциям (внешние ссылки) в исходном (символьном) виде.

Определение переменной – обычное контекстное определение, задающее тип, имя переменной, производящее инициализацию. При трансляции определения вычисляется размерность и резервируется память. Размерность массивов в определении обязательно должна быть задана.

```
int      a = 5 , B[10]={ 1,6,3,6,4,6,47,55,44,77 };
```

Объявление переменной имеет синтаксис определения переменной, предваренный словом **extern**. В нем задается тип и имя переменной, запоминается факт наличия переменной с указанными именем и типом. Размерность массивов в объявлении может отсутствовать

```
extern  int a,B[];
```

Время жизни переменной – интервал времени работы программы, в течение которого переменная существует, для нее отведена память и она может быть использована. Возможны три случая:

1) переменная создается функцией в стеке в момент начала выполнения функции и уничтожается при выходе из нее, переменная существует «от скобки до скобки»;

2) переменная создается транслятором при трансляции программы и размещается в программном модуле, такая переменная существует в течение всего времени работы программы, то есть «всегда»;

3) переменная создается и уничтожается работающей программой в те моменты, когда она «считает это необходимым», – динамические переменные (см. раздел 3.2).

Область действия переменной – та часть программы, где эта переменная может быть использована, то есть является доступной. Областью действия переменной могут быть:

- тело функции или блока, то есть «от скобки до скобки»;
- текущий модуль от места определения или объявления переменной до конца модуля, то есть в текущем файле;
- все модули программы.

Виды переменных (классы памяти) различаются в зависимости от сочетания основных свойств – времени жизни и области действия.

Автоматические переменные. Создаются при входе в функцию или блок и имеют областью действия тело той же функции или блока. При выходе уничтожаются. Место хранения – стек программы. Инициализация таких переменных заменяется обычным присваиванием значений при их создании. Если функция рекурсивна, то на каждый вызов создается свой набор таких переменных. В Паскале такие переменные называются **локальными** (общепринятый термин). Термин **автоматические** характеризует особенность их создания при входе в функцию, то есть время жизни. Синтаксис определения: любая переменная, определенная в начале тела функции или блока, по умолчанию является автоматической.

Внешние переменные. Создаются транслятором и имеют областью действия все модули программы. Размещаются транслятором в объектном модуле, а затем компоновщиком – в программном файле (сегменте данных) и инициализируются там же. Термин **внешние** характеризует доступность этих переменных из других модулей, или область действия. В Паскале такие переменные называются **глобальными** (общепринятый термин).

Синтаксис определения: любая переменная, определенная вне тела функции, по умолчанию является внешней.

Несмотря на то, что внешняя переменная потенциально доступна из любого модуля, сам факт ее существования должен быть известен транслятору. Если переменная определена в модуле, то она доступна от точки определения до конца файла. В других модулях требуется произвести объявление внешней переменной.

```
// Файл a.cpp - определение переменной
int      a,B[20]={1,5,4,7};
... область действия ...
```

```
// Файл b.cpp - объявление переменной
extern int a,B[];
... область действия ...
```

Определение переменной должно производиться только в одном модуле, при трансляции которого она создается и в котором размещается. Соответствие типов переменных в определении и объявлениях транслятором не может быть проверено. Ответственность за это соответствие ложится целиком на программиста.

Статические переменные. Имеют сходные с внешними переменными характеристики времени жизни и размещения в памяти, но ограниченную область действия.

Собственные статические переменные функции имеют синтаксис определения автоматических переменных, предваренный словом **static**. Область действия аналогична автоматическим – тело функции или блок. При рекурсивном вызове функции не дублируются. Назначение собственных статических переменных – сохранение значений, используемых функцией, между ее вызовами.

Статические переменные, определенные вне функции, имеют область действия, ограниченную текущим модулем. Они предназначены для создания собственных переменных модуля, которые не должны быть «видны» извне, чтобы не вступать в конфликт с одноименными внешними переменными в других модулях.

Определение функции – обычное задание функции в программе в виде заголовка и тела, по которому она транслируется во внутреннее представление в том модуле, где встречается.

Объявление функции – информация транслятору о наличии функции с заданным заголовком (прототипом) либо в другом модуле, либо далее по тексту текущего модуля – «вниз по течению». Объявление функции состоит из прототипа, предваренного словом **extern**, либо просто из прототипа функции.

Прототип функции – заголовок функции со списком формальных параметров, заданных в виде **абстрактных типов данных**.

```
int clrscr();           // Без контроля соответствия (анахронизм)
int clrscr(void);       // Без параметров
int strcmp(char*, char*);
extern int strcmp();    // Без контроля соответствия (анахронизм)
extern int strcmp(char*, char*);
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите вид объекта (переменная, функция), задаваемого в контекстном определении или объявлении, а также все неявно заданные типы данных.

```
//-----28-02.cpp
//-----1
char f(void);
//-----2
char *f(void);
//-----3
int (*p[5])(void);
//-----4
void ( (*p)(void) )(void);
//-----5
int (*f(void))();
```

```

//-----6
char **f(void);
//-----7
typedef char *PTR;
PTR a[20];
//-----8
typedef void (*PTR)(void);
PTR F(void);
//-----9
typedef void (*PTR)(void);
PTR F[20];
//-----10
struct list {...};
list *F(list *);
//-----11
void **p[20];
//-----12
char *(*pf)(char *);
//-----13
int F(char *,...);
//-----14
char **F(int);
//-----15
typedef char *PTR;
PTR F(int);

```

Найдите абстрактный тип данных и определите назначение.

```

//-----28-03.cpp
//-----1
char **p = (char**)malloc(sizeof(char *) * 20);
//-----2
char **p = (char**)malloc(sizeof(char * [20]));
//-----3
char **p = new char*[20];
//-----4
double d=2.56; double z=d-(int)d;
//-----5
long l;
((char *)&l) [2] = 5;
//-----6
extern int strcmp(char *, char *);

```

Найдите, где задано определение, объявление и вызов функции.

```

//-----28-04.cpp
//-----1
void F(void) { puts("Hello, Dolly"); }
//-----2
void F(void) { puts("Hello, Dolly"); }
void G(void){ F(); }
//-----3
void F(void);
void G(void){ F(); }
//-----4
void G(void){
void F(void);
F(); }

```

```

void F(void) { puts("Hello, Dolly"); }
//-----5
void F(void);
void G(void){ F(); }
void F(void) { puts("Hello, Dolly"); }
//-----6
extern void F(void);
void G(void){ F(); }

```



3. ПРОГРАММИСТ «СИСТЕМНЫЙ»

При переходе от уровня начинающего должен произойти качественный скачок в отношении процесса программирования. Прежде всего, должна быть отработана и адаптирована «под себя» технология нисходящего проектирования программ и данных. Необходимо также почувствовать, что программирование – это не столько написание отдельной программы, сколько процесс ее построения из множества взаимодействующих модулей, создания их иерархии, проектирования различных типов данных. И, наконец, необходимо научиться «кромсать» готовые алгоритмы, чтобы использовать стандартные программные решения, как на уровне вызова функций, так и на уровне использования алгоритмов.

Хорошим полигоном для овладения этими навыками являются структуры данных – традиционный раздел системного программирования. И хотя нормальный пользователь может при желании найти стандартные средства работы с ними, вопрос «Как это делается?» тоже достаточно интересен. Объем этого раздела позволяет вплотную приблизиться к пониманию того, как организованы базы данных, какие структуры данных и алгоритмы работы с ними используют операционные системы в своих внутренних механизмах. То есть освоить то, что отличает системного программиста от прикладного.

И последняя цель. Системный программист – не тот, кто гордится знаниями различных «хитростей» и способов проникновения в чужие системы. Это программист, озабоченный эффективностью работы своей программы и использования ею различных ресурсов (прежде всего памяти), понимающий ее проблемы и нужды на архитектурном уровне.

3.1. УКАЗАТЕЛИ И УПРАВЛЕНИЕ ПАМЯТЬЮ

Управление памятью в языках высокого уровня. Под управлением памятью имеются в виду возможности программы по размещению данных и по манипулированию ими. Поскольку единственным «представителем» памяти в программе выступают переменные, то управление памятью определяется тем, каким образом работает с ними и с образованными ими структурами данных язык программирования. Большинство языков программирования однозначно закрепляет за переменными их типы данных и ограничивает работу с памятью только областями, где эти переменные размещены. Программист не может выйти за пределы самим же определенного шаблона структуры данных. С другой стороны, это позволяет транслятору обнаруживать допущенные ошибки как в процессе трансляции, так и в процессе выполнения программы.

В языке Си ситуация принципиально иная по двум причинам. Во-первых, наличие операции адресной арифметики при работе с указателями позволяет, в принципе, выйти за пределы памяти, выделенной транслятором под указываемую переменную, и адресовать память как «до», так и «после» нее. Другое дело, что это должно делаться осознанно и корректно. Во-вторых, присваивание и преобразование указателей различных типов, речь о котором пойдет ниже, позволяет рассматривать одну и ту же память «под различным углом зрения» в смысле типов заполняющих ее переменных.

Присваивание указателей различного типа. Операцию присваивания указателей различных типов следует понимать как назначение указателя в левой части на ту же самую область памяти, на которую назначен указатель в правой. Но поскольку тип указываемых переменных у них разный, то эта область памяти по правилам интерпретации указателя будет рассматриваться как заполненная переменными либо одного, либо другого типа (рис. 3.1).

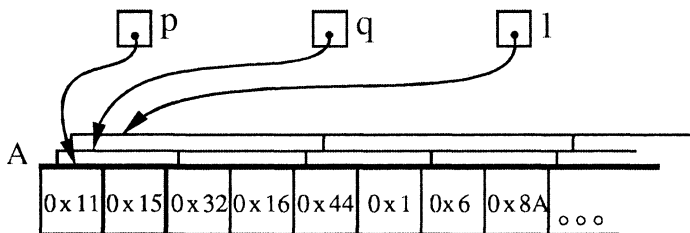


Рис. 3.1

```
char A[20]={0x11,0x15,0x32,0x16,0x44,0x1,0x6,0x8A};
char *p; int *q; long *l;
p = A; q = (int*)p; l = (long*)p;
p[2] = 5;           // Записать 5 во второй байт области A
q[1] = 7;           // Записать 7 в первое слово области A
```

Здесь **p** – указатель на область байтов, **q** – на область целых, **l** – на область длинных целых. Соответственно операции адресной арифметики ***(p+i)**, ***(q+i)**, ***(l+i)** или **p[i]**, **q[i]**, **l[i]** адресуют **i**-й байт, **i**-е целое и **i**-е длинное целое от начала области. Область памяти имеет различную структуру (байтовую, словную и т.д.) в зависимости от того, через какой указатель мы с ней работаем. При этом неважно, что сама область определена как массив типа **char**, – это имеет отношение только к операциям с использованием идентификатора массива.

Присваивание значения указателя одного типа указателю другого типа сопровождается действием, которое называется в Си **преобразованием типа указателя** и в Си++ обозначается всегда явно. Операция **(int*)p** меняет в текущем контексте тип указателя **char*** на **int***. На самом деле это действие – чистая фикция (команды транслятором не генерируются). Транслятор просто запоминает, что тип указуемой переменной изменился и операции адресной арифметики и косвенного обращения нужно выполнять с учетом нового типа указателя.

Явное преобразование типа указателя в выражении. Преобразование типа указателя можно выполнить не только при присваивании, но и внутри выражения, «на лету». В этом случае текущий указатель меняет тип указуемого элемента только в цепочке выполняемых операций.

```
char A[20]; ((int *)A)[2] = 5;
```

Имя массива **A** – указатель на его начало – имеет тип **char***, который явно преобразуется в **int***. Тем самым в текущем контексте мы ссылаемся на массив как на область целых переменных. Применительно к указателю на массив целых выполняются операции индексации и последующего присваивания. Результат: целое 5 записывается во второй элемент целого массива, размещенного в **A**.

Операция ***p++** применительно к любому указателю интерпретируется как «взять указуемую переменную и перейти к следующей», таким образом, значением указателя после выполнения операции будет адрес переменной, следующей за выбранной. Использование такой операции в сочетании с явным преобразованием типа позволяет извлекать или записывать переменные различных типов, последовательно расположенных в памяти.

```
char A[20], *p=A;
*p++ = 5;           // Записать в массив байт с кодом 5
*((int*)p)++ = 5;    // Записать в массив целое 5
*((double*)p)++ = 5.5; // Записать в массив вещественное 5.5
```

Работа с памятью на низком уровне. Операции преобразования типа указателя и адресной арифметики дают Си невиданную для языков высокого уровня свободу действий по управлению памятью. Традиционно языки программирования, даже если они работают с указателями или с их неявными эквивалентами – ссылками, не могут выйти за пределы единожды определенных типов данных для используемых в программе переменных. Напротив, в Си имеется возможность работать с памятью на «низком» уровне (можно сказать, ассемблерном или архитектурном). На этом уровне программист имеет дело не с переменными, а с помеченными областями памяти, внутри которых он размещает данные любых типов и в любой последовательности, в какой только пожелает. Естественно, что при этом ответственность за корректность размещения данных ложится целиком на программиста.

Операция **sizeof** вызывает подстановку транслятором соответствующего значения размерности указанного в ней типа данных в байтах. С этой точки зрения она является универсальным измерителем, который должен использоваться для корректного размещения данных различных типов в памяти.

Работа с последовательностью данных, определяемой форматом. Массив можно определить как последовательность переменных одного типа, структуру – как фиксированную последовательность переменных различных типов. Но существуют данные иного рода, в которых заранее неизвестны ни типы переменных, ни их количество, а заданы только общие правила их следования (формат). В таком формате значение предыдущей переменной может определять тип и количество расположенных за ней переменных.

Последовательности данных, определяемых форматом, широко используются при упаковке больших массивов, при представлении объектов с переменной размерностью и произвольными свойствами и т.д. При работе с ними требуется последовательно просматривать область памяти, извлекая из нее переменные разных типов, и на основе анализа их значений делать вывод о типах, следующих за ними. Такая задача решается с использованием операции явного преобразования типа указателя.

Другой вариант заключается в использовании объединения (**union**), которое, как известно, позволяет использовать общую память для размещения своих элементов. Если элементами объеди-

нения являются указатели, то операции присваивания можно исключить.

```
union ptr {
    int      *p;
    double   *d;
    long     *l; } PTR;
int A[100];
PTR.p=A;
*(PTR.p)++; *(PTR.l)++; *(PTR.d)++;
```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Размещение вещественного массива в заданной памяти. Массив байтов (тип **char**) заполняется вещественными переменными. Для этого необходимо преобразовать начальный адрес массива в указатель типа **double*** и «промерить» имеющийся массив с использованием операции **sizeof**.

```
#define N 100
double *d;
char A[N];
int sz = N / sizeof(double); // Количество вещественных в массиве байтов
for (i=0, d=(double*)A; i < sz; i++)
    d[i] = (double)i;
```

Фрагмент системы динамического распределения памяти. Свободные области динамически распределяемой памяти составляют двусвязный циклический список. Элемент списка – это заголовок и следующая непосредственно за ним свободная (распределяемая) область. При выделении памяти по принципу наиболее подходящего выделенная область делится на две части: первая сохраняет элемент списка, содержащего «остаток», а во второй создается новый элемент списка, и она возвращается в виде выделенной области (рис. 3.2).

```
//-----31-00.cpp
#define N 10 // Наименьшая распределяемая область
struct item{
    item *next,*prev; // Указатели в списке
    int size; // Размер следующей свободной области
} *ph; // Заголовок списка свободных областей
void *mymalloc(int sz){
    item *pmin,*q;
    for (pmin=q=ph; q!=NULL; q=q->next)
        if (q->size >=sz && q->size < pmin->size)
            pmin=q; // Указатель на наиболее близкий по размеру
    // Выделить полностью, если совпадает точно или остаток меньше N
    if (pmin->size==sz || pmin->size-sz < sizeof(item)+N){
        if (pmin->next==pmin) ph=NULL; // Исключение из списка
        else {
            pmin->next->prev=pmin->prev;
```

```

        pmin->prev->next=pmin->next;
    }
    return (void*)(pmin+1);    //Выделенная область "вслед за..."
}
else {                        // Новый элемент - в "хвосте"
    pmin->size -= sz + sizeof(item); // Размерность остатка
    item *pnew=(item*)((char*)(pmin+1) + pmin->size);
    pnew->size=sz;            // Адрес и размерность нового элемента
    return (void*)(pnew+1);    // Вернуть область нового элемента
}

```

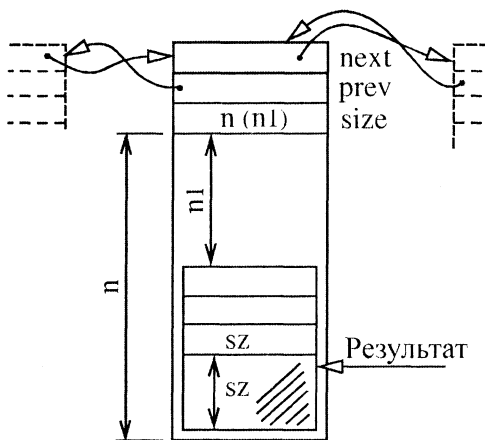


Рис. 3.2

Упаковка последовательности нулей. Программа упаковывает массив вещественных чисел, «сворачивая» последовательности подряд идущих нулевых элементов. Формат упакованной последовательности:

- последовательность ненулевых элементов кодируется целым счетчиком (типа **int**), за которым следуют сами элементы;
- последовательность нулевых элементов кодируется отрицательным значением целого счетчика;
- нулевое значение целого счетчика обозначает конец последовательности;

Примеры неупакованной и упакованной последовательностей:
 2.2, 3.3, 4.4, 5.5, 0.0, 0.0, 0.0, 1.1, 2.2, 0.0, 0.0, 4.4 и **4**, 2.2, 3.3, 4.4, 5.5, **-3**, **2**, 1.1, 2.2, **-2**, **1**, 4.4, **0**.

В процессе упаковки подсчитывается количество подряд идущих нулей. В выходной последовательности запоминается место расположения последнего счетчика – также в виде указателя. Смена счетчика происходит, если текущий и предыдущий элементы

относятся к разным последовательностям (комбинации «нулевой – ненулевой» и наоборот). Для записи в последовательность ненулевых значений из вещественного массива используется явное преобразование типа указателя **int*** в **double***.

```
//-----31-01.cpp
//----- Упаковка массива с нулевыми элементами
void pack(int *p, double v[], int n)
{ int *pcnt=p++; // Указатель на последний счетчик
  *pcnt=0; // Обнулить последний счетчик
  for (int i=0; i<n; i++)
  { // Смена счетчика
    if (i!=0 && (v[i]==0 && v[i-1]!=0) || v[i]!=0 && v[i-1]==0)
    { pcnt=p++; *pcnt=0; } // Обнулить последний счетчик
    if (v[i]==0) (*pcnt)--; // -1 к счетчику нулевых
    else {
      (*pcnt)++ ; // +1 к счетчику ненулевых
      double *q=(double*)p; // Сохранить само значение
      *q+=v[i]; p=(int*)q; }
  }
  *p++ = 0;}
//----- Распаковка массива с нулевыми элементами
int unpack(int *p, double v[])
{ int i=0,cnt;
  while ((cnt= *p++)!=0) // Пока нет нулевого счетчика
  {
    if (cnt<0) // Последовательность нулей
      while(cnt++!=0) v[i++]=0;
    else // Ненулевые элементы
      while(cnt--!=0) // извлечь с преобразованием
        double *q=(double*)p; // типа указателя
        v[i++] = *q++; p=(int*)q; }
  return i;}

```

Функции с переменным числом параметров. Формальные параметры представляют собой «ожидаемые» смещения в стеке относительно текущего положения указателя стека, по которым после вызова должны находиться соответствующие фактические параметры. Фактические параметры – реальные переменные, создаваемые в стеке перед вызовом функции. Такой механизм вызова устанавливает соответствие параметров только «по договоренности» между вызывающей и вызываемой функциями, а компилятор при использовании прототипа проверяет эти соглашения. Если в заголовке функции список формальных параметров заканчивается переменным списком (обозначенным как «...»), то компилятор просто прекращает проверку соответствия, допуская наличие в стеке некоторого «хвоста» из последовательности фактических параметров. Извлекаются они с помощью соответствующих макрокоманд. То же самое можно сделать, используя указатель на последний из явно определенных формальных параметров, рассматривая тем самым область стека как адресуемую указателем память.

Продвигая указатель по этой памяти, можно явным образом эти параметры извлекать.

```
void var_list_fun(int a1, int a2, int a3,...){
int *p=&a3;           // Указатель на последний явный параметр функции
int *q=&a3+1;         // Указатель на первый из переменного списка
...
}
```

Текущее количество фактических параметров, передаваемых при вызове, передается:

- отдельным параметром-счетчиком;
- параметром-ограничителем, значение которого отмечает конец списка параметров;
- форматной строкой, в которой перечислены спецификации параметров.

Функция с параметром-счетчиком. Первый параметр является счетчиком, определяющим количество параметров в переменном списке.

```
//-----31-02.cpp
//---- Сумма произвольного количества параметров по счетчику
int sum(int n,...)           // n - счетчик параметров
{ int s,*p = &n+1;           // Указатель на область параметров
  for (s=0; n > 0; n--)       // назначается на область памяти
    s += *p++;               // вслед за счетчиком
  return(s); }
void main(){ printf("sum(..=%d
sum(...=%d\n",sum(5,0,4,2,56,7),sum(2,6,46)); }
```

Функция с параметром-ограничителем. Указатель настраивается на первый параметр из списка, извлекая последующие до тех пор, пока не встретит значение-ограничитель.

```
//-----31-03.cpp
//---- Сумма произвольного количества ненулевых параметров
int sum(int a,...)
{ int s,*p = &a; // Указатель на область параметров назначается на
  for (s=0; *p > 0; p++ ) // первый параметр из переменного списка
    s += *p;             // Ограничитель - отрицательное
  return(s); }           // значение
void main() {
printf("sum(..=%d sum(...=%d\n",sum(4,2,56,7,0),sum(6,46,-1 ,7,0)); }
```

Функция с параметром – форматной строкой. Если в списке предполагается наличие параметров различных типов, то типы их могут быть переданы в функцию отдельной спецификацией (подобно форматной строке функции **printf**). В этом случае область фактических параметров представляет собой память, в которой последовательность переменных задается внешним форматом, а извлекаются они преобразованием типа указателя.

```
//-----31-04.cpp
//--- Функция с параметром форматной строкой ( printf)
int my_printf(char *s,...)
{ int *p = (int*)&s+1; // Указатель на начало списка параметров
while (*s != '\0') { // Просмотр форматной строки
    if (*s != '%') putchar(*s++); // Копирование форматной строки
    else { s++; // Спецификация параметра вида "%d"
        switch(*s++){ // Извлечение параметра
        case 'c': putchar(*p++); break; // Извлечение символа
        case 'd': printf( "%d", *((int*)p));
                    p+=sizeof(int); break; // Извлечение целого
        case 'f': printf( "%lf", *((double*)p));
                    p+=sizeof(double); break; // Извлечение вещественного
        case 's': puts( *((char**)p));
                    p+=sizeof(char*); // Извлечение указателя
                    break; // на строку
        }}}
void main(){my_printf("int=%d
double=%f char[]={%s char=%c ",44,5.5,"qwerty",'f');}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Разработать две функции, одна из которых вводит с клавиатуры данные в произвольной последовательности и размещает в памяти в переменном формате. Другая функция читает эти данные и выводит на экран.

1. Последовательность прямоугольных матриц вещественных чисел, предваренная двумя целыми переменными – размерностью матрицы.

2. Последовательность строк символов. Каждая строка предваряется целым – счетчиком символов. Ограничение последовательности – счетчик со значением 0.

3. Упакованный массив целых переменных. Байт-счетчик, имеющий положительное значение **n**, предваряет последовательность из **n** различных целых переменных; байт-счетчик, имеющий отрицательное значение **-n**, обозначает **n** подряд идущих одинаковых значений целой переменной. Примеры:

– исходная последовательность: **2 3 3 5 2 4 4 4 4 8 -6 8**

– упакованная последовательность: **(1) 2 (-3) 3 (2) 5 2 (-5) 4 (3) 8 -6 8**

4. Упакованная строка, содержащая символьное представление длинных целых чисел. Все символы строки, кроме цифр, помещаются в последовательность в исходном виде. Последовательность цифр преобразуется в целую переменную, которая записывается в упакованную строку, предваренная символом **\1**. Конец строки – символ **\0**. Примеры:

– исходная строка: "aa2456bbbb6665"

– упакованная строка: 'a' 'a' '\1' 2456 'b' 'b' 'b' 'b' '\1' 6665 '\0'

5. Произвольная последовательность переменных типа **char**, **int** и **long**. Перед каждой переменной размещается байт, определяющий ее тип (**0-char**, **1-int**, **2-long**). Последовательность вводится в виде целых переменных типа **long**, которые затем «укорачиваются» до минимальной размерности без потери значащих цифр.

6. Последовательность структурированных переменных типа **struct man { char name[20]; int dd,mm,yy; char addr[]; };** Последний компонент представляет собой строку переменной размерности, расположенную непосредственно за структурированной переменной. Конец последовательности – структурированная переменная с пустой строкой в поле **name**.

7. То же самое, что п. 4, но для шестнадцатеричных чисел:

– исходная строка: "aa0x24FFbbb0xAA65"

– упакованная строка: 'a' 'a' '\1' 0x24FF 'b' 'b' 'b' '\1' 0xAA65 '\0'.

8. В упакованной строке последовательность одинаковых символов длиной **N** заменяется на байт со значением 0, байт со значением **N** и байт – повторяющийся символ. Конец строки обозначается через два нулевых байта.

9. Произвольная последовательность строк и целых переменных. Байт со значением 0 обозначает начало строки (последовательность символов, ограниченная нулем). Байт со значением **N** – начало последовательности **N** целых чисел. Конец последовательности – два нулевых байта.

10. В начале области памяти размещается форматная строка, аналогичная используемой в **printf (%d, %f и %s** целое, вещественное и строку соответственно). Сразу же вслед за строкой размещается последовательность целых, вещественных и строк в соответствии с заданным форматом.

11. В начале области памяти размещается форматная строка. Выражение «**%nnnd**», где **nnn** – целое, определяет массив из **nnn** целых чисел, «**%d**» – одно целое число, «**%nnnf**» – массив из **nnn** вещественных чисел, «**%f**» – одно вещественное число. Сразу же вслед за строкой размещается последовательность целых, вещественных и их массивов в соответствии с заданным форматом.

12. Область памяти представляет собой строку. Если в ней встречается выражение «**%nnnd**», где **nnn** – целое, то сразу же за ним следует массив из **nnn** целых чисел (во внутреннем представлении, то есть типа **int**). За выражением «**%d**» – одно целое число,

за «%nnnf» – массив из **nnn** вещественных чисел, за «%f» – одно вещественное число.

13. Область памяти представляет собой строку. Если в ней встречается символ «%», то сразу же за ним находится указатель на другую (обычную) строку. Все сроки располагаются в той же области памяти вслед за основной строкой.

14. Разреженная матрица (содержащая значительное число нулевых элементов) упаковывается с сохранением значений ненулевых элементов в следующем формате: размерности (**int**), количество ненулевых элементов (**int**), для каждого элемента – координаты **x**, **y** (**int**) и значение (**double**).

Разработать функцию с переменным количеством параметров. Для извлечения параметров из списка использовать операцию преобразования типа указателя.

15. Первый параметр – строка, в которой каждый символ «*» обозначает место включения строки, являющейся очередным параметром. Функция выводит на экран полученный текст.

16. Каждый параметр – строка, последний параметр – **NULL**. Функция возвращает строку в динамической памяти, содержащую объединение строк-параметров.

17. Последовательность указателей на вещественные переменные, ограниченная **NULL**. Функция возвращает упорядоченный динамический массив указателей на эти переменные.

18. Последовательность вещественных массивов. Сначала идет целый параметр – размерность массива (**int**), затем – непосредственно последовательность значений типа **double**. Значение целого параметра – 0 – обозначает конец последовательности. Функция возвращает сумму всех элементов.

19. Последовательность вещественных массивов. Сначала идет целый параметр – размерность массива (**int**), затем указатель на массив значений типа **double** (имя массива). Значение целого параметра – 0 – обозначает конец последовательности. Функция возвращает сумму всех элементов.

20. Первый параметр – строка, в которой каждый символ «***n**», где **n** – цифра, обозначает место включения строки, являющейся **n+1**-параметром. Функция выводит на экран полученный текст.

21. Первым параметром является форматная строка. Выражение «%nnnd», где **nnn** – целое, определяет массив из **nnn** целых чисел, «%d» – одно целое число, «%nnnf» – массив из **nnn** вещественных чисел, «%f» – одно вещественное число. Сразу же вслед за строкой размещается последовательность целых, вещественных

и их массивов в соответствии с заданным форматом. Массив передается непосредственно в виде последовательности параметров (например, «%4d%2f», 44, 66, 55, 33, 66.5, 66.7).

22. Первым параметром является форматная строка. Выражение «%nnnd», где **nnn** – целое, определяет массив из **nnn** целых чисел, «%d» – одно целое число, «%nnnf» – массив из **nnn** вещественных чисел, «%f» – одно вещественное число. Сразу же вслед за строкой размещается последовательность целых, вещественных и их массивов в соответствии с заданным форматом. Массив передается в виде указателя (имя массива) (например, «%4d%2f», A, B).

23. Первый параметр – строка, в которой каждый символ «***n**», где **n** – цифра, обозначает место включения целого (**int**), являющегося **n+1**-параметром. Функция выводит на экран полученный текст, содержащий целые значения.

24. Параметр функции – целое – определяет количество строк в следующей за ним группе. Групп может быть несколько. Целое со значением 0 – конец последовательности.

25. Функция получает разреженный массив, содержащий значительное число нулевых элементов, в виде списка значений ненулевых элементов в следующем формате: размерность массива (**int**), количество ненулевых элементов (**int**), для каждого элемента – индекс (**int**) и значение (**double**). Функция создает и возвращает динамический массив с соответствующим содержимым.

ИНДИВИДУАЛЬНЫЕ ПРОЕКТЫ

Разработать собственные функции динамического распределения памяти (ДРП), используя в качестве «кучи» динамический массив, создаваемый обычной функцией распределения памяти. Разработанная функция **malloc** должна возвращать указатель на выделенную область, причем в память *перед указателем* должен быть записан размер выделенной области, необходимый при ее возвращении, и сохранена другая необходимая системная информация. При освобождении памяти соседние свободные области объединяются.

1. Свободные области – односвязный список. Выделенные области – односвязный список. Выделение по принципу наиболее подходящего.

2. Свободные области – односвязный список. Первый элемент списка – исходная «куча». Если при поиске не находится элемента с размером, точно совпадающим с требуемым, новый элемент вы-

деляется из «кучи». Возвращаемые элементы не «склеиваются», а используются при повторном выделении памяти того же размера.

3. Свободные области – динамический массив указателей. Выделение по принципу первого подходящего.

4. Свободные области – динамический массив указателей. Первая свободная область – исходная «куча». Если при поиске не находится элемента с размером, точно совпадающим с требуемым, новый элемент выделяется из «кучи». Возвращаемые элементы не «склеиваются», а используются при повторном выделении памяти того же размера.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите значения переменных после выполнения операций.

Замечание: переменные размещаются в памяти, начиная с младшего байта.

```
//-----31-05.cpp
//-----33221100 распределение long по байтам
long ll=0x12345678; // sizeof(long)=4, sizeof(int)=2
char A[20] = {0x12,0x34,0x56,0x78,0x9A,0xBC,0xDE,0xF0,0x12};
int a1=((int*)A)[2];
int a2=((int*)(A+3))[1];
long a3=((long*)A)[1];
long a4=((long*)(A+1))[1];
```

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определить способ размещения последовательности переменных в общей области памяти, которая читается или заполняется функцией (формат последовательности данных). Для вызова функции задайте набор глобальных переменных (транслятор размещает их в соответствии с последовательностью их определения) и передайте ей указатель на первую из них.

Пример оформления тестового задания

```
//-----31-06.cpp
double F(int *p) // По умолчанию - извлекается int
{ double s=0; // Начальная сумма равна 0
while (*p!=0){ // Пока не извлечен нулевой int
    int n=*p++; // Очередной int - счетчик цикла
    double **ss=(double**)p;
    double *q=*ss++; // следующий за ним - double*
    p=(int*)ss;
    while (n--!=0) s+=*q++; // Суммирование массива
    } return s; } // под указателем q
double d1[]={1,2,3,4},d2[]={5,6};
int a1=4; // Размерность первого массива
double *q1=d1; // Указатель на первый массив double*
```

```

int a2=2;           // Размерность второго массива
double *q2=d2;      // Указатель на второй массив double*
int a3=0;           // Ограничитель последовательности
void main(){
printf("%lf\n",F(&a1)); // Должна вывести 21 - сумму d1 и d2
}

```

Функция работает с указателем **p**, извлекая из-под него целые переменные, пока не обнаружит 0. Очередная переменная запоминается в **p** и используется в дальнейшем в качестве счетчика повторения цикла, то есть определяет количество элементов в некотором массиве. В том же цикле суммируемые значения извлекаются из-под указателя **q** типа **double***, то есть речь идет о массиве вещественных. Остается определить, как формируется **q**. Он извлекается из той же последовательности, что и целые переменные, – с использованием **p**. Для этого последний преобразуется «на лету» в указатель на извлекаемый тип, то есть приводится к типу **double****. Таким образом, последовательность представляет собой пары переменных – целая размерность массива и указатель на сам вещественный массив. Размерность, равная 0, – ограничитель последовательности (рис. 3.3).

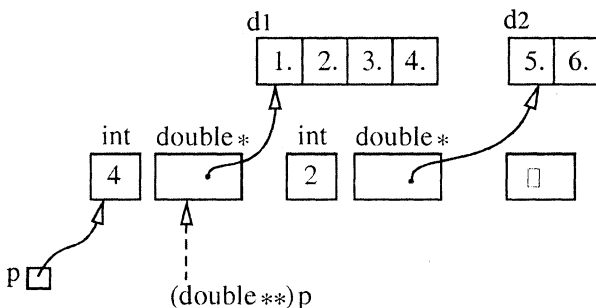


Рис. 3.3

```

//-----31-07.cpp
//----- 1
struct man {char name[20]; int dd,mm,yy; char *addr; };
char *F1(char *p, char *nm, char *ad)
{ man *q =(man*)p;
strcpy(q->name,nm);
strcpy((char*) (q+1 ),ad);
q->addr = (char*) (q+1 );
for (p=(char*) (q+1 ); *p!=0; p++);
p++; return p;}
//----- 2
struct man1 {char name[20]; int dd,mm,yy; char addr[]; };
char *F2(char *p, char *nm, char *ad)

```

```

{ man1 *q =(man1*)p;
strcpy(q->name,nm);
strcpy(q->addr,ad);
for (p=q->addr; *p!=0; p++);
p++; return p;}
//----- 3
int *F3(int *q, char *p[])
{ char *s;
for ( int i=0; p[i]!=NULL; i++);
*q = i;
for (s = (char*)(q+1), i=0; p[i]!=NULL; i++) {
    for ( int j=0; p[i][j]!='\0'; j++) *s++ = p[i][j];
    *s++ = '\0';
}
return (int*)s;}
//----- 4
double F4(int *p)
{ double *q,s; int m;
for (q=(double*)(p+1), m=*p, s=0.; m>0; m--) s+= *q++;
return s;}
//----- 5
char *F5(char *s, char *p[])
{ int i,j;
for (i=0; p[i]!=NULL; i++) {
    for (j=0; p[i][j]!='\0'; j++) *s++ = p[i][j];
    *s++ = '\0';
}
*s = '\0'; return s;}
//----- 6
union x {int *pi; long *pl; double *pd;};
double F6(int *p)
{ union x ptr;
double dd=0;
for (ptr.pi=p; *ptr.pi !=0; )
    switch (*ptr.pi++) {
case 1: dd += *ptr.pi++; break;
case 2: dd += *ptr.pl++; break;
case 3: dd += *ptr.pd++; break;
}
return dd;}
//----- 7
unsigned char *F7(unsigned char *s, char *p)
{ int n;
for (n=0; p[n] != '\0'; n++);
*((int*)s) = n; s+=sizeof(int);
for (; *p != '\0'; *s++ = *p++);
return s;}
//----- 8
int *F8(int *p, int n, double v[])
{ *p++ = n;
for (int i=0; i<n; i++)
    { *((double*)p) = v[i]; p+=sizeof(double)/sizeof(int); }
return p;}
//----- 9
double F9(int *p)
{ double s=0;
while(*p!=0) {
    if (*p>0) s+=*p++;

```

```

else
{ p++; s+= *((double*)p); p+=sizeof(double)/sizeof(int);}
} return s; }
//----- 10
double F10(char *p)
{ double s=0; char *q;
for (q=p; *q!=0; q++);
for (q++; *p!=0; p++)
switch(*p) {
case 'd': s+=*((int*)q); q+=sizeof(int); break;
case 'f': s+=*((double*)q); q+=sizeof(double); break;
case 'l': s+=*((long*)q); q+=sizeof(long); break; }
return s; }
//----- 11
int F11(char *p)
{ int s=0, *v; char *q;
for (q=p; *q!=0; q++);
q++; v=(int*)q;
for(; *p!=0; p++)
if (*p>='0' && *p<='9') s+=v[*p-'0'];
return s; }

```

Определите формат последовательности параметров функции и напишите ее вызов с фактическими параметрами – константами.

Пример оформления тестового задания

```

//-----31-08.cpp
double F(int a1,...) // Первый параметр - счетчик цикла
{ int i,n;
double s,*q=(double*)&a1+1); // Указатель на второй и последующие
for (s=0, n=a1; n!=0; n--) // параметры - типа double*
s += *q++; // Сумма параметров, начиная
return s; // со второго
void main() { printf("%lf\n",F(3,1.5,2.5,3.5)); }

```

Указатель **q** типа **double*** ссылается на второй параметр функции (первый из переменного списка) – **&a1+1** – указатель на область памяти, «следующую за...». Первый параметр используется в качестве счетчика повторений цикла, цикл суммирует значения, последовательно извлекаемые из-под указателя **q**. Результат – функция суммирует вещественные переменные из списка, предваренного целым счетчиком.

```

//-----31-09.cpp
//----- 1
void F1(int *p,...)
{ int **q, i, d;
for (i=1, q = &p, d=*p; q[i]!=NULL; i++) *q[i-1] = *q[i];
*q[i-1] = d;}
//----- 2
int *F2(int *p,...)
{ int **q, i, *s;
for (i=1, q = &p, s=p; q[i]!=NULL; i++)
if (*q[i] > *s) s = q[i];
}

```

```

return s; }
//-----3
int F3(int p[], int a1,...)
{ int *q, i;
for (i=0, q = &a1; q[i] > 0; i++) p[i] = q[i];
return i;}
//-----4
union x { int *pi; long *pl; double *pd; };
void F4(int p,...)
{ union x ptr;
for (ptr.pi = &p; *ptr.pi != 0; ) {
    switch(*ptr.pi++) {
        case 1: printf("%d", *ptr.pi++); break;
        case 2: printf("%ld", *ptr.pl++); break;
        case 3: printf("%lf", *ptr.pd++); break;
    }}
//-----5
char **F5(char *p,...)
{ char **q,**s; int i,n;
for (n=0, q = &p; q[n] !=NULL; n++);
s = new char*[n+1];
for (i=0, q = &p; q[i] !=NULL; i++) s[i]=q[i];
s[n]=NULL; return s;}
//-----6
char *F6(char *p,...)
{ char **q; int i,n;
for (i=0, n=0, q = &p; q[i] !=NULL; i++)
    if (strlen(q[i]) > strlen(q[n])) n=i;
return q[n]; }
//-----7
int F7(int a1,...)
{ int *q, s;
for (s=0, q = &a1; *q > 0; q++) s+= *q;
return s;}
//-----8
union xx { int *pi; long *pl; double *pd; };
double F8(int p,...)
{ union xx ptr;
double dd=0;
for (ptr.pi = &p; *ptr.pi != 0; )
{
    switch(*ptr.pi++) {
        case 1: dd+= *ptr.pi++; break;
        case 2: dd+= *ptr.pl++; break;
        case 3: dd+= *ptr.pd++; break;
    }}
return dd;}
//-----9
double F9(int a1,...)
{ double s=0; int *p=&a1;
while(*p!=0) {
    if (*p>0) s+=*p++;
    else
        { p++; s += *((double*)p); p+=sizeof(double)/sizeof(int); }
}
return s; }

```

```

//-----10
double F10(char *p,...)
{ double s;
  int *q=(int *)&p+1;
  for (s=0;*p!=0; p++)
    switch(*p) {
      case 'd': s+=*q++; break;
      case 'f': s+=*((double*)q)++; break;
      case 'l': s+=*((long*)q)++; break;
    } return s; }
//-----11
int F11(char *p,...)
{ int s=0, *q=(int *)&p+1;
  for(;*p!=0;p++)
    if (*p>='0' && *p<='9') s+=q[*p-'0'];
  return s; }
//-----12
double F12(int p,...)
{ double dd=0; int *q=&p;
  for (; *q != 0; ) {
    switch(*q++)
    {
      case 1: dd+= *q++; break;
      case 2: dd+= *((long*)q)++; break;
      case 3: dd+= *((double*)q)++; break;
    }
  }
  return dd; }

```

3.2. ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ И МАССИВЫ

Статический и динамический. Терминология – статический/динамический характеризует изменение свойств объекта во время работы программы. Если эти свойства не меняются (жестко задаются при трансляции), то они статические, если меняются – динамические. То же касается и существования самих объектов.

Статический – объект, создаваемый при трансляции, **динамический** – при выполнении программы.

По отношению к переменным это выглядит так: если переменная создается при трансляции (а с созданием переменной прежде всего связано распределение памяти под нее), то ее можно назвать статической, если же создается во время работы программы, – то динамической. С этой точки зрения все обычные (именованные) переменные являются статическими.

Основной недостаток обычных переменных – это их фиксированная размерность, которая определяется при трансляции (операция **sizeof** возвращает для них константу). Количество переменных в программе также ограничено (за исключением случая рекурсив-

ного вызова функции). Но при написании многих программ заранее неизвестна размерность обрабатываемых данных. При использовании обычных переменных в таких случаях возможен единственный выход – определять размерность «по максимуму». В ситуации, когда требуется обработать данные еще большей размерности, необходимо внести изменения в текст программы и перетранслировать ее. Для таких целей используется команда препроцессора **#define** с тем, чтобы не менять значение одной и той же константы в нескольких местах программы.

```
#define SZ 1000
int A[SZ];
struct xxx { int a; double b; } B[SZ];
for (i=0; i < SZ; i++) B[i].a = A[i];
```

Динамические переменные. На уровне библиотек в Си имеется механизм создания и уничтожения переменных работающей программой. Такие переменные называются **динамическими**,

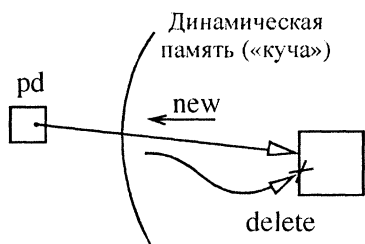


Рис. 3.4

а область памяти, в которой они создаются – **динамической памятью**, или **«кучей»** (рис. 3.4). «Куча» представляет собой дополнительную область памяти по отношению к той, которую занимает программа в момент загрузки – сегменты команд, глобальных (статических) данных и локальных переменных (стека). Основные свойства динамических переменных:

- динамические переменные создаются и уничтожаются работающей программой путем выполнения специальных операторов или вызовов функций;
- количество и размерность динамических переменных могут меняться в процессе работы программы и зависят от количества вызовов соответствующих функций и передаваемых при вызове параметров;
- динамическая переменная не имеет имени, доступ к ней возможен только через указатель;
- при выполнении функции создания динамической переменной в «куче» выделяется свободная память необходимого размера и возвращается указатель на нее (адрес);
- функция уничтожения динамической переменной получает указатель на уничтожаемую переменную.

Самые важные свойства динамических переменных – это их «безымянность» и доступность по указателю, чем и определяется возможность варьировать число таких переменных в программе. Из этого можно сделать следующие выводы:

- если динамическая переменная создана, а указатель на нее «потерян» программой, то такая переменная представляет собой «вещь в себе» – существует, но недоступна для использования;

- динамическая переменная может, в свою очередь, содержать один или несколько указателей на другие динамические переменные. В этом случае мы получаем динамические структуры данных, в которых количество переменных и связи между ними могут меняться в процессе работы программы (списки, деревья, виртуальные массивы);

- управление динамической памятью построено обычно таким образом, что ответственность за корректное использование указателей на динамические переменные несет программа (точнее, программист, написавший ее). Ошибки в процессе создания, уничтожения и работы с динамическими переменными (повторная попытка уничтожения динамической переменной, попытка уничтожения переменной, не являющейся динамической, и т.д.), трудно обнаруживаются и приводят к непредсказуемым последствиям в работе программы.

Операторы управления динамической памятью. Операторы **new** и **delete** используют при работе обозначения **абстрактных типов данных** для создаваемых переменных:

- при создании динамической переменной в операторе **new** указывается ее тип, сам оператор имеет тип результата – указатель на создаваемый тип, а значение – адрес созданной переменной или массива;

- если выделяется память под массив динамических переменных, то в операторе **new** добавляются квадратные скобки;

- оператор **delete** получает указатель на уничтожаемую переменную или массив.

```
double *pd;
pd = new double;           // Обычная динамическая переменная
if (pd != NULL){
    *pd = 5;
    delete pd;}
double *pdm;               // Массив динамических переменных
pdm = new double[20];
if (pdm != NULL){
    for (i=0; i<20; i++) pdm[i]=0;
    delete pd; }
```


Функции управления динамической памятью низкого уровня. Работать с памятью на Си можно и на «низком» уровне, то есть рассматривая переменные просто как области памяти известной размерности, используя операции **sizeof** для получения размерности переменных и преобразование типа указателя для изменения «точки зрения» на содержимое памяти (см. раздел 3.1). Функции распределения памяти низкого уровня «не вникают» в содержание создаваемых переменных, единственно важным для них является их размерность, выраженная естественным для Си способом в байтах (при помощи операции **sizeof**). Адрес выделенной области памяти также возвращается в виде указателя типа **void*** – абстрактный адрес памяти без определения адресуемого типа данных.

```
void    *malloc(int size); // Выделить область памяти размером
                                // в size байтов и вернуть адрес
void    free(void *p);    // Освободить область памяти,
                                // выделенную по адресу p
void    *realloc(void *p, int size); // Расширить выделенную область памяти
                                // до размера size, при изменении адреса
                                // переписать старое содержимое блока
#include <alloc.h>         // Библиотека функций управления памятью
double *pd;               // Обычная динамическая переменная
pd = (double*)malloc(sizeof(double));
if (pd != NULL){
    *pd = 5;
    free((double*)pd); }
double *pdm;              // Массив динамических переменных
pdm = (double*)malloc(sizeof(double)*20);
if (pdm != NULL){
    for (i=0; i<20; i++) pdm[i]=0;
    free((void*)pdm); }
```

Заметим, что оператор **delete**, функции **free** и **realloc** не содержат размерности возвращаемой области памяти. Очевидно, что библиотека, управляющая динамической памятью, должна сохранять информацию о размерности выделенных блоков.

Динамические массивы. Поскольку любой указатель в Си по определению адресует массив элементов указуемого типа неограниченной размерности, то функция **malloc** и оператор **new** могут использоваться для создания не только отдельных переменных, но и их массивов. Тот же самый указатель, который запоминал адрес отдельной динамической переменной, будет работать теперь с массивом. Размерность его задается значением в квадратных скобках оператора **new**. В функции **malloc** объем требуемой памяти указывается как произведение размерности элементов на их коли-

чество. Это происходит во время работы программы, и, следовательно, размерность массива может меняться от одного выполнения программы к другому.

Массивы, создаваемые в динамической памяти, называются **динамическими**. Свойства указателей позволяют одинаковым образом обращаться как с динамическими, так и с обычными массивами. Во многих языках интерпретирующего типа (например, Бейсик) подобный механизм скрыт в самом трансляторе, поэтому массивы там «по своей природе» могут быть переменной размерности, определяемой во время работы программы.

Динамические массивы и проблемы размерности данных. Как известно, любого ресурса всегда не хватает. В компьютерах это прежде всего относится к памяти. Если на проблему ее распределения посмотреть с обычных житейских позиций, то можно извлечь много полезного для понимания принципов статического и динамического распределения памяти. Пусть наша программа обрабатывает данные от нескольких источников, причем объемы их заранее неизвестны. Рассмотрим, как можно поступить в таком случае:

- самый неэффективный вариант: под каждый вид данных резервировать память заранее «по максимуму». Применительно к массиву это означает, что мы заранее выбираем такую размерность, которая никогда не будет превышена. Но, тем не менее, такое «никогда» рано или поздно может случиться, поэтому процесс заполнения массива лучше контролировать;

- приемлемый вариант может быть реализован, если в какой-то момент времени выполнения программа «узнает», какова в этот раз будет размерность обрабатываемых данных. Тогда она может создать динамический массив такой размерности и работать с ним. К сожалению, подобное «знание» не всегда возможно;

- идеальный вариант заключается в создании такой структуры данных, которая автоматически увеличивает свою размерность при ее заполнении. К сожалению, в случае с массивом ни язык, ни библиотека здесь не помогут – его можно реализовать только программно, по справедливости назвав **виртуальным**.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Динамический массив заданной размерности. Простейший случай, когда программа непосредственно получает требуемую размерность динамического массива и создает его.

```
//-----32-00.cpp
//---- Динамический массив predetermined размерности
int *GetArray(){
int N,i; // Размерность массива
int *p; // Указатель на массив
printf("Элементов в массиве:"); // в динамической памяти
scanf("%d",&N);
if ((p = new int[N + 1]) == NULL)
return NULL; // или malloc((N+1)*sizeof(double))
for (i=0; i<N; i++) {
printf("%d-ый элемент:",i);
scanf("%d",&p[i]); }
p[i] = 0; // В конце последовательности - 0
return(p); } // Вернуть указатель
```

Динамический массив – предварительное определение размерности. Если программа заранее не знает размерности массива, она может попытаться ее вычислить. Иногда это требует «двойной работы»: необходимо сначала выполнить алгоритм генерации данных (или его часть), чтобы определить (или оценить) размерность полученных данных, а потом повторить его с уже имеющейся динамической памятью. Например, чтобы возратить в динамическом массиве разложение заданного числа на простые множители, необходимо сначала провести это разложение с целью определения их количества, а затем повторить – для заполнения динамического массива.

```
//-----32-01.cpp
//----- Динамический массив простых множителей числа
int *mnoг(long vv){
long nn=vv;
for (int sz=1; vv!=1; sz++){ // Цикл определения количества
for (int i=2; vv%i!=0; i++); // Определить очередной множитель
vv = vv / i; }
int *p=new int[sz];
for (int k=0; nn!=1; k++){ // Повторный цикл заполнения
for (int i=2; nn%i!=0; i++); // Определить очередной множитель
p[k]=i; // Сохранить множитель в массиве
nn = nn / i; }
p[k]=0; return p; } // Вернуть указатель на дин. массив
```

Строка – динамический массив. Наиболее показательное применение динамических массивов символов при работе со строками. В идеальном случае при любой операции над строкой создается динамический массив символов, размерность которого равна длине строки.

```
//-----32-02.cpp
// Объединить две строки в одну в динамическом массиве
char *TwoToOne(char *p1, char *p2){
char *out; int n1,n2;
for (n1=0; p1[n1]!='\0'; n1++); // Длина первой строки
for (n2=0; p2[n2]!='\0'; n2++); // Длина второй строки
```

```

if ((out = new char [n1+n2+1]) == NULL)
return NULL; // Выделить память под результат
for (n1=0; *p1!='\0';) out[n1++] = *p1++;
while(*p2!=0) out[n1++] = *p2++; // Копировать строки
out[n1] = '\0';
return out; } // Вернуть указатель на динам. массив

```

Динамический массив – изменение размерности при переполнении. Для снятия ограничений на размерность массива необходимо отслеживать процесс заполнения динамического массива и при переполнении – перераспределять память: выделять память большего объема, переписывать туда содержимое старой и освобождать старую. Эффективно делать это периодически, изменяя размерность массива кратно (линейно) или по степеням (экспоненциально).

```

//-----32-03.cpp
//---- Создание динамического массива произвольной размерности
// Размерность массива меняется при заполнении кратно N - N, 2N, 3N ...
#define N 5
int *GetArray(){
int i , *p; // Указатель на массив
p = new int[N]; // Массив начальной размерности
for (i=0; i<N; i++) {
printf("%d-ый элемент:",i);
scanf("%d",&p[i]);
if ((i+1)%N==0){ // Массив заполнен ???
int *q=new int[i+1+N]; // Создать новый и переписать
for (int j=0; j<=i; j++)
q[j]=p[j];
delete p; // Старый уничтожить
p=q; // Считать новый за старый
}
if (p[i]==0) return p; // Ограничитель ввода - 0
} }

```

Более изящно это перераспределение можно сделать с помощью функции низкого уровня **realloc**, которая резервирует память новой размерности и переписывает в нее содержимое старой области памяти (либо расширяет существующую):

```

p = (int*) realloc((void*)p,sizeof(int)*(i+1+N));

```

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определите содержательный смысл функции, назначение и способ формирования динамического массива.

```

//-----32-04.cpp
//----- 1
char *F1(char *s)
{ char *p,*q; int n;
for (n=0; s[n] !='\0'; n++);

```

```

p = new char[n+1];
for (q=p; n >=0; n--) *q++ = *s++;
return p; }
//----- 2
int *F2()
{ int n,i,*p;
scanf("%d",&n);
p=new int[n+1];
for (p[0]=n, i=0; i<n; i++) scanf("%d",&p[i+1]);
return p; }
//----- 3
int *F3()
{ int n,i,*p;
scanf("%d",&n);
p=new int[n+1];
for (i=0; i<n; i++)
    { scanf("%d",&p[i]); if (p[i]<0) break; }
p[i]=-1;
return p; }
//----- 4
char *F4(char *p, char *q)
{ int n1, n2;
for (n1=0; p[n1]!=0; n1++);
for (n2=0; q[n2]!=0; n2++);
char *s,*v;
s=v=new char[n1+n2+1];
while(*p!=0) *s++ = *p++;
while(*q!=0) *s++ = *q++;
*s=0;
return v; }
//----- 5
double *F5(int n, double v[]){
double *p=new double[n+1];
p[0]=n;
for (int i=0; i<n; i++) p[i+1]=v[i];
return p; }
//----- 6
int *F6()
{ int *p, n=10,i;
p=new int[n];
for (i=0;;i++){
    if (i==n) { n=n*2; p=(int*)realloc(p,sizeof(int)*n); }
    scanf("%d",&p[i]);
    if (p[i]==0) break;}
return p;}
//----- 7
void *F7(void *p, int n)
{ char *pp, *qq, *ss;
qq = ss = new char [n];
for (pp= (char*)p; n!=0; n--) *pp++ = *qq++;
return ss;}
//----- 8
int *F8(int n)
{ int s,i,m,k,*p;
s = 10; p = new int[s];
for (i=2, m=0; i<n; i++) {
    for (k=0; k<m; k++)
        if (i % p[k]==0) break;

```

```

if (k==m)
    { p[m++] = i;
      if (m==s){
        s=s*2;
        p= (int*) realloc( (void*) p,sizeof(int)*s);
      }}
return p; }

```

3.3. ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЕ

Динамическое связывание. Компилятор превращает вызов функции в команду процессора, в которой присутствует адрес этой функции. Если же функция внешняя, то это же самое делает компоновщик на этапе сборки программы. Это называется **статическим связыванием** в том смысле, что в момент загрузки программы все связи между вызовами функций и самими функциями установлены. **Динамическим связыванием** называется связывание вызова внешней функции с ее адресом во время работы программы. Соответствующие средства имеются обычно на системном уровне (например, **DLL – dynamic linking library**, динамически связываемые библиотеки). На уровне языка программирования они довольно редки (например, процедурный тип в Паскале). Си позволяет работать с архитектурной первоосновой динамического связывания – указателем на функцию.

Указатель на функцию – переменная, которая содержит адрес некоторой функции. Соответственно, косвенное обращение по этому указателю представляет собой вызов функции.

Определение указателя на функцию имеет вид:

```

int  (*pf)();           // Без контроля параметров вызова
int  (*pf)(void);       // Без параметров, с контролем по прототипу
int  (*pf)(int, char*); // С контролем по прототипу

```

В соответствии с принципом контекстного определения типа данных эту конструкцию следует понимать так: **pf** – переменная, при косвенном обращении к которой получается функция с соответствующим прототипом, например, **int f(int, char*)**, то есть **pf** содержит адрес функции или указатель на функцию. Следует обратить внимание на то, что в определении указателя присутствует прототип – указатель ссылается не на произвольную функцию, а только на одну из функций с заданной схемой формальных параметров и результата.

Перед началом работы с указателем его необходимо назначить на соответствующий объект, в данном случае – на функцию. В синтаксисе Си выражение вида **&имя_функции** имеет смысл –

начальный адрес функции или указатель на функцию. Кроме того, по аналогии с именем массива использование имени функции без скобок интерпретируется как указатель на эту функцию. Указатель может быть инициализирован и при определении. Возможны следующие способы назначения указателей:

```
int    INC(int a) { return a+1; }
extern    int DEC(int);
int    (*pf)(int);
pf = &INC;
pf = INC;                                // Присваивание указателя
int    (*pp)(int) = &DEC;                // Инициализация указателя
```

Естественно, что функция, на которую формируется указатель, должна быть известна транслятору – определена или объявлена как внешняя. Синтаксис вызова функции по указателю совпадает с синтаксисом ее определения.

```
n = (*pf)(1) + (*pp)(n);                // Эквивалентно
n = INC(1) + DEC(n);
```

Указатель на функцию как средство параметризации алгоритма. Оригинальность и обособленность такого типа данных заключается в том, что указуемым объектом является не переменная (компонент данных программы), а функция (компонент алгоритма). Но сущность указателя при этом не меняется: если обычный указатель позволяет параметризовать алгоритм обработки данных, то указатель на функцию позволяет параметризовать сам алгоритм. То есть некоторая его часть может быть заранее неизвестна (не определена, произвольна) и будет подключаться к основному алгоритму только в момент его выполнения (динамическое связывание) (рис. 3.5).

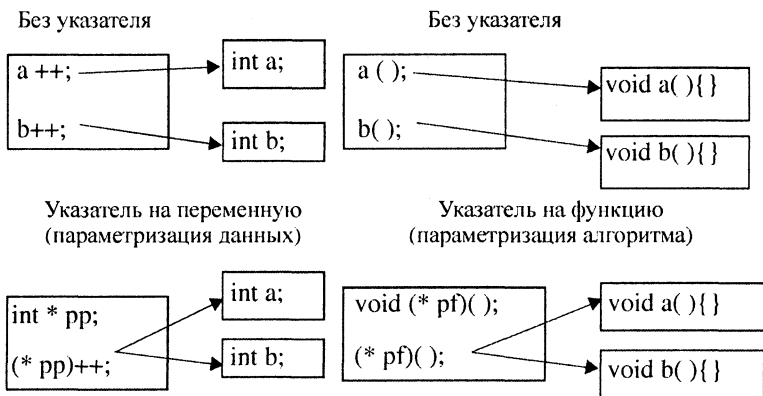


Рис. 3.5

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Вызов функции по имени. Программа-интерпретатор должна вызывать заданную функцию, получив ее имя. В принципе, это можно сделать с помощью обычного переключателя (**switch**), добавляя для каждой новой функции новое ветвление. Программу можно сделать более регулярной и «изолировать» от данных, если использовать наряду с массивом имен массив указателей на функции.

```
extern double sin(double);
extern double cos(double);
extern double tan(double);
char *names[] =
    { "sin", "cos", "tan", NULL};    // Массив имен (указатели на строки)
double (*pf[])(double) =
    { sin, cos, tan};              // Массив функций (адреса функций)
```

Массив указателей на функции **pf** инициализирован адресами библиотечных функций **sin**, **cos** и **tan**. Обратите внимание на контекст определения типа переменной, заданный последовательностью операций, – массив, указатель, функция с прототипом **double f(double)**.

```
//-----33-01.cpp
//---- Вызов функции по имени из заданного списка
double call_by_name(char *pn, double arg) {
    for ( int i=0; names[i]!=NULL; i++)
        if (strcmp(names[i],pn) == 0) {    // Имя найдено -
            return ((*pf[i])(arg));        // вызов функции по i-му
        }                                  // указателю в массиве pf
    return 0.;}
```

Указатель на функцию как формальный параметр. Это типичный случай реализации алгоритма, в котором некоторый внутренний шаг задан в виде действия общего вида. Оно осуществляется получением указателя на необходимую функцию и обращением к ней через этот указатель. Пример: функция вычисления определенного интеграла для произвольной подынтегральной функции.

```
//-----33-02.cpp
//-----Численное интегрирование произвольной функции
double INTEG(double a, double b, int n, double(*pf)(double))
// a,b - границы интегрирования, n - число точек
// pf - подынтегральная функция
{ double s,h,x;
  for (s=0., x=a, h = (b-a)/n; x <=b; x+=h) s += (*pf)(x) * h;
  return s; }
extern double sin(double);
void main() { printf("sin(0..pi/2)=%lf\n",INTEG(0.,3.1415926/2,40,sin)); }
```

Итератор. Рассмотрим случай, когда структура данных (массив указателей, список, дерево) включает в себя переменные одно-

го типа, но сам тип может меняться в каждом конкретном экземпляре структуры данных (например, массивы указателей на **int**, **double**, **struct man**). Очевидно, что алгоритмы поиска, сортировки, включения, исключения и других действий будут совпадать с точностью до операции над этими переменными. Например, для сортировки массивов указателей на целые переменные и строки могут использоваться идентичные алгоритмы, различающиеся только операцией сравнения двух переменных соответствующих типов (операция «>» для целых и функция **strcmp** для строк). Если эту операцию вынести за пределы алгоритма, реализовать отдельной функцией, а указатель на нее передавать в качестве параметра, то мы получим универсальную функцию сортировки массивов указателей на переменные любого типа данных, то есть **итератор**.

Типичными итераторами являются:

- итератор обхода (**foreach**), выполняющий для каждой переменной в структуре данных указанную функцию;
- итератор поиска (**firstthat**), выполняющий для каждой переменной в структуре данных функцию проверки и возвращающий указатель на первую переменную, которая удовлетворяет условию, проверяемому в функции;
- итераторы сортировки, поиска минимального, двоичного поиска, включения и исключения элементов в упорядоченную структуру данных, основанные на операции сравнения.

```
//-----33-03.cpp
//----- Итераторы foreach, firstthat и поиска минимального для списка
struct list { list *next;           // Указатель на следующий
void *pdata; };                   // Указатель на данные
//----- Итератор: для каждого элемента списка
void ForEach(list *pv, void (*pf)(void*) ) {
for ( ; pv !=NULL; pv = pv->next)
    (*pf)(pv->pdata);
}
//----- Итератор: поиск первого в списке по условию
void *FirstThat(list *pv, int (*pf)(void*)) {
for ( ; pv !=NULL; pv = pv->next)
    if ((*pf)(pv->pdata)) return pv->pdata;
return NULL; }
//----- Итератор: поиск минимального в списке
void *FindMin(list *pv, int (*pf)(void*,void*))
{ list *pmin;
for ( pmin=pv; pv !=NULL; pv = pv->next)
    if ((*pf)(pv->pdata ,pmin->pdata) <0) pmin=pv;
return pmin; }
//----- Примеры использования итератора -----
//----- Функция вывода строки
void print(void *p) { puts((char*)p); }
//----- Функция проверки : длины строки >5
int bigstr(void *p) { return strlen((char*)p) > 5; }
```

```

//----- Функция сравнения строк по длине
int scmp(void *p1, void *p2)
{ return strlen((char*)p1)- strlen((char*)p2); }
//----- Вызов итераторов для статического списка,
// содержащего указатели на строки
list a1={NULL,"aaaa"}, a2={&a1,"bbbbbb"}, a3={&a2,"cccc"},
*PH=&a3;

//----- Итератор сортировки для массива указателей
void Sort(void **pp, int (*pf)(void*,void*))
{ int i,k;
do
for (k=0,i=1; pp[i] !=NULL; i++)
if ( (*pf)(pp[i-1],pp[i])>=0) // вызов функции сравнения
{ void *q; // перестановка указателей
k++; q = pp[i-1]; pp[i-1] = pp[i]; pp[i] = q;
}
while(k); }
// Пример вызова итератора сортировки для массива
// указателей на целые переменные
int cmp_int(void *p1, void *p2)
{ return *(int*)p1-*(int*)p2; }
int b1=5, b2=6, b3=3, b4=2;
void *PP[] = {&b1, &b2, &b3, &b4, NULL};
void main()
{ char *pp;
ForEach(PH,print);
pp = (char*) FirstThat(PH,bigstr);
if (pp !=NULL) puts(pp);
pp = (char*) FindMin(PH,scmp);
if (pp !=NULL) puts(pp);
Sort(PP,cmp_int);
for (int i=0; PP[i]!=NULL;i++) printf("%d ",*(int*)PP[i]);
puts("");}

```

Из приведенных примеров просматривается общая схема итератора (рис. 3.6):

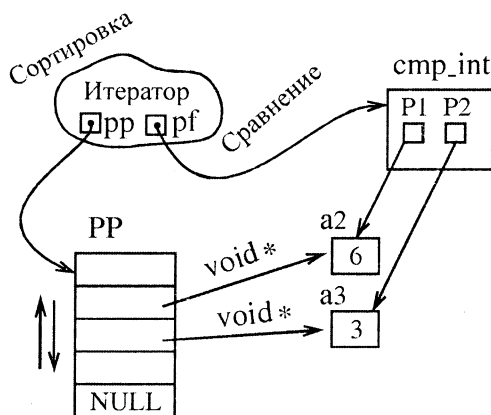


Рис. 3.6

– структура данных, обрабатываемая итератором, содержит в своих элементах указатели на переменные произвольного (неизвестного для итератора) типа **void***, но одинакового в каждом экземпляре структуры данных;

– итератор получает в качестве параметров указатель на структуру данных и указатель на функцию обработки входящих в структуру данных переменных;

– итератор выполняет алгоритм обработки структуры данных в соответствии со своим назначением: **foreach** обходит все переменные, **firstthat** обходит и проверяет все переменные, итератор сортировки сортирует указатели на хранимые объекты (или соответствующие элементы структуры данных, например, элементы списка);

– действие, которое надлежит выполнить над хранимыми объектами произвольного типа (например, сравнение), определяется внешней функцией, передаваемой в итератор как формальный параметр-указатель. Итераторы **foreach** и **firstthat** вызывают функцию, переданную по указателю с параметром – указателем на переменную, которую нужно обработать или проверить. Итераторы сортировки, ускоренного поиска и другие вызывают функцию по указателю для сравнения двух переменных, указатели на которые берутся из структуры данных и становятся параметрами функции сравнения.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Для заданной в варианте структуры данных, каждый элемент которой содержит указатели на элементы произвольного типа **void***, написать итератор. Проверить его работу на примере вызова итератора для структуры данных с соответствующими элементами и конкретной функцией.

1. Односвязный список, элемент которого содержит указатель типа **void*** на элемент данных. Функция включения в конец списка и итератор сортировки методом вставок: исключается первый элемент и включается в новый список с порядке возрастания. Проверить на примере элементов данных – строк и функции сравнения **strcmp**.

2. Дерево, каждая вершина которого содержит указатель на элемент данных **void*** и не более четырех указателей на поддеревья. Итератор поиска первого подходящего **firstthat** и функция включения в поддерево с минимальной длиной ветви. Проверить

на примере элементов данных – строк и функции проверки на длину строки – не менее 10 символов.

3. Динамический массив указателей типа **void***, содержащий указатели на упорядоченные элементы данных. Итераторы включения с сохранением упорядоченности и **foreach**. Предусмотреть увеличение размерности динамического массива при включении данных. Проверить на примерах элементов данных типов **int** и **float** (две проверки).

4. Двусвязный циклический список, элемент которого содержит указатель типа **void*** на элемент данных. Итераторы **foreach** и включения с сохранением упорядоченности. Проверить на примере элементов данных структурированного типа, содержащих фамилию, год рождения и номер группы, с использованием функций сравнения по году рождения и по фамилии.

5. Двоичное дерево, каждая вершина которого содержит указатель типа **void***. Итераторы **foreach**, двоичного поиска и включения с сохранением упорядоченности. Проверить на примере элементов данных структурированного типа, содержащих фамилию, год рождения и номер группы, с использованием функций сравнения по году рождения и по фамилии.

6. Динамический массив указателей типа **void*** на неупорядоченные элементы данных. Итератор поиска минимального элемента. Проверить на примере элементов данных структурированного типа, содержащих фамилию, год рождения и номер группы, с использованием функций сравнения по году рождения и по фамилии.

7. Динамический массив указателей типа **void***, содержащий указатели на элементы данных. Функция включения элемента последним, итераторы сортировки и **foreach**. Предусмотреть увеличение размерности динамического массива при включении данных. Проверить на примерах элементов данных типов **int** и **float** (две проверки).

8. Двусвязный циклический список, элемент которого содержит указатель типа **void*** на элемент данных. Функция включения элемента первым, итераторы **foreach** и сортировки выбором (ищется максимальный элемент и включается в начало нового списка). Проверить на примере элементов данных структурированного типа, содержащих фамилию, год рождения и номер группы, с использованием функций сравнения по году рождения и по фамилии.

9. Односвязный список, элемент которого содержит указатель типа **void*** на элемент данных. Функция включения элемента пер-

вым, итераторы **foreach**, поиска минимального и сортировки выбором: выбирается максимальный элемент и вставляется первым в новый список. Проверить на примере элементов данных – строк и функции сравнения **strcmp**.

10. Дерево, каждая вершина которого содержит указатель на элемент данных **void*** и не более четырех указателей на поддеревья. Итератор поиска минимального элемента и функция включения в поддерево с минимальным количеством вершин. Проверить на примере элементов данных – строк и функции сравнения двух строк по длине.

11. Односвязный список, элемент которого содержит указатель типа **void*** на упорядоченные элементы данных. Итераторы включения с сохранением упорядоченности и **foreach**. Проверить на примере элементов данных – строк и функции сравнения **strcmp**.

12. Двусвязный циклический список, элемент которого содержит указатель типа **void*** на элемент данных. Функция включения элемента последним, итераторы **foreach** и сортировки вставками (выбирается первый элемент и включается в новый список с сохранением упорядоченности). Проверить на примере элементов данных типов **int** и **float** (две проверки).

13. Дерево, каждая вершина которого содержит указатель на элемент данных **void*** и не более четырех указателей на поддеревья. Итератор поиска минимального элемента и функция включения в поддерево с минимальной длиной ветви. Проверить на примерах элементов данных типов **int** и **float** (две проверки).

14. Дерево, каждая вершина которого содержит указатель на элемент данных **void*** и не более четырех указателей на поддеревья. Итераторы **foteach** (с выводом уровня вложенности) и включения нового элемента таким образом, чтобы меньшие элементы были ближе к корню дерева. Проверить на примерах элементов данных типов **int** и **float** (две проверки).

15. Двоичное дерево, каждая вершина которого содержит указатель типа **void***. Итератор **foreach**, включения с сохранением упорядоченности и функция получения указателя на элемент по его логическому номеру в порядке возрастания. Проверить на примерах элементов данных типов **int** и **float** (две проверки).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Тестовые задания содержат итерационные циклы, используемые в приближенных вычислениях. Содержательно определить смысл алгоритма и назначение указателя на функцию.

```
//-----33-04.cpp
//----- 1
double F1(double a, double b, double (*pf)(double))
{ double m;
if ((*pf)(a) * (*pf)(b) > 0) return(a);
while ( b-a > 0.0001 ) {
    m = (b + a)/2;
    if ((*pf)(a) * (*pf)(m) < 0) b=m; else a=m;
}
return a ;}
//----- 2
double F2(double x, double s0, double (*pf)(double,int))
{ double s; int n;
for (n=1, s=0.0; fabs(s0) > 0.0001; n++)
    { s += s0; s0 = s0 * (pf)(x,n); }
return s; }
double ff(double x, int n) { return( x/n); }
void main1()
{ double x,y; y = F2(x,1,ff); }
//----- 3
double F(double a, double b, double (*pf)(double))
{ double dd;
for (dd = 0.0001; b-a > dd;)
    if ((*pf)(a) > (*pf)(b)) b -=dd; else a +=dd;
return a; }
//----- 4
double F4(double x, double (*pf)(double))
{ double x1;
do {
    x1 = x; x = (*pf)(x1) + x1;
    if (fabs(x) > fabs(x1)) return(0.0);
} while (fabs(x1-x) > 0.0001);
return x; }
//----- 5
double F5(double a, double b, int n, double(*pf)(double))
{ double s,h,x;
for (s=0., x=a, h = (b-a)/n; x <=b; x+=h)
    s += (*pf)(x) * h;
return s;}
extern double sin(double);
void main2() { printf("%lf\n",F5(0.,1.,40,sin)); }
//----- 6
double P6(double(*ff [])(double) ,int n , double x )
{ return (* ff [n])(x) ; }
double(*FF6[])(double)={sin,cos,tan};
void main3(){ printf("%lf\n",P6( FF6,1,0.5 ));}
//----- 7
typedef double(*PF)(double) ;
double P7( PF ff [],int n , double x )
{ return (* ff [n])(x) ; }
PF FF7[]={sin,cos,tan};
void main4(){ printf("%lf\n",P7( FF7,2,1.5 ));}
```

ГОЛОВОЛОМКИ, ЗАГАДКИ

Результат здесь очевиден. Будет выведена строка «I'm foo» или значение 6. Значительно труднее объяснить:

– что такое **P** – функция, указатель на функцию? Если функция, то где у нее определения формальных параметров и результата и что она делает? Если указатель, то где обращение по нему?

– где находится операция, по которой на самом деле производится вызов функций **foo** и **inc**.

Рекомендуется оттранслировать фрагмент через Ассемблер и проанализировать код. Он будет не в пример проще.

```
//-----33-05.cpp
```

```
//----- 1
```

```
void ( *P1(void(*ff)(void)))(void) { return ff; }
```

```
void foo1(void){ printf("I'm foo\n"); }
```

```
void main1(){(*P1(foo1))();}
```

```
//----- 2
```

```
int ( *P2(int(*ff)(int))(int) { return ff; }
```

```
int inc2(int n){ return n+1; }
```

```
void main2(){ printf("%d\n",(*P2(inc2))( 5 ));}
```

```
//----- 3
```

```
typedef void (*PF3)(void);
```

```
PF3 P3(PF3 ff) { return ff; }
```

```
void foo3(void){ printf("I'm foo\n"); }
```

```
void main3(){(*P3(foo3))();}
```

```
//----- 4
```

```
typedef int (*PF4)(int);
```

```
PF4 P4(PF4 ff) { return ff; }
```

```
int inc4(int n){ return n+1; }
```

```
void main4(){ printf("%d\n",(*P4(inc4))( 7 ));}
```

Определите смысл следующего архитектурно-зависимого фрагмента.

```
//-----33-06.cpp
```

```
void (*pf)(void) = (void*)(void)0x1000;
```

```
void main() { (*pf)(); }
```

3.4. РЕКУРСИЯ

Я оглянулся посмотреть,
не оглянулась ли она,
чтоб посмотреть, не оглянулся ли я...

М. Леонидов

О рекурсии несерьезно. Несерьезные, но хорошо иллюстрирующие принцип примеры можно найти в детских считалках и в клинической психиатрии:

У попа была собака, он ее любил.

Она съела кусок мяса, он ее убил.

Камнем придавил и на камне написал:

У попа была собака...

Я хочу Вам написать, что я хочу Вам написать, что я хочу Вам написать . . . (из письма пациента психиатру)

Главное подмечено верно: некоторое действие, включающее в себя такое же (или аналогичное) действие, имеет отношение не к логическому мышлению, а скорее к рефлексии – попытке думать о себе самом в третьем лице, что, как известно, в больших дозах до добра не доводит.

Рекурсия в природе, науке, программировании. Рекурсивным называется способ построения объекта (понятия, системы, описание действия), в котором определение объекта включает аналогичный объект (понятие, систему, действие) в виде некоторой его части. Общеизвестный пример рекурсивного изображения – предмет между двумя зеркалами: в каждом из них виден бесконечный ряд отражений. Более серьезные примеры рекурсии можно обнаружить в математике:

- рекуррентные соотношения определяют некоторый элемент последовательности через несколько предыдущих. Например, числа Фибоначчи: $F(n)=F(n-1)+F(n-2)$, где $F(0)=1$, $F(1)=1$. Если рассматривать этот ряд от младших членов к старшим, способ его построения задается циклическим алгоритмом, а если наоборот, от заданного $n=n_0$, то способ определения этого элемента через предыдущие будет рекурсивным.

В программировании таких примеров еще больше:

- рекурсивное определение в синтаксисе языка. Например, определение любого конкретного оператора (условный, цикл, блок) в качестве составных частей включает произвольный оператор;

- рекурсивная структура данных – элемент структуры данных содержит один или несколько указателей на аналогичную структуру данных. Например, односвязный список можно определить как элемент списка, содержащий указатель **NULL** или указатель на аналогичный список;

- рекурсивная функция – тело функции содержит прямой или косвенный (через другую функцию) собственный вызов.

Основные свойства рекурсии. Очевидно, что рекурсия не может быть безусловной, в этом случае она становится бесконечной. Это видно хотя бы из приведенной выше считалки. Рекурсия должна иметь внутри себя условие завершения, по которому очередной шаг ее уже не производится.

Другая, еще не отмеченная особенность: наряду с **линейной** рекурсией, когда определение объекта включает в себя единственный аналогичный объект, существует еще и **ветвящаяся** рекурсия, когда таких включаемых объектов много.

Особенности работы рекурсивной функции. Рекурсивные функции лишь на первый взгляд выглядят как обычные фрагменты программ. Чтобы ощутить их специфику, достаточно мысленно проследить по тексту программы процесс ее выполнения. В обычной программе мы будем следовать по цепочке вызовов функций, но ни разу повторно не войдем в один и тот же фрагмент, пока из него не выйдем. Можно сказать, что процесс выполнения программы «ложится» однозначно на текст программы. Другое дело – рекурсия. Если попытаться отследить по тексту программы процесс ее выполнения, то мы придем к такой ситуации: войдя в рекурсивную функцию **F**, мы «двигаемся» по ее тексту до тех пор, пока не встретим ее вызова, после чего мы опять начнем выполнять ту же самую функцию сначала. При этом следует отметить самое важное свойство рекурсивной функции – ее первый вызов еще не закончился. Чисто внешне создается впечатление, что текст функции воспроизводится (копируется) всякий раз, когда функция сама себя вызывает:

<code>void main()</code>	<code>void F()</code>	<code>void F()</code>	<code>void F()</code>
<code>{</code>	<code>{</code>	<code>{</code>	<code>{</code>
<code>F();</code>	<code>...if()F();</code>	<code>...if()F();</code>	<code>...if()F();</code>
<code>}</code>	<code>}</code>	<code>}</code>	<code>}</code>

На самом деле этот эффект воспроизводится в компьютере. Однако копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с локальными данными (формальные, фактические параметры, локальные переменные и точка возврата). Алгоритмическая часть (операторы, выражения) рекурсивной функции и глобальные переменные не меняются, поэтому они присутствуют в памяти компьютера в единственном экземпляре.

Рекурсивная функция и стек. Каждый рекурсивный вызов порождает новый «экземпляр» формальных параметров и локальных переменных, причем старый «экземпляр» не уничтожается, а сохраняется в стеке по принципу вложенности. Здесь имеет место единственный случай, когда одному имени переменной в процессе работы программы соответствует несколько ее экземпляров. Происходит это в такой последовательности:

- в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров. Обычно это производится в порядке, обратном их следованию в списке;
- при вызове функции в стек записывается точка возврата – адрес той части программы, где находится вызов функции;
- в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

Перечисленные переменные образуют группу (фрейм стека). Стек «помнит историю» рекурсивных вызовов в виде последовательности (цепочки) таких фреймов. Программа в каждый конкретный момент работает с последним вызовом и с последним фреймом. При завершении рекурсии программа возвращается к предыдущей версии рекурсивной функции и к предыдущему фрейму в стеке.

Рекурсивный алгоритм как процесс. Рекурсивный вызов, «экземпляр» рекурсивной функции является одним из идентичных повторяющихся шагов некоторого процесса, который в целом и решает поставленную задачу. В терминах процесса и его шагов основные параметры рекурсивной функции получают дополнительный смысл:

- формальные параметры рекурсивной функции представляют собой начальное состояние для текущего шага процесса;
- фактические параметры рекурсивного вызова представляют собой начальное состояние для следующего шага – перехода из текущего при рекурсивном вызове;
- автоматические переменные представляют собой внутренние характеристики процесса на текущем шаге его выполнения;
- внешние переменные представляют собой глобальное состояние всей системы, через которое отдельные шаги в последовательности могут взаимодействовать.

Это значит, что формальные параметры рекурсивной функции, глобальные и локальные переменные не могут быть взаимозаменяемы, как это иногда делается в обычных функциях.

Инварианты рекурсивного алгоритма. Специфика рекурсивных алгоритмов состоит в том, что они полностью исключают «исторический» подход к проектированию программы. Попытки логически проследить последовательность рекурсивных вызовов заранее обречены на провал. Их можно прокомментировать примерно такой фразой: «Функция **F** выполняет ... и вызывает **F**, которая выполняет ... и вызывает **F**...». Ясно, что для логического анализа программы в этом мало пользы.

Тем не менее, эта фраза смутно напоминает нам попытки «исторического» анализа циклических программ (см. раздел 1.7). Там для того чтобы понять, что делает цикл, предлагалось использовать некоторый **инвариант** (условие, соотношение), сохраняемый шагом цикла. Наличие такого инварианта позволяет «не заглядывать вперед» к последующим и «не оборачиваться назад» к предыдущим шагам цикла, ибо на них делается то же самое.

Аналогичная ситуация имеет место в рекурсии. Только она усугубляется тем, что при ветвящейся рекурсии «исторический» подход вообще неприменим, поскольку: «Функция F выполняет ... и вызывает F второй раз, которая выполняет ... и вызывает F в третий раз ... а потом, когда опять вернется в первый вызов, вызовет F еще раз во второй раз...».

Отсюда первая заповедь: алгоритм должен разрабатываться, не выходя за рамки текущего рекурсивного вызова. Остальные принципы уже упоминались:

- рекурсивная функция разрабатывается как обобщенный шаг процесса, который вызывается в произвольных начальных условиях и приводит к следующему шагу в некоторых новых условиях;

- для шага процесса – рекурсивного вызова, необходимо определить инварианты – сохраняемые в процессе выполнения алгоритма условия и соотношения;

- начальные условия очередного шага должны быть формальными параметрами функции;

- начальные условия следующего шага должны быть сформированы в виде фактических параметров рекурсивного вызова;

- локальными переменными функции должны быть объявлены все переменные, которые имеют отношение к протеканию текущего шага процесса и к его состоянию;

- в рекурсивной функции обязательна проверка условий завершения рекурсии, при которых следующий шаг процесса не выполняется.

Этапы разработки рекурсивной функции. Сознательное ограничение процесса проектирования рекурсивной функции текущим шагом сильно меняет и технологию проектирования программы. Прежде всего классический принцип последовательного приближения к цели, последовательной детализации алгоритма здесь очень сильно ограничен, поскольку цель достигается всем процессом, а не отдельным шагом. Отсюда следует рекомендация, сильно смахивающая на фокус: необходимо разработать ряд самостоятельных фрагментов рекурсивной функции, которые в совокупности автоматически приводят к заветной цели. Попутно нужно заметить, что если попытки отследить рекурсию непродуктивны, то столь же ограничены и возможности отладки уже написанных программ.

Итак, перечислим последовательность и содержание шагов в проектировании и «сведении вместе» фрагментов рекурсивной функции.

1. «Зацепить рекурсию» – определить, что составляет шаг рекурсивного алгоритма.

2. Инварианты рекурсивного алгоритма. Основные свойства, соотношения, которые присутствуют на входе рекурсивной функции и которые сохраняются до следующего рекурсивного вызова, но уже в состоянии, более близком к цели.

3. Глобальные переменные – общие данные процесса в целом.

4. Начальное состояние шага рекурсивного алгоритма – формальные параметры рекурсивной функции.

5. Ограничения рекурсии – обнаружение «успеха» – достижения цели на текущем шаге рекурсии и отсечение «неудач» – заведомо неприемлемых вариантов.

6. Правила перебора возможных вариантов – способы формирования рекурсивного вызова.

7. Начальное состояние следующего шага – фактические параметры рекурсивного вызова.

8. Содержание и способ обработки результата – полный перебор с сохранением всех допустимых вариантов, первый возможный, оптимальный.

9. Условия первоначального вызова рекурсивной функции в **main**.

Рекурсия и математическая индукция. Принцип программирования рекурсивных функций имеет много общего с методом математической индукции. Напомним, что этот метод используется для доказательства корректности утверждений для бесконечной последовательности состояний, а именно: если утверждение верно в начальном состоянии, а из его справедливости в n -м состоянии можно доказать его справедливость в $n+1$ -м, то такое утверждение будет справедливым всегда. Этот принцип и применяется при разработке рекурсивных функций: сама рекурсивная функция представляет собой переход из n -го в $n+1$ -е состояние некоторого процесса. Если этот переход корректен, то есть соблюдение некоторых условий на входе функции приводит к их соблюдению на выходе (в рекурсивном вызове), то эти условия будут соблюдаться во всей цепочке состояний (при безусловной корректности первого вызова). Отсюда следует, что самое важное в определении рекурсии – выделить те условия (инварианты), которые соблюдаются (сохраняются) во всех точках процесса, и обеспечить их справедливость от входа в рекурсивную функцию до ее рекурсивного вызова. При этом «категорически не приветствуется» заглядывать в следующий шаг рекурсии или интересоваться состоянием процес-

са на предыдущем шаге. Да в этом и нет необходимости с точки зрения приведенного здесь метода доказательства.

Рекурсия и поисковые задачи. С помощью рекурсии легко решаются задачи, связанные с поиском, основанном на полном или частичном переборе возможных вариантов. Принцип рекурсивности заключается здесь в том, что процесс поиска разбивается на шаги, на каждом из которых выбирается и проверяется очередной элемент из множества, а алгоритм поиска повторяется, но уже для «оставшихся» данных. При этом вовсе не важно, каким образом цепочка шагов достигнет цели и сколько вариантов будет перебираться. Единственное, на что следует обратить внимание, – полнота перебираемых вариантов с точки зрения комбинаторики.

Само множество, в котором производится поиск, обычно реализуется в виде глобальных данных, в которых каждый шаг выбирает необходимые элементы, а по завершении поиска возвращает их обратно.

Результат рекурсивной функции. Результат рекурсивной функции обычно связан со способом перебора вариантов и методом достижения цели в процессе рекурсивного поиска.

1. Используется полный перебор возможных вариантов и вывод (сохранение) всех вариантов, достигающих цели. Обычно рекурсивная функция имеет результат **void**, следовательно, она не может повлиять на характер последующего протекания процесса поиска. Если при поиске обнаруживаются подходящие варианты (успешное завершение рекурсии), то они могут сохраняться в глобальной структуре данных, с которой работают все шаги рекурсивного алгоритма.

2. Рекурсивная функция выполняет поиск первого попавшегося успешного варианта. Ее результатом обычно является логическое значение. При этом **истина** соответствует успешному завершению поиска, а **ложь** – неудачному. Общая для всех алгоритмов схема: если рекурсивный вызов возвращает **истину**, то она должна быть немедленно «передана наверх», то есть текущий вызов также должен быть завершен со значением **истина**. Если рекурсивный вызов возвращает **ложь**, по поиск должен быть продолжен. При завершении полного перебора всех вариантов рекурсивная функция также должна вернуть **ложь**. Характеристики оптимального варианта могут быть возвращены в глобальных данных либо посылке.

3. При поиске производится выбор между подходящими вариантами наиболее оптимального. Обычно для этого используется

минимум или максимум какой-либо характеристики выбираемого варианта. Тогда рекурсивная функция возвращает значение, которое служит оценкой для всех просмотренных ею вариантов, а текущий рекурсивный вызов выбирает из них минимум или максимум с учетом данных текущего шага.

Все сказанное о результате рекурсивной функции касается только самого процесса выбора вариантов. Открытым остается вопрос о том, как возвратить подмножество (последовательность) выбранных элементов, дающих оптимальный результат, и сопровождающие их характеристики. Здесь также возможны варианты:

- при полном переборе и поиске первого подходящего варианта рекурсивная функция сама выводит параметры выбранного варианта в случае успешного поиска. Это приемлемо, но не очень хорошо с точки зрения технологии программирования;

- при полном переборе и поиске первого подходящего варианта выбранный записывается в область глобальных данных или возвращается по ссылке;

- при поиске оптимального варианта каждый шаг получает от рекурсивного вызова структуру данных с параметрами оптимального варианта, выбирает из них одну, модифицирует и возвращает «наверх» с учетом текущего шага. Здесь удобно использовать динамические структуры данных (списки, динамические массивы), а также структурированные переменные, содержащие статические данные достаточной размерности.

Рекурсия как повод к размышлению. И последнее. В Нагорной проповеди Нового Завета Иисус высказал одну из заповедей блаженства: «Итак, не заботьтесь о завтрашнем дне, ибо завтрашний сам будет заботиться о своем: довольно для каждого дня своей заботы». Сказанное справедливо и в проектировании рекурсивной функции: следует сосредоточить внимание на текущем шаге рекурсии, не заботясь о том, когда она была вызвана и каков будет ее следующий шаг: на самом деле он будет делать то же самое, что и текущий (хотя и не написанный). Если «сегодняшний» вызов функции корректен и все ее действия приводят к такому же корректному вызову ее «завтра», то цель рано или поздно будет достигнута.

Трудоемкость рекурсивных алгоритмов. Трудоемкость – это зависимость времени выполнения алгоритма от размерности входных данных. В рекурсивных функциях размерность входных данных определяет глубину рекурсии. Если имеется ветвящаяся ре-

курсия – цикл из m повторений, то при глубине рекурсии N общее количество рекурсивных вызовов будет порядка m^N , поскольку с каждым шагом рекурсии оно увеличивается в m раз. Показательный характер функции говорит о том, что трудоемкость рекурсивных алгоритмов значительно превышает трудоемкость известных нам алгоритмов сортировки и поиска (см. раздел 2.6).

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Линейная рекурсия. Простейшим примером рекурсии является линейная рекурсия, когда функция содержит единственный условный вызов самой себя. В таком случае рекурсия становится эквивалентной обычному циклу. Действительно, любой циклический алгоритм можно преобразовать в линейно-рекурсивный, и наоборот.

```
//----- Рекурсивный алгоритм вычисления факториала
int fact(int n) {
if (n==1) return 1;
return n * fact(n-1); }
//----- Циклический алгоритм вычисления факториала
int fact(int n) {
for (int s=1; n!=0; n--) s *=n;
return s;}
```

Генерация вложенных описаний. Естественным выглядит использование рекурсии при обработке и интерпретации описаний, допускающих вложенность. Здесь просто на каждую единицу описания необходимо спроектировать функцию, которая рекурсивно вызывает сама себя при обнаружении вложенного фрагмента.

Пусть требуется «развернуть» текстовую строку, в которой повторяющиеся фрагменты заключены в скобки, а после открывающей скобки может находиться целая константа, задающая число повторений этого фрагмента в выходной строке. Например: «aaa(3bc(4d)a(2e))aaa» разворачивается в «aaabcbcdaddddaeeebcdaddddaeeebcdaddddaeeaaaa».

1. Шаг рекурсии – отработка заключенного в скобках фрагмента. Инвариант рекурсии: функция получает указатель на первый за скобкой символ фрагмента и должна при рекурсивном вызове передать такой же указатель на вложенный фрагмент.

```
void step(char *p){
... if (*p=='('){
    p++; step(p);
}
}
```

2. Результат работы – сгенерированная строка – может быть глобальным массивом. В процессе ее заполнения необходим также глобальный указатель, которым будут пользоваться все рекурсивные вызовы. Более естественно передать его всем через ссылку. Отсюда имеем окончательный список параметров.

```
void step(char *p, char *&out){
... if (*p=='('){
    p++; step(p,out);
}
}
```

3. Шаг рекурсии состоит в извлечении целой константы – счетчика повторений. Затем внешний цикл производит заданное количество повторений, а внутренний – переписывает символы текущего фрагмента из входной строки в выходную, пока не встретит конца фрагмента (символ ') или конец строки – «защита от дурака» и первоначальный вызов).

```
void step(char *p, char *&out){
for (int n=0; while(*p>='0' && *p<='9') // Накопление константы
    n=n*10+*p++ - '0';
if (n==0) n=1; // При отсутствии – n=1
while(n--!=0){ // Цикл повтора фрагмента
    for(char *q=p; *q!=0 && *q!=''); q++){
        if (*q!='(') // Цикл посимвольного копирования
            *out++ = *q; // Все, кроме '(' – копировать
        else {
            q++; // Пропустить '('
            step(q,out); // Рекурсия для вложенного фрагмента
        }
    }
}}
```

4. Необходимо еще раз обратить внимание на инвариант процесса: каждый шаг должен «брать на себя» текущий фрагмент и, соответственно, передавать рекурсивным вызовам вложенные фрагменты. Но отсюда следует, что сам он должен «пропускать» эти фрагменты в своем алгоритме. Между вызываемой и вызывающей функцией должна быть «обратная связь» по результату: каждый рекурсивный вызов должен возвращать указатель, продвинутый по входной строке на просмотренный фрагмент. С учетом тонкостей пропуска закрывающихся скобок получим окончательный вариант.

```
//-----34-01.cpp
//---- Генерация вложенных повторяющихся фрагментов
char *step(char *p, char *&out){
int n=0; char *q;
while(*p>='0' && *p<='9') // Накопление константы
    n=n*10+*p++ - '0' ;
```



```

if (n==0) n=1; // При отсутствии n=1
while(n--!=0){ // Цикл повтора фрагмента
    for(q=p; *q!=0 && *q!='('; q++){
        if (*q!= '(') // Цикл посимвольного копирования
            *out++ = *q; // Все, кроме '('
        копировать
    }
    else {
        q++; // Пропустить '('
        q=step(q,out); // Рекурсия для вложенного фрагмента
    }
}
if (*q== '(') q++;
return q;}

```

5. В заключение необходимо проверить условия первоначального вызова. Если передать на вход функции любую строку, не начинающуюся с целой константы, то она будет считать всю ее повторяющимся фрагментом с числом повторов, равным 1. Это обеспечат сделанные нами добавления – **n=1** при отсутствии константы, а также завершение по концу строки.

```

void main(){
char s[80],*ps=s; step("aaa(2b(3cd)b)aaa",ps); *ps=0;
puts(s); }

```

Задача о восьми ферзях. Расположить восемь ферзей на шахматной доске так, чтобы они не находились друг у друга «под боем».

1. Поскольку ферзи «бьют» друг друга по вертикали (то есть на каждой вертикали их не более одного), то шаг рекурсии состоит в выставлении ферзя на очередную вертикаль. Инвариант процесса – первые **i-1** ферзей уже корректно выставлены, шаг добавляет еще одного ферзя, сохраняя корректность. Формальный параметр шага – номер вертикали (**i**), фактический параметр рекурсивного вызова – номер следующей вертикали (**i+1**). Алгоритм ищет первую подходящую расстановку и возвращает логическое значение – расстановка найдена (1) или не найдена (0). Общие данные представляют собой доску с уже выставленными ферзями, достаточно иметь одномерный массив, индекс в котором обозначает позицию ферзя по вертикали, а значение – позицию по горизонтали.

```

int R[8];
int step(int i){
    ... step(i+1);
}

```

2. Перебор вариантов заключается в последовательном выставлении очередного ферзя на все восемь клеток вертикали. Если после выставления он находится под боем, клетка пропускается. Если нет, то производится попытка выставить следующего за ним

вызовом рекурсивной функции. Схема поиска первого подходящего варианта говорит о том, что при положительном результате рекурсивного вызова (цепочка построена до конца) необходимо прервать поиск и вернуть этот вариант «также и от себя». В противном случае – перебор продолжается. По окончании просмотра – вернуть 0.

```
int R[8];
int step(int i){
for (int j=0; j<8; j++){
    R[i]=j;
    if (!TEST(i)) continue;    // Под боем - пропустить
    if (step(i+1)) return 1;    // Цепочка построена - выйти
    }
return 0;}                    // Цикл завершен - неудача
```

3. Поскольку каждый ферзь «выставляется» в глобальном массиве, то по завершении цепочки «успешных» выходов из рекурсивных вызовов в нем и будет находиться первый подходящий вариант. И наконец последние штрихи. В рекурсивной функции, «ретранслирующей успех» от вызываемой функции к вызывающей, нет первопричины этого «успеха». Ферзи считаются успешно выставленными, если рекурсивная функция достигает несуществующей вертикали. Эта проверка должна быть сделана в самом начале тела функции. Функция **TEST** проверяет нахождение **i**-го ферзя со всеми предыдущими ферзями на одной горизонтали и диагонали. Первоначально функция вызывается для **i=0**.

```
//-----34-02.cpp
//----- Задача о восьми ферзях
int R[8];
int TEST(int i){
    for (int j=i-1; j>=0; j--){
        if(R[i]==R[j]) return 0;    // По горизонтали
        if(abs(R[i]-R[j])==i-j) return 0;    // По диагонали
    }
return 1; }
int step(int i){
if (i==8) return 1;
for (int j=0; j<8; j++){
    R[i]=j;
    if (!TEST(i)) continue;    // Под боем - пропустить
    if (step(i+1)) return 1;    // Цепочка построена - выйти
    }
return 0;}                    // Цикл завершен - неудача
#include <stdio.h>
void main(){ step(0);
for (int i=0; i<8; i++) printf("%d ",R[i]);
printf("\n"); }
```

Поиск выхода в лабиринте. С точки зрения математики лабиринт представляет собой граф, а алгоритм поиска выхода из него производит поиск пути, соединяющего заданные вершины. В данном примере мы воспользуемся более простым, естественным представлением лабиринта. Зададим его в виде двумерного массива, в котором значение 1 будет обозначать «стенку», а 0 – «проход».

```
int LB[10][10]={
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,0,1,0,0,0,1,1},
...
{1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1}};
```

1. Рекурсивная функция пытается сделать «шаг в лабиринте» в одном из четырех направлений из точки, в которой мы сейчас находимся. Инвариант процесса состоит в том, что если мы находимся в «корректной точке» (не на стене лабиринта) и не вернулись в нее повторно, то в соседней точке будет то же самое. Рекурсивный характер алгоритма состоит в том, что в каждой соседней точке реализуется тот же самый алгоритм поиска. Формальными параметрами рекурсивной функции в данном случае являются координаты точки, из которой в данный момент осуществляется поиск. Фактические параметры – координаты соседней точки.

```
void step(int x,int y){
step(x+1,y);...
step(x,y+1);...
step(x-1,y);...
step(x,y-1);...
}
```

2. Поиск производится по принципу «первого подходящего», вид результата и способ его формирования аналогичен предыдущему примеру. Определение первоначального «успешного варианта» – достижение границы лабиринта. Отсечение недопустимых вариантов – текущая точка является «стеной».

```
int step(int x,int y){
if (LB[x][y]==1) return 0;
if (x==0 || x==9 || y==0 || y==9) return 1; ...
if (step(x+1,y)) return 1;
if (step(x,y+1)) return 1;
if (step(x-1,y)) return 1;
if (step(x,y-1)) return 1;
return 0;
}
```

3. Сами параметры успешного варианта – путь к выходу из лабиринта – могут быть сохранены в глобальных данных – в виде специально выделенных для этого значений. Тогда при входе в очередную точку ее нужно отмечать (значение – 2), а если поиск не привел к цели – снимать отметку перед возвратом из рекурсивной функции. Отметка пройденных точек позволяет «убить второго зайца» – исключить заикливания алгоритма. Для этого нужно просто добавить еще одно ограничение – при входе в очередную точку сразу же возвращается отрицательный результат, если это «стенка» и если она уже отмечена.

```
//-----34-03.cpp
//----- Поиск выхода в лабиринте
int step(int x,int y){
  if (x<0 || x>9 || y<0 || y>9) return 1;    // Края
  if (LB[x][y]!=0) return 0;                 // Стенки и циклы
  LB[x][y]=2;                               // Отметить точку
  if (step(x+1,y)) return 1;
  if (step(x,y+1)) return 1;
  if (step(x-1,y)) return 1;
  if (step(x,y-1)) return 1;
  LB[x][y]=0;                               // Снять отметку
  return 0;}
```

Обход конем шахматной доски. Приведенных выше примеров вполне достаточно, чтобы проиллюстрировать следующий пример лишь формальным перечислением принятых решений:

- шаг процесса заключается в выставлении коня на очередную клетку доски с заданными координатами;

- рекурсивная функция делает восемь попыток движения конем на соседние клетки, используется массив относительных смещений;

- доска представляет собой глобальный двумерный массив, при «прохождении» коня клетка заполняется номером шага, этим сохраняется последовательность ходов при достижении успеха, это же служит защитой от повторных прохождений той же самой клетки;

- глобальный счетчик номера хода используется для определения условий достижения «успеха» – пройдены все клетки доски;

- реализован алгоритм поиска первого подходящего варианта;

- рекурсия ограничена выходом за пределы доски и повторным обходом отмеченной (пройденной) клетки.

```
//-----34-04.cpp
//-----Обход шахматной доски конем
#define N 5
int desk[N][N];                               // Поля доски
```

```

int nstep; // Номер шага
int step(int x0, int y0){
static int xy[8][2] =
    {{ 1,-2},{ 1, 2},{-1,-2},{-1, 2},{ 2,-1},{ 2, 1},{-2, 1},{-2,-1}};
if (nstep == N*N) return 1; // Все поля отмечены - успех
if (x0 < 0 || x0 >= N || y0 < 0 || y0 >= N )
    return 0; // Выход за пределы доски
if (desk[x0][y0] !=0)
    return 0; // Поле уже пройдено
desk[x0][y0] = ++nstep; // Отметить свободное поле
for ( int i=0; i<8; i++) // Локальный параметр - номер хода
if (step(x0+xy[i][0], y0+xy[i][1]))
    return 1; // Поиск успешного хода
nstep--; // Вернуться на ход назад
desk[x0][y0] = 0; // Стереть отметку поля
return 0; } // Последовательность не найдена
#include <stdio.h>
void main(){
for ( int i=0; i<N; i++)
    for ( int j=0; j<N; j++) desk[i][j] =0;
nstep = 0; // Установить номер шага
step(0,0);
for (i=0; i<N; i++,printf("\n"))
for (int j=0; j<N; j++) printf ("%2d ",desk[i][j]);
} // Вызвать функцию для исходной позиции

```

Линейный кроссворд. Для заданного набора слов требуется построить линейный кроссворд. Если окончание одного слова совпадает с началом следующего более чем в одной букве (например, *матрас-расист*), то такие слова можно объединить в цепочку. Первоначально ставится задача – получить любую такую цепочку, окончательно – цепочку минимальной длины.

Начало проектирования любой рекурсивной программы заключается в определении шага рекурсивного процесса. Пусть имеется уже составленная цепочка из выбранных слов. Очередной шаг процесса состоит в попытке присоединения к имеющейся цепочке еще одного слова из оставшихся. Если это возможно, то для новой цепочки необходимо попытаться присоединить следующее слово и так далее, то есть выполнить следующий шаг рекурсивного процесса. Таким образом:

- рекурсивная функция выполняет попытку присоединения очередного слова к уже выстроенной цепочке;
- результатом функции является логическое значение (данную цепочку можно достроить), функция ищет первый подходящий вариант;
- условием завершения рекурсии является отсутствие еще не присоединенных к цепочке слов (успешное завершение) либо невозможность завершения цепочки ни через одно из оставшихся слов (неудача).

Множество возможных вариантов строится на основе обычного комбинаторного перебора всех допустимых сочетаний (последовательностей) из элементов множества (в данном случае – слов). Это множество является глобальной структурой данных, из которой на каждом шаге извлекается очередной элемент, но по завершении просмотра варианта (после рекурсивного вызова) возвращается обратно.

Для представления множества слов будем использовать массив указателей на строки. Извлечение строки из множества будет заключаться в установке указателя на строку нулевой длины. Теперь можем «набросать» общий вид рекурсивной функции:

```
char *w[]={ "РАСИСТ", "МАТРАС", "МАСТЕР", "СИСТЕМА",
            "СТЕРВА", NULL};
int step(char *lw)      // Параметр – текущее слово цепочки
{ int n;               // Результат – можно присоединить
                      // оставшиеся
                      // Проверка условия завершения рекурсии
                      // - все слова из w[] присоединены
for (int n=0; w[n]!=NULL;n++) { // Проверка на присоединение
    char *pw;              // очередного слова
    if (*w[n]==0) continue;
    pw=w[n];              // Пустое слово – пропустить
    w[n]="";              // Исключить проверяемое слово
    if (step(pw))          // Попытка присоединить слово
        return 1;        // Удача - завершить успешно
    w[n]=pw;              // Возвратить исключенное слово
}                          // Неудача – нельзя присоединить
return 0;                 // ни одного слова
}
```

Данный «набросок» не содержит некоторых частных случаев, которые не меняют общей картины:

- проверка условия завершения рекурсии – если массив указателей содержит только пустые строки, то рекурсивная последовательность шагов завершена успешно (все слова выбраны на предыдущих шагах);

- проверка совпадения «хвоста» очередного слова и начала выбираемого на текущем шаге – делается отдельной функцией;

- сама цепочка выбранных слов выводится в процессе «ретрансляции» положительного результата непосредственно на экран в обратном порядке (что не совсем «красиво»). В принципе она может быть сформирована и в глобальных данных.

```
//-----34-05.cpp
//---- Линейный кроссворд
char *w[]={ "РАСИСТ", "МАТРАС", "МАСТЕР", "СИСТЕМА",
            "СТЕРВА", NULL};
int step(char *lw)      // Параметр – текущее слово цепочки
{ int n;
```

```

for (n=0; w[n]!=NULL;n++)
    if (*w[n]!=0) break;
if (w[n]==NULL) // Цепочка выстроена, все слова
    return 1;    // из w[] присоединены
for (n=0; w[n]!=NULL;n++)
{
    char *pw; // Проверка на присоединение
               // очередного слова
    if (*w[n]==0) continue;
    pw=w[n];
    w[n]=""; // Пустое слово – пропустить
              // Исключить проверяемое слово из
              // множества
    if (TEST(lw,pw))
    {
        if (step(pw)) // Попытка присоединить слово
            // Присоединено – попытка вывести
            // цепочку из нового слова
            puts(pw);
        return 1;
    }
    // Удача – вывести слово и выйти
    w[n]=pw;
} return 0; } // Возвратить исключенное слово

```

Чисто технические детали: функция **TEST** проверяет, не совпадает ли окончание первой строки с началом второй, путем обычного сравнения строк при заданной длине «хвоста» первой строки.

```

int TEST(char *s, char *r)
{ int n,k;
  n=strlen(s);
  if (n==0) return 1;
  for (;*s!=0 && n>1; s++,n--)
      if (strncmp(s,r,n)==0) return 1;
  return 0; }

```

Другая техническая проблема – удобство первоначального запуска рекурсии. Функция **TEST** при первом параметре – пустой строке – возвращает ИСТИНУ при любом виде второй строки. Этого достаточно, чтобы запустить первый шаг рекурсии. При наличии пустой строки в качестве параметра функции **step** на первом шаге рекурсии будет производиться безусловная проверка каждого слова на предмет создания цепочки.

```
void main() { step(""); }
```

Линейный кроссворд. Более изящный способ перебора вариантов. Одно из условий успешной реализации поискового алгоритма – полный перебор всех возможных вариантов. Здесь мы не будем вторгаться в область комбинаторики. С точки зрения технологии проектирования рекурсивной функции должен соблюдаться принцип: в исходном множестве элементы могут «тасоваться» каким угодно образом, лишь бы каждый шаг рекурсии обеспечивал

просмотр всех для него возможных, а при получении отрицательного результата – восстанавливал исходную картину. В примере с линейным кроссвордом возможен следующий алгоритм перебора, удовлетворяющий этим условиям:

- для очередного (*i*) слова просматриваются все последующие;
- если начало одного из них (*j*) совпадает с окончанием текущего, то оно переставляется со следующим (*i+1*), то есть замещает в возможной цепочке *i+1*-е слово;
- если рекурсивный вызов не смог достроить цепочку, то переставленные слова возвращаются на свои места;
- при успешном завершении слова будут расположены в нужном порядке.

```
//-----34-06.cpp
//----- Линейный кроссворд с перестановками слов
char *w[]={ "РАСИСТ", "МАТРАС", "МАСТЕР", "СИСТЕМА",
            "СТЕПБА", NULL };
int step( int i ) {                // Параметр - номер слова
if ( w[i+1]==NULL) return 1;      // Успех все слова выставлены
for ( int n=i+1; w[n]!=NULL;n++)  // Проверка на присоединение
    if ( i== -1 || TEST( w[i],w[n])) { // оставшихся слов
        char *q;                  // Переставить следующее
        q=w[i+1]; w[i+1]=w[n]; w[n]=q; // с выбранным
        if ( step(i+1)) return 1;  // Успех - выйти
        q=w[i+1]; w[i+1]=w[n]; w[n]=q; // Вернуть все и продолжить
    }
return 0; }
void main()
{ step(-1); for (int i=0; w[i]!=NULL; i++) puts (w[i]); }
```

Поиск кратчайшего пути в графе. Расстояния между городами заданы матрицей. Для каждой пары городов *i, j* элемент **R(i, j)** матрицы содержит значение расстояния между ними либо 0, если они не связаны непосредственно (матрица симметрична относительно главной диагонали). Для начала – требуется найти значение минимального пути между двумя заданными городами.

Схема рекурсивного процесса поиска для этой задачи принципиально не отличается от предыдущих. Шаг рекурсии – перемещение из текущего города в соседний в поисках пути. Формальные параметры функции (начальное состояние процесса) – индекс текущего города в матрице расстояний и индекс города – цели. «Успешное» ограничение рекурсии – формальные параметры совпадают. Функция включает цикл перебора всех соседей и рекурсивного вызова для каждого из них, если между ними имеется прямой путь. «Зацикливание» предотвращается отметкой пройденных городов.


```

#define N 5
int R[N][N]={0,4,2,0,0},{4,0,0,1,3},{2,0,1,0,6},{0,0,3,0,0},{0,0,6,0,0};
int M[N]={0,0,0,0,0};
... step(int src, int dst){
if (src==dst) return...
if (M[src]==1) return...
M[src]=1;
for (int i=0; i<N; i++){
    if (R[src][i]==0) continue;
    ... step(i,dst);
}
M[src]=0;
return ...; }

```

Далее следуют особенности оптимального поиска. Прежде всего, рекурсивный процесс обеспечивает полный перебор. Рекурсивный вызов возвращает оптимальное значение – минимальное расстояние от текущего города до города – цели, либо **-1**, если путь отсутствует. Текущий шаг рекурсии должен сохранить этот инвариант, полученный от соседей. Для этого он добавляет к каждому допустимому (не равному **-1**) результату рекурсивного вызова расстояние от текущего города до соседа и выбирает из них минимальный, возвращая в качестве «своего» результата.

```

//-----34-07.cpp
//----- Поиск минимального пути между городами
#define N 5
int R[N][N]={0,4,2,0,0},{4,0,0,1,3},{2,0,0,0,6},{0,1,0,0,0},{0,3,6,0,0};
int M[N]={0,0,0,0,0}; // Отметка пройденных городов
int step(int src, int dst){
if (src==dst) return 0; // Успех от цели до цели 0
if (M[src]==1) return -1; // Повторное прохождение - -1
M[src]=1;
int min=-1; // Минимальный путь от src до dst
for (int i=0; i<N; i++){
    if (R[src][i]==0) continue; // Соседи не связаны - пропустить
    int x=step(i,dst); // Результат от соседа до цели
    if (x==-1) continue; // Путь не найден – пропустить
    x+=R[src][i]; // Добавить расстояние до соседа
    if (min==-1 || x < min) // Зафиксировать минимум
        min=x;
}
M[src]=0; // Снять отметку
return min; }

```

Линейный кроссворд – поиск самой короткой цепочки. В предыдущем примере намечена основная схема поиска оптимального варианта. Проблемы возникают, если наряду с самим значением необходимо передать параметры самого варианта. Понятно, что сделать это отдельной глобальной переменной невозможно, потому что каждый рекурсивный вызов будет хранить у себя «недосчитанную» оптимальную конфигурацию. Выход из

создавшегося положения – использовать динамические структуры данных. При этом функция должна получать динамические структуры данных от рекурсивных вызовов, выбирать соответствующую оптимальному варианту, остальные – уничтожать, а к выбранному – присоединять собственные данные.

При составлении линейного кроссворда с перекрытиями рекурсивная функция возвращает построенную с учетом перекрытий результирующую цепочку из слов, присоединяемых к заданному слову, причем цепочку минимальной длины. Сама цепочка передается указателем на строку в динамическом массиве, **NULL**-указатель используется для индикации «неудачи» – невозможности выстроить хотя бы одну цепочку.

Цикл рекурсивного вызова не меняет свою схему, он тоже содержит контекст поиска минимума. Но в данном случае функция проверки «перекрытия» слов возвращает количество «неперекрытых» символов в первом слове, 0 – если первое слово окажется пустой строкой, либо –1, если слова не перекрываются.

```
//-----34-08.cpp
//----- Построение самой короткой цепочки с перекрытиями
char *w[]={"aaa123","3fff","fffaaa","123fff",NULL};
int TEST(char *s, char *r) {
    int n,k; k=n=strlen(s);
    if (n==0) return 0;
    for (;*s!=0 && n>0; s++,n--)
        if (strncmp(s,r,n)==0) return n;
    return -1;}
char *step( int i)
{ char *s,*pp, *pmin= NULL;
  if (w[i+1]==NULL){ // Слово последнее
    s=new char [strlen(w[i])+1]; // -возвратить его
    strcpy(s,w[i]);
    return s; }
  char *smin=NULL; // Указатель на минимальную цепочку
  for ( int n=i+1; w[n]!=NULL;n++){
    int l; char *q;
    if ((l=TEST(w[i],w[n]))!=-1) { // Переставить следующее слово
      q=w[i+1]; w[i+1]=w[n]; w[n]=q; // с выбранным
      if ( (pp=step(i+1) )!=NULL) { // Успех - соединить цепочки
        s=new char[l+strlen(pp)+1]; // с учетом перекрытия
        strcpy(s,w[i]);
        strcat(s,pp+l);
        delete pp;
        if (smin==NULL) smin=s; // Выбор минимальной
        else // по длине
        if (strlen(smin)>strlen(s)) // с уничтожением
            { delete smin; smin=s; } // замещенных
      }
      q=w[i+1]; w[i+1]=w[n]; w[n]=q; // Вернуть все и продолжить
    }
  }
  return smin;}
void main() { char *q=step(0); puts(q); delete q; }
```

Поиск кратчайшего пути – сохранение оптимального варианта. В Си++ для возврата рекурсивной функцией совокупности параметров можно использовать результат функции – структурированную переменную, то есть возвращать структурированный тип по значению. В этом случае все проблемы по распределению памяти для временного хранения результата функции решаются транслятором. В самой функции используются операции присваивания структурированных переменных. Например, при поиске минимального пути пройденную последовательность городов можно возвращать в статическом массиве, включенном в структурированную переменную. В нее же следует включить и само значение минимального пути. Для заполнения такой структуры потребуются контролировать номер шага рекурсии, это делается дополнительным формальным параметром **I**. Заметим, что найденный путь заполняется от конца в обратном порядке.

```
//-----34-09.cpp
//----- Сохранение оптимального пути обхода
#define N 5
struct way{
    int lnt; // Длина цепочки городов
    int min; // Значение пути
    int town[N]; // Последовательность обхода
};
int R[N][N]={0,4,2,0,0},{4,0,0,1,3},{2,0,0,0,6},{0,1,0,0,0},{0,3,6,0,0}};
int M[N]={0,0,0,0,0}; // Отметка пройденных городов
way step(int src, int dst, int l){
    way mway,x; // Оптимальный и текущий результат
    mway.min=-1; // Первоначально результат отрицательный
    mway.town[l]=src; // Заполнить текущий город
    if (src==dst){
        mway.lnt=l; // Запомнить длину цепочки
        mway.min=0;
        return mway; // Успех от цели до цели 0
    }
    if (M[src]==1) return mway; // Повторное прохождение - -1
    M[src]=1;
    for (int i=0; i<N; i++){ // Рекурсия возвращает way
        if (R[src][i]==0) continue; // Соседи не связаны - пропустить
        x=step(i,dst,l+1); // Результат от соседа до цели
        if (x.min===-1) continue; // Путь не найден пропустить
        x.min+=R[src][i]; // Добавить расстояние до соседа
        x.town[l]=src; // Добавить текущий город в путь
        if (mway.min===-1 || x.min < mway.min) // Сохранить новый way
            mway=x;
    }
    M[src]=0; // Снять отметку
    return mway; }
#include <stdio.h>
void main(){
    way w=step(0,3,0);
    printf("\nmin=%d\ntowns:",w.min);
    for (int i=0; i<=w.lnt; i++) printf("%d-",w.town[i]); }
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

1. Используя фрагмент программы просмотра каталога, приведенный ниже, написать рекурсивную функцию поиска файлов с одинаковыми именами во всех подкаталогах диска. (Структуру данных, содержащую имена найденных файлов, можно реализовать в виде глобального односвязного списка.)

```
#include <dir.h>
#define FA_DIREC 0x10
void showdir(char *dir)
{ struct fblk DIR; int done; char dirname[40];
  strcpy(dirname,dir);
  strcat(dirname,"*.");
  done=findfirst(dirname,&DIR,FA_DIREC);
  while(! done)
  { if (DIR.ff_attrib & FA_DIREC)
    {
      if (DIR.ff_name[0] != '.')
        printf ("Подкаталог %s\n",DIR.ff_name);
    }
    else
      printf("Файл %s%s\n",dir,DIR.ff_name);
    done=findnext(&DIR);
  } }
void main() showdir("E:\\a\\b\\"); }
```

2. Реализовать рекурсивный алгоритм построения цепочки из имеющегося набора костей домино.

3. Программа отображает на экране структуру данных – дерево. Для равномерного размещения вершин программа должна «знать» для каждой вершины интервал позиций экрана, который выделен для данного поддерева, и количество вершин в поддереве. Само дерево можно задать статически (инициализация).

4. Расстояния между городами заданы матрицей (если между городами i, j есть прямой путь с расстоянием N , то элементы матрицы $A(i, j)$ и $A(j, i)$ содержат значение N , иначе 0). Написать программу поиска минимального пути обхода всех городов без посещения дважды одного и того же города (задача коммивояжера).

5. Разместить на шахматной доске максимальное количество коней так, чтобы они не находились друг у друга «под боем».

6. Программа генерирует текст из строки, содержащей определения циклических фрагментов вида «...(**Иван, Петр, Федор = Жил-был * у самого синего моря**)...» Символ «*» определяет место подстановки имени из списка в очередное повторение фрагмента. Допускается вложенность фрагментов. Полученный текст поместить в выходную строку.

7. Задан набор слов (массив указателей на строки). Построить из них любую цепочку таким образом, чтобы символ в конце слова совпадал с символом в начале следующего.

8. Задан набор слов в виде массива указателей на строки. Построить из них любую цепочку таким образом, чтобы символ в начале следующего слова совпадал с одним из символов в середине предыдущего (не первым и не последним).

9. Задан массив целых. Построить из них любую последовательность таким образом, чтобы последняя цифра предыдущего числа совпадала с первой цифрой следующего.

10. Задача раскраски карты. Страны на карте заданы матрицей смежности. Если страны i, j имеют на карте общую границу, то элемент матрицы $A[i, j]$ равен 1, иначе – 0. Смежные страны не должны иметь одинакового цвета. «Раскрасить» карту минимальным количеством цветов.

11. Разместить на шахматной доске максимальное количество слонов и ладей так, чтобы они не находились друг у друга «под боем».

12. Задача проведения границы на карте («создание военных блоков»). Страны на карте заданы матрицей смежности. Если страны i, j имеют на карте общую границу, то элемент матрицы $A[i, j]$ равен 1, иначе – 0. Необходимо разбить страны на две группы так, чтобы количество пар смежных стран из противоположных групп было минимальным.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определите вид рекурсии (линейная, ветвящаяся), сформулируйте содержательный результат рекурсивного алгоритма.

```
//-----34-12.cpp
//----- 1
long F1(int n)
{ if (n==1) return 1;
  return (n * F1(n-1)); }
//----- 2
double F2(double *pk, double x, int n)
{
  if (n==0) return(*pk);
  return *pk + x *F2(pk+1,x,n-1);
}
void z3()
{ double B[]={ 5.,0.7,4.,3. },X=3., Y;
  Y = F2(B,X,4); }
```

```

//----- 3
void F3(int in[], int a, int b)
{ int i,j,mode;
  if (a>=b) return;
  for (i=a, j=b, mode=1; i != j; mode >0 ? i++ : j--)
    if (in[i] > in[j])
      { int c;
        c = in[i]; in[i] = in[j]; in[j]=c; mode = -mode;
      }
  F3(in,a,i-1); F3(in,i+1,b);
}
//----- 4
char *F4(char *p, char *s) {
  if ( *s =='\0') return p;
  *p++ = *s;
  p=F4(p, s+1);
  *p++ = *s;
  return p;}
void z4()
{ char *q, S[80];
  *F4(S, "abcd")=0; }
//----- 5
void F5(char *p, char *s){
  if ( *s =='\0') return;
  *p++ = *s;
  F5(p, s+1);
  *p++ = *s; }
void z5()
{ char *q, S[80];
  q = S; F5(q,"abcd"); *q=0; }
//----- 6
void F6(int p[], int nn)
{ int i;
  if (nn==1) { p[0]=0; return; }
  for (i=2; nn % i !=0; i++);
  p[0]=i;
  F6(p+1,nn / i); }
//----- 7
long F7(int n){
  if (n==0 || n==1) return 1;
  return F7(n-1)+F7(n-2);}

```

3.5. СТРУКТУРЫ ДАННЫХ. МАССИВЫ УКАЗАТЕЛЕЙ

Не в совокупности ищи единства,
но в единообразии разделения.

Козьма Прутков

Структуры данных «в узком смысле». Среди структур данных можно выделить группу, играющую в программе роль «массивов», то есть предназначенную для хранения, упорядочения и поиска элементов данных (чаще всего одинакового типа). Сами хранимые элементы данных являются ее «прикладной» частью. Орга-

низирующая часть структуры данных, выполняющая функции хранения и упорядочения, и будет в дальнейшем изложении называться **структурой данных**.

Любая структура данных с точки зрения внешнего пользователя представляет собой упорядоченную последовательность хранимых элементов данных (своего рода «виртуальный массив»). Получить элементы данных из нее возможно, обходя ее в определенном раз и навсегда порядке. В этом же порядке нумеруются и хранимые в ней элементы.

Логический номер элемента данных – номер элемента, полученный при естественном последовательном обходе структуры данных (рис. 3.7).

Различают неупорядоченные и упорядоченные структуры данных. В последних все операции с ее элементами производятся с сохранением этого порядка. Перечислим «джентльменский набор» операций над структурами данных, который иллюстрирует их в последующем изложении:

- выбор (поиск) по логическому номеру;
- «быстрый» (двоичный) поиск в упорядоченной структуре данных по ключу (значению);
- добавление последним;
- включение по логическому номеру;
- включение с сохранением упорядоченности;
- удаление по логическому номеру;
- сортировка неупорядоченной структуры данных.

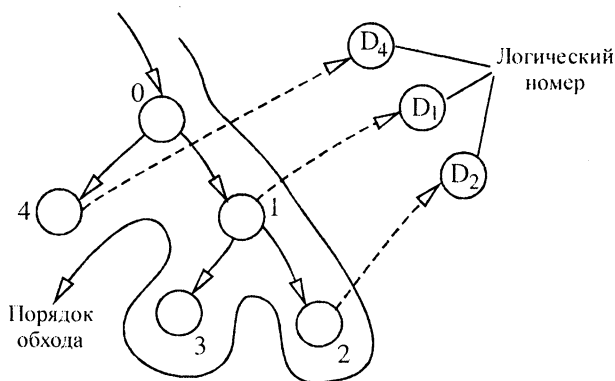


Рис. 3.7

Структуры данных могут включать в себя хранимые элементы данных. В этом случае они уже заранее «настроены» на заданный тип, решают проблемы с распределением памяти под него, а функции принимают и передают его по значению.

В случае, если структуры данных хранят указатели на элементы данных, последние становятся независимыми от структуры. Предельный случай такой независимости: структура данных хранит указатель типа **void*** на элементы неизвестного ей (произвольного) типа.

Статические и динамические структуры данных. Структура данных характеризуется количеством, размерностями переменных и их взаимосвязями. Если они определяются при трансляции, а при выполнении программы не могут быть изменены, то речь идет о статической структуре данных, если определяются при выполнении – о динамической. Естественно, алгоритмы работы со структурами данных не зависят от «происхождения» последних.

Статическая структура данных – совокупность фиксированного количества переменных постоянной размерности с неизменным характером связей между ними.

Динамическая структура данных – совокупность переменных, количество, размерность или характер связей между которыми меняется во время работы программ.

Динамические структуры данных базируются на двух элементах языка программирования:

- динамических переменных, количество которых может меняться и в конечном счете определяется самой программой. Кроме того, возможность создания динамических массивов позволяет говорить о данных переменной размерности;

- указателях, которые обеспечивают непосредственную взаимосвязь данных и возможность изменения этих связей.

Таким образом, близко к истине и такое определение: динамические структуры данных – это динамические переменные и массивы, связанные указателями.

Массив указателей как тип данных и как структура данных. Переменная, тип данных которой звучит как «массив указателей», в Си выглядит так:

```
double *p[20];
```

В соответствии с принципом контекстного определения типа данных переменную **p** следует понимать как массив (операция **[]**), каждым элементом которого является указатель на переменную

типа **double** (операция *). Исходя из правил адресной арифметики, эту переменную можно рассматривать как массив указателей на отдельные переменные типа **double** и как массив указателей на массивы этих переменных. Пока что ограничимся первым, более простым случаем.

Переменная **p** является массивом указателей как тип данных, но не как структура данных. Чтобы превратиться в структуру данных, она должна быть дополнена указуемыми переменными и указателями (связями) (рис. 3.8).

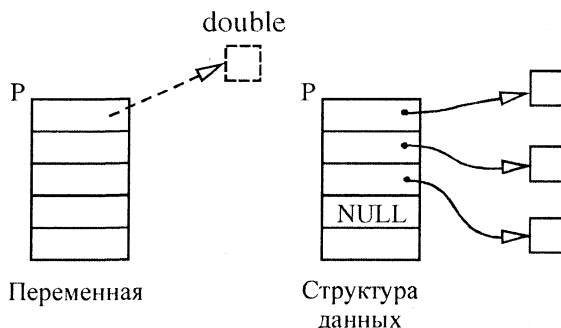


Рис. 3.8

Способы формирования массивов указателей. Статический массив указателей формируется при трансляции: переменные (сам массив указателей и указуемые переменные) определяются статически, как обычные именованные переменные, а указатели инициализируются. Структура данных включена непосредственно в программный код и «готова к работе».

```
double a1,a2,a3, *pd[] = { &a1, &a2, &a3, NULL};
```

Сразу же отметим один технический момент. Размерность массива указателей и текущее количество указателей в нем – вещи разные. Обычно массив указателей содержит **последовательность указателей**, ограниченную **NULL**-указателем.

Промежуточные варианты массива указателей могут содержать как статические, так и динамические компоненты.

Вариант 1. Переменные определяются статически, указатели устанавливаются программно. Этот вариант наиболее часто используется, когда указуемые переменные представлены массивом.

```
double d[19], *pd[20];
for (i=0; i<19; i++) pd[i] = &d[i];
pd[i] = NULL;
```

Вариант 2. Указуемые переменные создаются динамически, массив указателей – статически.

```
double *p, *pd[20];
for (i=0; i<19; i++){
    p = new double; *p = i; pd[i] = p;
}
pd[i] = NULL;
```

Все переменные **динамического массива указателей**, в том числе и сам массив указателей, создаются динамически. Результатом работы является указатель на создаваемый массив указателей (адрес массива указателей).

```
double **pp, *p;
pp = new double *[20];           // Память под массив
for (i=0; i<19; i++)             // Из 20 указателей типа double*
{
    p = new double;
    *p = i; pp[i] = p;
}
pp[i] = NULL;
```

Работа с массивом указателей. При работе с массивом указателей используются контексты:

- **pd[i]** – **i**-й указатель в массиве;
- ***pd[i]** – значение **i**-й указуемой переменной.

Алгоритмы работы с массивом указателей и обычным массивом внешне очень похожи. Разница же состоит в том, что размещение данных в обычном массиве соответствует их **физическому порядку** следования в памяти, а массив указателей позволяет сформировать **логический порядок** следования элементов, в соответствии с размещением указателей на них. Тогда изменение порядка следования (включение, исключение, упорядочение, перестановка), которое в обычном массиве заключается в перемещении самих элементов, в массиве указателей должно сопровождаться соответствующими операциями над указателями. Очевидные преимущества возникают, когда сами указуемые переменные достаточно большие либо перемещение их невозможно по каким-то причинам (например, на них ссылаются другие части программы). Для сравнения приведем функции сортировки массива и массива указателей.

```
//-----35-01.cpp
//--- Сортировка массива и массива указателей
#include <stdio.h>
void sort1 (double d[],int sz)
{ int i,k;
  do {
```

```

for ( k=0, i=0; i<sz-1; i++)
    if (d[i] > d[i+1])
    { double c; c = d[i];
      d[i] = d[i+1]; d[i+1] = c; k=1;}
}while (k); }

void sort2 (double *pd[])
{ int i,k;
  do {
    for ( k=0, i=0; pd[i+1]!=NULL;i++)
      if (*pd[i] > *pd[i+1]) // Сравнение указуемых переменных
      {double *c; // Перестановка указателей
       c = pd[i];pd[i] = pd[i+1];pd[i+1] = c; k = 1; }
    }while (k);}

```

Динамический массив указателей (ДМУ). Если массив указателей создается в процессе работы программы, то для доступа к нему в свою очередь необходим указатель (рис. 3.9). По правилам работы с динамическими переменными и массивами он должен иметь тип «указатель на указуемый тип», то есть указатель на указатель. В соответствии с принципами контекстного определения типа это можно сделать так:

```
double **pp;
```

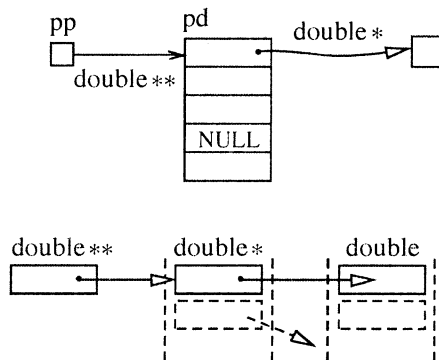


Рис. 3.9

Поскольку по правилам адресной арифметики любой указатель может ссылаться как на отдельную переменную, так и на область памяти (массив), то в применении к двойному указателю получаются четыре варианта интерпретации переменной, а именно:

- указатель на одиночный указатель на переменную типа **double**;
- указатель на одиночный указатель на массив переменных типа **double**;

- указатель на массив, содержащий указатели на одиночные переменные типа **double**;
- указатель на массив, содержащий указатели на массивы переменных типа **double**.

Напомним, что конкретный способ интерпретации указателя задается программно (в зависимости от того, как программа работает с указателем).

Третья интерпретация позволяет использовать двойной указатель для работы с известными нам массивами указателей следующим образом:

```
double **pp, *pd[20];
pp = pd;           // Или pp = &pd[0];
pp[i]              // Эквивалентно pd[i]
*pp[i]             // Эквивалентно *pd[i]
```

Здесь повторяется та же самая система эквивалентности обычных массивов и указателей – типов **double*** и **double[]**, но применительно к массивам указателей; **double *[]** задает массив указателей, а **double**** – указатель на него. Причем синтаксис работы с обоими переменными идентичен.

Массив указателей типа **double *[]** является статической структурой данных, размерность которой определяется при трансляции. Двойной указатель типа **double**** может ссылаться и на динамический массив указателей, который создается во время работы программы под заданную размерность:

```
//-----35-02.cpp
//----- Динамический массив указателей из заданного массива
#include <stdio.h>
double **create( double in[], int n){
double **pp = new double *[n+1]; // Создать ДМУ
    for ( int i=0; i<n; i++) {      // Создать динамическую переменную
        pp[i] = new double;        // и запомнить ее адрес в ДМУ
        *pp[i]=in[i];              // Копировать значение из входного
    }
pp[ n] = NULL;                    // Ограничитель ДМУ
return pp; }                     // Возвратить указатель на ДМУ
```

Массив указателей на массивы как альтернатива двумерному массиву. Указуемым объектом в массиве указателей может быть как отдельная переменная, так и массив таких переменных. В последнем случае мы имеем функциональный аналог двумерного массива: первый индекс выбирает указатель на массив, второй – элемент этого массива. Более того, аналогия здесь даже синтаксическая: выражение **p[i][j]** приемлемо в обоих случаях и с точки зрения логической организации данных обозначает одно и то же – **j**-й элемент **i**-й строки (рис. 3.10). Преимущество массива указате-

лей проявляется, если речь идет о переменной размерности. Двумерный массив всегда должен иметь фиксированную вторую размерность (для вычисления адресов транслятор должен знать длину строки матрицы). Для массива указателей – это излишне.

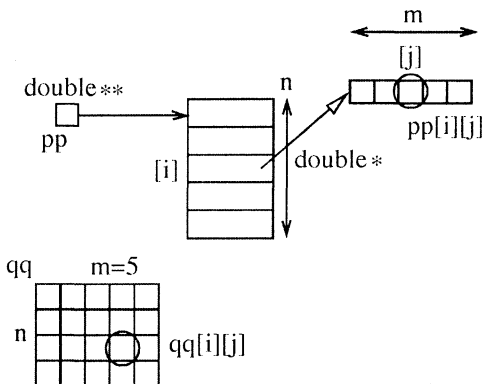


Рис. 3.10

```
//-----35-03.cpp
//--- Матрица любой размерности - массив указателей на массивы
#include <stdio.h>
double F(double **p , int n, int m )
{ double s=0 ;
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++) s+=p[i][j];
  return s; }
//----- Пример вызова для статической структуры данных
double a1[3]={2,3,4};
double a2[3]={2,3,4};
double a3[3]={2,3,4};
double *pp[3]={a1,a2,a3};
void main(){
  printf("sum(3,3)=%2.0lf\n",F(pp,3,3)); // Вызов для матрицы 3x3
  printf("sum(2,2)=%2.0lf\n",F(pp,2,2)); } // Вызов для ее части 2x2
```

Массивы указателей на строки. Другой широко распространенной интерпретацией массива указателей на массивы является массив указателей на строки. Он создается для указуемых переменных типа **char** и обычно понимается как массив указателей на массивы символов (строки) с соответствующим определением:

```
char *pc[20];
```

Способы формирования массива указателей на строки. В полностью статической структуре данных массив указателей создается статически и инициализируется строковыми константами – вся структура данных включается в программный код. На-

помним, что строковая константа во всех контекстах понимается как указатель на сформированный транслятором массив, инициализированный символами строки.

```
char    *pc[] = { "aaa", "bbb", "ccc", NULL};
```

Статический массив указателей может ссылаться на строки, для размещения которых используется двумерный массив символов (массив строк). В этом случае динамически назначаются только указатели.

```
char    *pc[20], cc[19][80];
for (i=0; i<19; i++) pc[i] = cc[i];
pc[i] = NULL;
```

Здесь используются две особенности организации двумерных массивов. Во-первых, двумерный массив интерпретируется как массив элементов первого индекса, состоящих из элементов второго индекса, в данном случае – 19 массивов символов по 80 символов в каждом. Во-вторых, идентификатор двумерного массива с одним индексом интерпретируется как указатель на начало соответствующего массива элементов второго индекса, в данном случае – указатель на *i*-й массив из 80 символов (строку).

В еще одном промежуточном варианте статический массив указателей заполняется указателями на строки, которые создаются как динамические массивы.

```
//---- Ввод с клавиатуры ограниченного количества строк
char *pc[20], *p, c[80];
for (i=0; i<19; i++){
    gets(c);                                // Ввод строки
    if (strlen(c)==0) break;                // Пустая строка - конец
    pc[i] = new char[strlen(c)+1];         // Динамический массив
    strcpy(pc[i],c);                       // под строку
}
pc[i] = NULL;
```

В полностью динамической структуре данных массив указателей также создается в динамической памяти (см. ниже о динамическом массиве указателей на строки).

Дуализм двумерного массива и массива указателей на строки. Синтаксис операции извлечения символа из массива указателей на строки идентичен синтаксису двумерного массива символов. Первая индексация извлекает из массива *i*-й указатель, вторая извлекает *j*-й символ из строки, адресуемой указателем.

```
char    *p[]={“aaa”,“bbb”,“ccc”,NULL};
char    A[][20]={“aaa”,“bbb”,“ccc”};
```

```

p[i]           // Указатель на i-ю строку в массиве указателей
A[i]           // Указатель на начало i-й строки в двумерном массиве
p[i][j]        // j-й символ в i-й строке массива указателей
A[i][j]        // j-й символ в i-й строке двумерного массива

```

Отмеченное свойство означает единство логической организации двух структур данных, но физическая их реализация различна.

Динамический массив указателей на строки. Для массива указателей на строки типа `char*[]` существует аналог – двойной указатель типа `char**`, который интерпретируется как указатель на массив указателей на строки. Двойной указатель используется для работы с полностью динамической структурой данных. В последнем случае и сами строки, и массив указателей на них представлены динамическими массивами. В качестве примера рассмотрим создание массива указателей на строки при чтении строк из файла. Увеличение размерности динамического массива при его переполнении производится функцией перераспределения памяти **realloc**.

```

//-----35-05.cpp
//----- Создание ДМУ из строк файла
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#define SIZE0 10 // Кратность размерности ДМУ
char **loadfile(FILE *fd){
    char str[80];
    char ** pp = new char* [SIZE0]; // Создать динамический
    if (pp ==NULL) return(NULL);    // массив указателей
    for ( int i=0; fgets(str,80,fd) !=NULL; i++) {
        pp[i] = new char [strlen(str)+1]; // Создать динамический
        if (pp[i]==NULL) return NULL;    // массив символов и
        strcpy(pp[i],str);               // копировать туда строку
        if ((i+1) % SIZE0 ==0) {        // Расширить при переполнении
            pp = (char**) realloc( (void*) pp,sizeof(char *) *(i+1+SIZE0));
            if (pp ==NULL) return NULL;
        }
    }
    pp[i] = NULL; // Ограничитель массива указателей
    return pp; }

```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Динамический массив указателей переменной размерности. Единственная проблема динамического массива указателей – его фиксированная (хотя бы и динамическая) размерность, решается явным перераспределением памяти при его переполнении. Это происходит в том случае, когда программа не может «заранее определить» размерность хранимых данных, либо когда эта размерность меняется в широких пределах. Следующий фрагмент интегрирует последовательность вводимых строк в динамический мас-

сив указателей (сами строки также хранятся в динамических массивах). Размерность массива указателей m кратна N , то есть $m = N, 2N, 3N...$, при его заполнении создается новый массив указателей размерностью $m+N$, в который переписывается содержимое старого, после чего старый уничтожается.

```
//-----35-06.cpp
//--- Увеличение размерности динамического массива указателей
#include <stdio.h>
#include <string.h>
#define N 10
char **Load(){
char c[80]; int m=N;           // Первоначальная размерность
char **pp=new char*[N];       // Первоначальный массив указателей
for (int i=0; gets(c), c[0] != 0; i++){ // Ввод до пустой строки

    pp[i]=new char[strlen(c)+1]; // Создание динамического массива
    strcpy(pp[i], c);           // и копирование в него строки
    if ((i+1)%N==0){           // Заполнен последний указатель
        char **qq = new char* [i+1+N];
        for (int j=0; j<=i; j++) // Копировать указатели из
            qq[j]=pp[j];         // старого ДМУ в новый
        delete pp;              // Удалить старый ДМУ
        pp=qq;                  // Считать новый за старый
    }
}
pp[i]=NULL; return pp;} // Возвратить указатель на ДМУ
```

Используя функцию перераспределения памяти нижнего уровня **realloc**, можно упростить процедуру перераспределения:

```
pp=(char**)realloc((void*)pp, sizeof(char*)*(i+1+N));
```

Логическое упорядочение слов по длине. Требуется создать структуру данных, которая бы содержала информацию о словах в строке, упорядоченных по длине, и при этом не создавала бы копии слов, то есть ссылалась бы на их оригиналы. Логичное решение: динамический массив указателей необходимой размерности, в который записываются указатели на начальные символы строк, после чего массив сортируется. Для получения длины слова в исходной строке используется собственная функция.

```
//-----35-07.cpp
//--- Массив указателей на отсортированные по длине слова
#include <stdio.h>
int _strlen(char *p){
for (int i=0; *p!=0 && *p!=' '; p++, i++);
return i;}
char **SortedWords(char *p){
int nw=0, k; char *q;
for (q=p; *q!=0; q++) // Подсчет количества слов по концам слов
if (q[0]!=' ' && (q[1]==' ' || q[1]==0)) nw++;
char **qq=new char*[nw+1]; // Создать ДМУ на строки (символы строки)
nw=0;
```



```

if (*p!= ' ') qq[nw++]=p;    // Строка начинается со слова внести
for (p++; *p!=0; p++)       // Поиск по началу слова
if (p[0]!=' ' && p[-1]!=' ')
    qq[nw++]=p;
qq[nw]=NULL;
do {
    // Сортировка массива указателей
    k=0; // с использованием собственной функции
    for (int i=0; i<nw-1; i++)
        if (_strlen(qq[i])>_strlen(qq[i+1])){
            k++;
            char *g=qq[i]; qq[i]=qq[i+1]; qq[i+1]=g;
        }
    } while(k); return qq; }

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Используя программу – заготовку, содержащую функцию вывода меню для группы строк, заданной массивом указателей, реализовать функции редактирования файла. Для вывода на экран редактируемого текста использовать функцию с массивом указателей на строки редактируемого текста, для создания меню программы – статический массив указателей на строки меню. Строки редактируемого текста разместить в динамической памяти. В программе предусмотреть меню основных операций, в которое включить просмотр текста, добавление строки к тексту и действие над текстом в соответствии с вариантом задания 1–6:

1. Удаление, вставка, перемещение строки.
2. Отметка начала и конца блока, удаление, перемещение блока строк.
3. Редактирование выбранной строки: стирание и вставка символов.
4. Поиск по образцу в выделенном блоке и замена на другой образец.
5. Отметка начала и конца блока, копирование блока строк, запись блока в отдельный файл.
6. Чтение файла и включение текста после текущей строки.
7. Функция получает строку текста и возвращает динамический массив указателей на слова. Каждое слово копируется в отдельный массив в динамической памяти.
8. Функция получает массив указателей на строки и возвращает строку в динамической памяти, содержащую объединенный текст из входных строк.
9. Функция получает массив вещественных чисел. Размерность массива – формальный параметр. Функция создает динамический массив указателей на переменные в этом массиве и сортирует указатели (без перемещения указываемых переменных).

10. Функция получает на вход и возвращает в качестве результата динамический массив указателей на упорядоченные по алфавиту строки и включает в нее копию заданной строки с сохранением упорядоченности. Если после включения размерность массива становится кратной N ($k = N, 2N, 4N, \dots$), то создается новый массив указателей, размерность которого в два раза больше. В него переписываются указатели из старого массива, а старый разрушается.

11. Функция находит в строке заданную подстроку и возвращает динамический массив указателей на все вхождения этой подстроки.

12. Функция находит в строке фрагменты, содержащие последовательность одинаковых символов длиной более трех, и возвращает динамический массив указателей на такие фрагменты.

13. Функция находит в строке фрагменты, симметричные относительно центрального символа, длиной в семь символов и более (например, «**abcdcba**»), и возвращает динамический массив указателей на начала этих фрагментов.

14. Функция сравнивает по алфавиту два слова, ограниченные пробелом или концом строки. С ее использованием другая функция возвращает динамический массив указателей на слова во входной строке, отсортированные в алфавитном порядке (массив содержит указатели на слова в исходной строке).

15. Функция возвращает динамический массив указателей на слова во входной строке, отсортированные по длине (массив содержит указатели на копии слов из исходной строки).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Содержательно сформулируйте действие, производимое функцией над массивом указателей. Напишите вызов функции для статических данных.

Пример выполнения тестового задания

```
//-----35-08.cpp
double * F(double *p[], int k) {
    for ( int i=0; p[i]!=0; i++) ; // Текущая размерность массива указателей
    if (k>=i) return NULL;       // Больше текущей размерности - неудача
    double *q=p[k];              // Запомнить k-й указатель
    for (; k < i; k++)p[k] = p[k+1]; // Сдвинуть "хвост" на 1 к началу – удалить
    return q;                     // k-й и вернуть его
    double a1=4,a2=7,a3=5,a4=1,*pp[]={&a1,&a2,&a3,&a4,NULL};
    void main() { printf("\nУдален по n=2 ...%2.0lf\n",*F(pp,2));
    for (int i=0; pp[i]!=NULL;i++) printf(" %2.0lf",*pp[i]);
    }                             // Выведет 5 ... 4,7,1.
```

Функция возвращает указатель на **double**. Поскольку она получает массив указателей, можно предположить, что он берется оттуда. Действительно, из массива копируется указатель, номер которого задан формальным параметром. То есть функция возвращает указатель по заданному логическому номеру. Первоначально подсчитывается текущая размерность структуры данных – количество указателей в массиве. Если логический номер превышает его – возвращается **NULL**. И последнее. После запоминания **k**-го указателя все последующие указатели сдвигаются на 1 к началу, таким образом, выделенный указатель «затирается». То есть функция исключает указатель по логическому номеру и возвращает его в качестве результата. Для задания статической структуры данных сначала определяются указываемые переменные типа **double**, а затем массив указателей инициализируется их адресами.

```
//-----35-09.cpp
//----- 1
int F1(double *p[])
{ int n;
  for (n=0; p[n]!=NULL; n++);
  return n; }
//----- 2
void F2(double *p[])
{ int i,k;
  do {
    k=0;
    for (i=1; p[i]!=NULL; i++)
      if (*p[i-1] > *p[i])
        { double *dd;
          dd=p[i]; p[i]=p[i-1]; p[i-1]=dd; k++;
        }
  } while (k);}
//----- 3
void F3(double *p[], double *q)
{ int i,n;
  for (i=0; p[i]!=0; i++)
    if (*p[i] > *q) break;
  for (n=i; p[n]!=NULL; n++);
  for (; n >=i; n--) p[n+1] = p[n];
  p[i] = q;}
//----- 4
int F4(double **p[])
{ int k,i,j;
  for (k=i=0; p[i]!=NULL; i++)
    for (j=0; p[i][j] !=NULL; j++, k++);
  return k;}
//----- 5
char **F5(char a[][80], int n)
{ int i; char **p;
  p = new char* [n+1];
  for (i=0; i<n; i++) p[i]=a[i];
  p[n]=NULL;
```

```

return p;}
//----- 6
char *F6(char *p[])
{ int i,sz,l,k;
for (i=sz=k=0; p[i]!=NULL; i++)
    if ((l=strlen(p[i])) >sz) { sz=l; k=i; }
return(p[k]); }
//----- 7
char **F7(char c[])
{ char **p; int i,n, cnt;
p = new char* [20];
for (i=n=cnt=0; c[n]!=0; n++){
    if (c[n]==' '){
        { c[n]='\0'; cnt=0; }
    else
        { cnt++;
          if (cnt==1) p[i++]=&c[n];
          if (i==19) break;
        }
    }
}
p[i]=NULL; return(p);}
//----- 8
char *F8(char *p[], int m)
{ int n; char *q;
for (n=0; p[n]!=NULL; n++);
if (m >=n) return (NULL);
q = p[m];
for (n=m; p[n]!=NULL; n++) p[n]=p[n+1];
return q;}
//----- 9
int F9(char *p[], char *str)
{ int h,l,m;
for (h=0; p[h]!=NULL; h++){
    for (h--,l=0; h >= l;){
        m = (l+h) / 2;
        int k= strcmp(p[m],str);
        if (k==0) return m;
        if (k<0) l = m+1; else h = m-1;
    } return -1;}
}
//----- 10
char **F10()
{int n; char **p, s[80];
p = new char* [ 100 ];
for (n=0; n<99 & ( gets(s),s[0]!='\0'); n++ ){
    p[n]=new char[strlen(s)+1];
    strcpy(p[n],s); }
p[n]=NULL; return(p); }
//----- 11
void F11(char *p[], int m)
{ int n; char *q;
for (n=0; p[n]!=0; n++);
if (m >= n) return;
for (; n > m; n--) p[n+1] = p[n];
p[m+1] = new char [strlen(p[m]+1)];
strcpy(p[m+1],p[m]);}
//----- 12
char *F12(char **p[], int n)

```

```

{ int k,i,j;
for (k=i=0; p[i]!=NULL; i++)
    for (j=0; p[i][j] !=NULL; j++, k++)
        if (k==n) return(p[i][j]);
return(NULL);}
//----- 13
double F13(double *p[] , int n )
{ double s=0 ;
for (int i=0; p[i]!=NULL; i++)
    for (int j=0; j<n; j++) s+=p[i][j];
return s; }

```

3.6. СТРУКТУРЫ ДАННЫХ. ЛИНЕЙНЫЕ СПИСКИ

Щас по списку и пойдем...

М. Жванецкий. Ставь птицу

Списковая структура данных – множество переменных, связанных между собой указателями. Каждый элемент списковой структуры содержит один или несколько указателей на аналогичные элементы (элементы такого же типа). Прежде всего в программе необходимо определить тип данных – элемент списковой структуры.

```

struct elem          // Определение структурированного типа
{
    int value;        // Значение элемента
    elem *next;       // Единственный указатель или
    elem *next,*pred; // Два указателя или
    elem *links[10];  // Ограниченное количество указателей или
    elem **plinks;    // Произвольное количество указателей
};

```

Переменная такого типа может содержать один, два, не более 10 и произвольное (динамический массив) количество указателей на аналогичные переменные. Но отсюда никак нельзя определить ни количества таких переменных в структуре данных, ни характера связей между ними (последовательный, циклический, произвольный). Следовательно, конкретный тип структуры данных (линейный список, дерево, граф) зависит от функций, которые с ней работают. Значит, структура данных «защита» не столько в определении ее элементов, сколько в алгоритмах их обработки.

Списковые структуры данных обычно динамические. Этому есть две причины:

- сами переменные таких структур создаются как динамические переменные, то есть количество их может быть произвольным;
- количество связей между переменными и их характер также определяются динамически в процессе работы программы.

В программе списковые структуры данных доступны через указатель на некоторый ее элемент, который называется **заголовком**.

Линейный список представляет собой линейную последовательность переменных, каждая из которых связана указателями со своими соседями. Списки бывают следующих видов:

- односвязные – каждый элемент списка имеет указатель на следующий;
- двусвязные – каждый элемент списка имеет указатель на следующий и на предыдущий элементы;
- циклические – первый и последний элементы списка ссылаются друг на друга, и цепочка представляет собой кольцо (рис. 3.11).

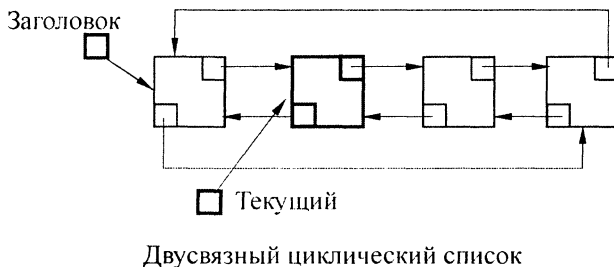
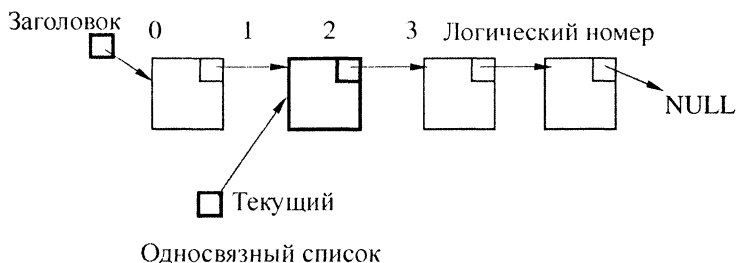


Рис. 3.11

Основное свойство линейных списков как структур данных: последовательность обхода списка зависит не от физического размещения элементов списка в памяти, а от последовательности их связывания указателями. Точно так же определяется нумерация элементов списка: **логический номер** элемента в списке – это номер, получаемый им в процессе обхода списка.

Списки – структуры данных с последовательным доступом. Работа со списками осуществляется исключительно через указатели. Каждый из них перемещается по списку (переустанавливается

с элемента на элемент), приобретая одну из смысловых интерпретаций – указатель на **первый, последний, текущий, предыдущий, новый** и иные элементы списка. Здесь уместна аналогия с массивом и индексом в нем, но при условии, что индекс меняется линейно, а не произвольно, а текущее количество заполненных элементов в массиве задано отдельной переменной.

Описания, действия	Список	Массив
Определение	<code>struct list {int val; list *next, *pred;};</code>	<code>int A[100]; int n;</code>
Пустая структура данных	<code>list *ph=NULL;</code>	<code>n=0;</code>
Первый	<code>list *p; p=ph;</code>	<code>int i=0;</code>
Следующий	<code>p->next</code>	<code>i+1</code>
Предыдущий	<code>p->pred</code>	<code>i-1</code>
К следующему	<code>p=p->next</code>	<code>i++</code>
К предыдущему	<code>p=p->pred</code>	<code>i--</code>
Просмотр	<code>for (p=ph; p!=NULL; p=p->next) ...p->val...</code>	<code>for (i=0; i<n; i++) ...A[i]...</code>
Последний	<code>p->next ==NULL</code>	<code>i==n-1</code>
К последнему	<code>for (p=ph; p->next!=NULL; p=p->next);</code>	<code>i=n-1</code>
Новый	<code>list *q = new list; q->val = v;</code>	<code>int v;</code>
Включить последним	<code>for (p=ph; p->next!=NULL; p=p->next); q->next=NULL; p->next=q;</code>	<code>A[n++]=v;</code>
Включить первым	<code>q->next=ph; ph=q;</code>	<code>for (i=n; i>0; i--) A[i]=A[i-1]; A[0]=v; n++;</code>

Способы формирования списков. *Статический список* представляет собой обычные переменные – элементы списка, связи между ними инициализируются транслятором, вся структура данных «зашивается» в программный код.

```
struct list { int val; list *next; } a={0,NULL}, b={1,&a}, c={2,&b}, *ph = &c;
```

Заметим, что по условиям определения переменных список создается «хвостом вперед».

Список может содержать ограниченное количество элементов, взятых из массива. Связи устанавливаются динамически, то есть программой. Такой вариант используется, когда фиксированное количество элементов образуют несколько различных динамических структур (например, очередей), в которых элементы списка переносятся из одной структуры в другую.

```
list    A[100],*ph;    // Создать список элементов,
for (i=0; i<99; i++) { // размещенных в статическом массиве
    A[i].next = A+i+1;
    A[i].val = i;
}
A[99].next = NULL;
ph = &A[0];
```

В *динамическом списке* элементы являются динамическими переменными, связи между ними устанавливаются программно.

```
list *ph=NULL;           // Список пустой
for (int i=0; i<10; i++){ // Создать список из 10 элементов,
    list *q=new list;     // включая очередной в начало
    q->val=i;              // списка
    q->next=ph;
    ph=q; }
```

Заголовок списка. В программе список обычно задается **заголовком** – указателем на первый элемент. Пустому списку соответствует **NULL**-указатель. Функция, работающая со списком, должна иметь обязательный параметр – заголовок списка.

```
//-----36-01.cpp
//----- Формальный параметр - заголовок списка
void F1(list *p) {
    for (; p!=NULL; p=p->next) puts(p->val);
}
```

Учитывая тот факт, что параметры в Си передаются по значению (в виде копии), этот вариант полезен только в том случае, когда первый (по счету) элемент списка остается таковым в процессе работы со списком. В противном случае необходимо изменение самого указателя (заголовка), которое может производиться:

- возвратом измененного значения заголовка в виде результата функции;
- передачей указателя на заголовок списка (указателя на указатель);
- передачей ссылки на заголовок. Напомним, что ссылка – неявный указатель, использующий при работе синтаксис объекта, который «отображается» на соответствующий ему фактический параметр.

```
//-----36-02.cpp
//---- Включение в начало списка с изменением заголовка
// Вариант 1. Измененный указатель возвращается
list *Ins1(list *ph, int v)
{ list *q=new list;
  q->val=v; q->next=ph; ph=q;
  return ph; }
```



```

//-----
// Вариант 2. Используется указатель на заголовок
void Ins2(list **pp, int v)
{ list *q=new list;
q->val=v; q->next=*pp; *pp=q; }
//-----
// Вариант 3. Используется ссылка на указатель
void Ins3(list *pp, int v)
{ list *q=new list;
q->val=v; q->next=pp; pp=q; }
//----- Пример вызова -----
void main(){
list *ph=NULL;           // Пустой список
ph=Ins1(ph,5);           // Сохранить новый заголовок
Ins2(&ph,66);            // Передается адрес заголовка
Ins3(ph,7);              // Передается ссылка на заголовок
}

```

Изменение порядка следования. Логический порядок следования элементов списка меняется путем переустановки указателей в элементах списка, что производится операциями присваивания указателей. Для их понимания пользуются несколькими содержательными интерпретациями.

1. Графическая интерпретация присваивания указателя:

– в левой части операции присваивания должно находиться обозначение ячейки, в которую заносится новое значение указателя, причем она может быть достижима только через имеющиеся рабочие указатели. На рис. 3.12 этому соответствует цепочка операций **q->pred->next=p;**

– в правой части операции присваивания должно находиться обозначение ячейки, из которой берется значение указателя, – на рис. 3.12 – **p**.

2. **Адресная интерпретация присваивания указателя** (рис. 3.13). Содержимым указателя является адрес указуемой переменной. В свете этой фразы предыдущая картинка может стать более понятной.

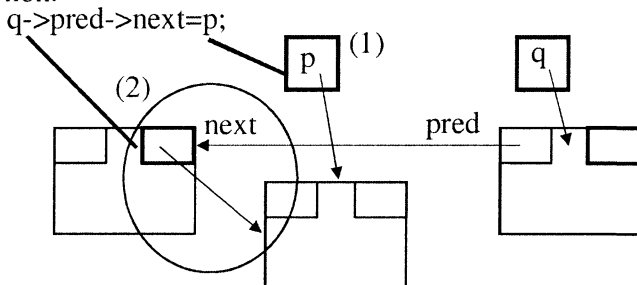


Рис. 3.12

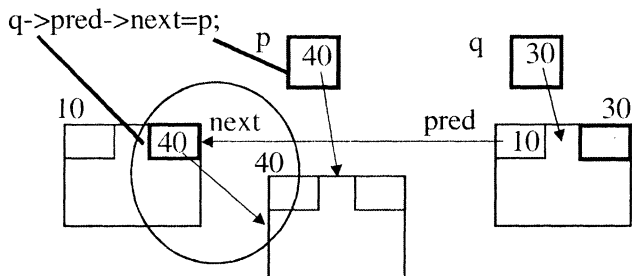


Рис. 3.13

3. Смысловая интерпретация присваивания указателя. При работе со списками каждый указатель имеет определенный смысл – первый, текущий, следующий, предыдущий и иные элементы списка. Поля **pred**, **next** также интерпретируются как указатели на следующий и предыдущий в элементе списка, доступном через указатель. Тогда смысл присваивания указателей однозначно переводится в словесное описание. Например, последовательность действий по включению нового элемента (указатель **q**) в двусвязный список перед текущим (указатель **p**) комментируется так:

```
q->next=p;           // Следующий для нового = текущий
q->pred=p->pred;      // Предыдущий для нового = предыдущий
if (p->pred == NULL) // текущего
    ph = q;          // Включение в начало списка
else                 // Включение в середину
    p->pred->next = q; // Следующий для предыдущего = новый
p->pred=q;           // Предыдущий для текущего = новый
```

Односвязный список. Простейший случай – элемент списка содержит единственный указатель на следующий, что позволяет двигаться по списку только в одном направлении. В ряде случаев включения и исключения элементов требуется сохранение указателя на предыдущий элемент. Например, для включения в список с сохранением порядка возрастания место включения нового элемента – перед первым, который больше вводимого при просмотре списка от начала. Это требует изменения значения указателя в предыдущем элементе списка.

```
//-----36-03.cpp
//--- Включение в односвязный с сохранением порядка
// pr - указатель на предыдущий элемент списка
void InsSort(list *&ph, int v)
{ list *q, *pr, *p;
  q=new list; q->val=v;
  // Перед переходом к следующему указатель на текущий
  // Запоминается как указатель на предыдущий
```

```

for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p,p=p->next);
if (pr==NULL) // Включение перед первым
    { q->next=ph; ph=q; }
else // Иначе после предыдущего
    { q->next=p; // Следующий для нового = текущий
      pr->next=q; } // Следующий для предыдущего = новый

```

Дополнительная проверка «крайних» ситуаций показывает, что фрагмент, производящий поиск места включения, корректно работает и в случае пустого списка (работает по ветке – включение перед первым).

Двусвязный список позволяет двигаться по цепочке элементов в обоих направлениях, имея доступными следующий и предыдущий элементы. «Расплачиваться» за это приходится увеличением количества операций над указателями.

```

//-----36-04.cpp
// Удаление элемента списка по заданному логическому номеру
void Del(list *amp, int n)
{ list *q; // Указатель на текущий элемент
  for (q = pp; q!=NULL && n!=0; q = q->next, n--); // Отсчитать n -ый
  if (q==NULL) return; // Нет элемента с таким номером
  if (q->pred==NULL) // Удаление первого -
    pp=q->next; // Коррекция заголовка
  else q->pred->next = q->next; // Следующий для предыдущего =
                               // Следующий за текущим
  if (q->next!=NULL) // Удаление не последнего -
    q->next->pred = q->pred; // Предыдущий для следующего =
                               // предыдущий текущего
  delete q; }

```

Циклический список позволяет моделировать линейные цепочки элементов, исключив постоянные проверки на «первый» и «последний». Особенности такого списка:

- поле **next** последнего элемента ссылается на первый элемент, а поле **pred** первого – на последний элемент списка;

- единственный элемент списка ссылается сам на себя (**q->next=q** и **q->pred=q**);

- операции включения элемента в начало и конец списка идентичны за исключением того, что в первом случае меняется заголовок.

Цикл просмотра такого списка предполагает возвращение указателя текущего элемента на начало списка в цикле с постусловием.

```

list *p=ph;
do
{
    // Тело цикла для текущего элемента – p
    p=p->next;
} while (p!=ph);

```

Все перечисленные особенности можно увидеть в примере включения нового элемента с сохранением упорядоченности.

```
//-----36-05.cpp
//--- Включение в циклический список с сохранением порядка
list *InsSort(list *ph, int v) // Функция возвращает новый заголовок
{ list *q = new list; // Новый элемент как единственный
  q->val = v; q->next = q->pred = q;
  if (ph == NULL) return q; // Список пуст – вернуть новый
  list *p = ph;
  do { if (v < p->val) break; // Место вставки перед первым,
        p->next; // большим заданного, иначе –
      } while (p!=ph); // перед первым в списке (после последнего)
  q->next = p; // Следующий за новым = текущий
  q->pred = p->pred; // Предыдущий для нового =
                    // предыдущий текущего
  p->pred->next = q; // Следующий для предыдущего = новый
  p->pred = q; // Предыдущий для текущего = новый
  if ( ph->val > v) ph=q; // Включение перед первым -
  return ph; } // коррекция заголовка
```

Поиск места включения завершается обнаружением первого элемента, больше заданного, либо возвращением на начало списка. В обоих случаях место вставки перед текущим элементом выбирается корректно: вставка перед первым есть вставка после последнего.

Представление стека и очереди односвязным списком. Стек можно смоделировать с помощью односвязного списка, реализуя операцию **POP** как исключение первого элемента, а **PUSH** – как включение в начало списка. Для моделирования очереди используются два указателя на первый и последний элементы списка (для прямого доступа к концу списка).

```
//-----36-11.cpp
// list *PH[2]; - заголовок очереди, [0]-первый, [1]-последний
void intoFIFO(list *ph[], int v)
{
  list *p= new list; // Поставить в конец очереди
  p->val = v; // Создать элемент списка;
  p->next = NULL; // и заполнить его
  if (ph[0] == NULL) // Новый элемент - последний
    ph[0] = ph[1] = p; // Включение в пустую очередь
  else { // Включение за последним элементом
    ph[1]->next = p; // Следующий за последним = новый
    ph[1] = p; // Последний = новый
  }
}
int fromFIFO(list *ph[]) // Извлечение из очереди
{
  if (ph[0] ==NULL) return -1; // Очередь пуста
  list *q = ph[0]; // Исключение первого элемента
  ph[0] = q->next;
  if (ph[0] ==NULL) ph[1] = NULL; // Элемент единственный
  int v = q->val;
  delete q; return v; }
```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Сортировка односвязного списка вставками. Особенностью сортировки списка является сохранение его элементов «на своих местах». В процессе сортировки элементы перемещаются из входного списка в выходной путем «переброски» указателей. В случае вставок внешний цикл поочередно выбирает элементы входного списка, а внутренний – включает их в выходной список с сохранением порядка. Заметим, что программа составлена достаточно формально из перечисленных операций со списками.

```
//-----36-06.cpp
//--- Сортировка односвязного списка вставками
// Функция возвращает заголовок нового списка
list *sort(list *ph)
{ list *q, *out, *p , *pr;
  out = NULL; // Выходной список пуст
  while (ph !=NULL) // Пока не пуст входной список
  { q = ph; ph = ph->next; // Исключить очередной
    // Поиск места включения
    for ( p=out,pr=NULL; p!=NULL && q->val>p->val; pr=p,p=p->next);
    if (pr==NULL) // Включение перед первым
    { q->next=out; out=q; }
    else // Иначе после предыдущего
    { q->next=p; pr->next=q; }
  }
  return out; }
```

Включение в двусвязный список с сохранением порядка. Программа адекватно реагирует на четыре ситуации: включение в пустой список, в начало, в конец и в середину списка.

```
//-----36-07.cpp
//-----Включение в двусвязный список с сохранением порядка
void InsSort(list * &ph, int v)
{ list *q , *p=new list; // Новый элемент списка
  p->val = v;
  p->pred = p->next = NULL;
  if (ph == NULL) { // Включение в пустой список
    ph=p; return ;
  } // Поиск места включения - q
  for (q=ph; q !=NULL && v > q->val; q=q->next) ;
  if (q == NULL) // Включение в конец списка
  { // Восстановить указатель на последний
    for (q=ph; q->next!=NULL; q=q->next) ;
    p->pred = q;
    q->next = p;
    return;
  } // Включить перед текущим
  p->next=q; // Следующий за новым = текущий
  p->pred=q->pred; // Предыдущий нового = предыдущий текущего
  if (q->pred == NULL) // Включение в начало списка
```

```

    ph = p;
else      // Включение в середину
    q->pred->next = p; // Следующий за предыдущим = новый
    q->pred=p; }      // Предыдущий текущего = новый

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

1. Сортировка двусвязного циклического списка вставками путем исключения первого элемента и включения в новый список с сохранением его упорядоченности.

2. Сортировка двусвязного списка путем исключения элемента с минимальным значением и включения его в начало нового списка.

3. Сортировка двусвязного циклического списка перестановкой соседних элементов.

4. Элемент односвязного списка содержит указатель на строку в динамической памяти. Написать функции просмотра списка и включения очередной строки с сохранением упорядоченности по длине строки и по алфавиту.

5. Элемент односвязного списка содержит массив из четырех целых переменных. Массив может быть заполнен частично. Все значения целых переменных хранятся в порядке возрастания. Написать функцию включения значения в элемент списка с сохранением упорядоченности. При переполнении массива создается новый элемент списка и в него включается половина значений из переполненного.

6. Элемент двусвязного циклического списка содержит указатель на строку в динамической памяти. Написать функции просмотра списка и включения очередной строки с сохранением упорядоченности по длине строки и по алфавиту.

7. Элемент двусвязного циклического списка содержит массив из четырех целых переменных. Массив может быть заполнен частично. Все значения целых переменных хранятся в порядке возрастания. Написать функцию включения значения в элемент списка с сохранением упорядоченности. При переполнении массива создается новый элемент списка и в него включается половина значений из переполненного.

8. Элемент двусвязного списка содержит указатель на строку. Вставить строку в конец списка. В список помещается копия входной строки в динамической памяти.

9. Элемент односвязного списка содержит указатель на строку. Строки упорядочены по возрастанию. Вставить строку в список с сохранением упорядоченности. В список помещается копия входной строки в динамической памяти.

10. Элемент односвязного списка содержит указатель на строку. Отсортировать список путем исключения максимального элемента и включения в начало нового списка.

11. Элемент двусвязного циклического списка содержит указатель на строку. Строки упорядочены по возрастанию. Вставить строку в список с сохранением упорядоченности. В список помещается копия входной строки в динамической памяти.

12. Элемент односвязного списка содержит массив указателей на строки. Строки читаются из текстового файла функцией **fgets**, и указатели на них помещаются в структуру данных. Элементы списка и сами строки должны создаваться в динамической памяти в процессе чтения файла. В исходном состоянии структура данных пуста.

13. Сортировка односвязного списка рекурсивным разделением. Функция разделяет список на две части относительно значения первого элемента и вызывает себя рекурсивно с полученными списками. Функция возвращает в качестве результата указатель на отсортированный список. Полученные от рекурсивного вызова списки «склеиваются» и возвращаются назад.

14. Сортировка односвязного списка простым однократным слиянием. Список разделяется на **n** частей, каждый сортируется независимо. Затем производится слияние в выходной список. Промежуточная структура данных – массив указателей на списки.

15. Сортировка односвязного списка циклическим слиянием.

16. Шейкер-сортировка двусвязного циклического списка. Используются указатели на границы отсортированных частей списка.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Пример оформления тестового задания

```
//-----36-08.cpp
#include <stdio.h>
struct list { int val; list *next ,*pred; };
extern list a,b,c; // Статический двусвязный список
list a={0, &b,NULL}, b={1,&c,&a}, c={2, NULL,&b}, *ph = &a;
//----- 0
// Включение в конец двусвязного списка
void F0(list **ph, int v) // ph - указатель на заголовок
{ list *p,*q = new list; // Создать новый элемент списка
q->val = v; q->next = q->pred = NULL; // По умолчанию - единственный
if (*ph == NULL) { // Список пуст - включить новый
*ph=p; return; }
for ( p=*ph ; p ->next !=NULL; p = p->next); // Найти последний
p ->next = q; q->pred=p;} // Новый - следующий за последним
```

```
//-----
void main(){ // Фактический параметр адрес заголовка
F0(&ph,5); F0(&ph,4);
// Просмотр списка в прямом и обратном направлениях
for (list *q=ph; q->next!=NULL; q=q->next) printf("%d ",q->val);
for (; q!=NULL; q=q->pred) printf("%d ",q->val);
}
```

Определите вид списка, «смысл» каждого указателя, выполняемое действие над списком, напишите вызов функции для статического списка.

```
//-----36-09.cpp
struct list { int val; list *next,*pred; };
//----- 1
int F1(list *p)
{ int n;
for (n=0; p!=NULL; p=p->next, n++);
return n; }
//----- 2
list *F2(list *ph, int v)
{ list *q = new list;
q->val = v; q->next = ph; ph = q;
return ph; }
//----- 3
list *F3(list *p, int n)
{ for (; n!=0 && p!=NULL; n--, p=p->next);
return p; }
//----- 4
list *F4(list *ph, int v)
{ list *p,*q = new list;
q->val = v; q->next = NULL;
if (ph == NULL) return q;
for ( p=ph ; p->next !=NULL; p = p->next);
p->next = q; return ph; }
//----- 5
list *F5(list *ph, int n)
{ list *q ,*pr,*p;
for ( p=ph,pr=NULL; n!=0 && p!=NULL; n--, pr=p, p =p->next);
if (p==NULL) return ph;
if (pr==NULL) { q=ph; ph=ph->next; }
else { q=p; pr->next=p->next; }
delete q;
return ph; }
//----- 6
int F6(list *p)
{ int n; list *q;
if (p==NULL) return 0;
for (q = p, p = p->next, n=1; p !=q; p=p->next, n++);
return n; }
//----- 7
list *F7(list *p, int v)
{ list *q;
q = new list;
q->val = v; q->next = q->pred = q;
if (p == NULL) p = q;
```



```

else
{ q->next = p; q->pred = p->pred;
  p->pred->next = q; p->pred = q; p=q;
}
return p; }
//----- 8
list *F8(list *ph)
{ list *q, *out, *p , *pr;
out = NULL;
while (ph !=NULL)
{ q = ph; ph = ph->next;
  for ( p=out,pr=NULL; p!=NULL && q->val>p->val; pr=p,p=p->next);
  if (pr==NULL)
  { q->next=out; out=q; }
  else
  { q->next=p; pr->next=q; }
} return out; }
//----- 9
list *F9(list *pp, int n)
{ list *q;
for (q = pp; n!=0; q = q->next, n--);
  if (q->next == q) { delete q; return NULL; }
if (q == pp) pp = q->next;
q->pred->next = q->next;
q->next->pred = q->pred;
delete q; return pp; }
//----- 10
void F10(list **p, int v)
{ list *q;
q = new list; q->val = v; q->next = *p; *p = q; }
//----- 11
list *F11(list **pp, int n)
{ list *q;
for (q = *pp; n!=0; q = q->next, n--);
  if (q->next == q) { *pp = NULL; return q; }
if (q == *pp) *pp = q->next;
q->pred->next = q->next;
q->next->pred = q->pred;
return q; }
//----- 12
list *F12(list *ph, int v)
{ list *q ,*pr,*p;
q=new list; q->val=v; q->next=NULL;
if (ph==NULL) return q;
for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p,p=p->next);
  if (pr==NULL) { q->next=ph; ph=q; }
  else { q->next=p; pr->next=q; }
return ph; }
//----- 13
list *F13(list *ph, int v)
{ list *q = new list;
q->val = v; q->next = q->pred = q;
if (ph == NULL) return q;
list *p = ph;
do {
  if (v < p->val) break;
  p=p->next;

```

```

    } while (p!=ph);
    q->next = p; q->pred = p->pred;
    p->pred->next = q; p->pred = q;
    if ( ph->val > v) ph=q;
    return ph; }

```

ГОЛОВОЛОМКИ, ЗАГАДКИ

Определите действие, выполняемое над списком. *Подсказки:* переменная **tmp** – заголовок временного списка, который возвращается функцией. Переменная **pp** – указатель на переменную (элемент), в которой находится указатель на текущий элемент списка, адресуемого **tmp**.

```

//-----36-10.cpp
list *F(list *ph)
{ list *q, *tmp, **pp;
  tmp = NULL;
  while (ph !=NULL)
  { q = ph; ph = ph->next;
    for (pp = &tmp; *pp !=NULL && (*pp)->val < q->val;
         pp = &(*pp)->next);
    q->next = *pp; *pp = q;
  }
  return tmp; }

```

3.7. СТРУКТУРЫ ДАННЫХ. ДЕРЕВЬЯ

По аналогии с рекурсивным вызовом функции существуют структуры данных, допускающие рекурсивное определение: элемент структуры данных содержит один или несколько указателей на аналогичные структуры данных. Формально это соответствует тому факту, что в определении структурированного типа содержатся указатели на структуры того же типа.

```

struct xxx {
    xxx *l,*r;           // Явно обозначенные указатели
xxx *pp[10];           // Статический массив указателей
xxx **pp;              // Динамический массив указателей
...};

```

Рекурсивные структуры данных и рекурсивные функции. Алгоритмы обработки рекурсивных структур данных по необходимости рекурсивны. Функция, обрабатывающая структуру данных, получает в качестве параметра указатель на ее элемент. Если этот элемент содержит корректные (не равные **NULL**) указатели на другие элементы и если алгоритм требует их просмотра, то данная функция вызывается рекурсивно с параметром – указателем на новый элемент. Проверку на **NULL** можно выполнить и в начале очередного шага рекурсии.

```

struct xxx{
    xxx *p[4];      // Массив указателей на элементы структур данных
    ...};
void F(xxx *q) {    // Функция обработки рекурсивной структуры данных
if (q==NULL) return; // Получает указатель на текущий элемент и
...                // рекурсивно вызывается для указателей
for (int i=0; i<4; i++) // на связанные с ним элементы
    F(q->p[i]);
}

```

Линейная рекурсия в списке. Односвязный список определяется как рекурсивная структура данных. Список – это либо пустой список, либо элемент списка, содержащий указатель на список. Любая циклическая программа, работающая со списком, преобразуется в рекурсивный эквивалент, получающий в качестве параметра указатель на очередной элемент списка (рис. 3.14).

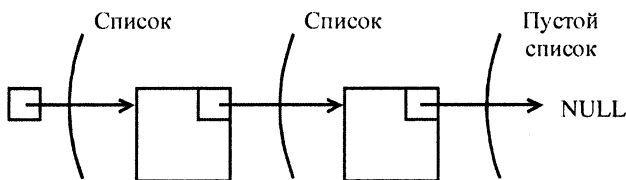


Рис. 3.14

```

//-----37-01.cpp
//---- Линейная рекурсия в списке
struct list { list *next; int val; };
//----- Просмотр списка -----
void scan(list *p) {
if (p == NULL) { puts(""); return; } // Указатель NULL - конец списка
printf("%d ", p->val);
scan(p->next);                      // Рекурсивный вызов для указателя
}                                   // на следующий элемент
//----- Включение в конец списка -----
// Функция получает ссылку на текущий элемент списка
void insert(list *&ph, int v) {
if (ph == NULL) { // Включить новый элемент под NULL-указатель
    ph=new list;
    ph->val=v;
    ph->next=NULL;
} // Рекурсивно передается ссылка на поле next
else insert(ph->next, v); } // текущего элемента
//----- Включение с сохранением порядка -----
// Возвращается измененный указатель на оставшуюся часть списка
list *insord(list *ph, int v) { // на место пустого или перед большим
if (ph==NULL || ph->val > v){
    list *pnew=new list;
    pnew->val=v;
    pnew->next=ph;
    return pnew;
}
ph->next=insord(ph->next,v);
return ph; } // Сохранить возможно измененный указатель

```

Деревья. Определение дерева имеет исключительно рекурсивную природу. Элемент этой структуры данных называется вершиной. Дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число связей с другими деревьями (ветвей). Ниже лежащие деревья для текущей вершины называются поддеревьями, а их вершины – потомками. По отношению к потомкам текущая вершина называется предком (рис. 3.15). Вершина дерева – структурированная переменная, содержащая некоторое количество (отдельные переменные, массив, динамический массив) указателей на потомков.

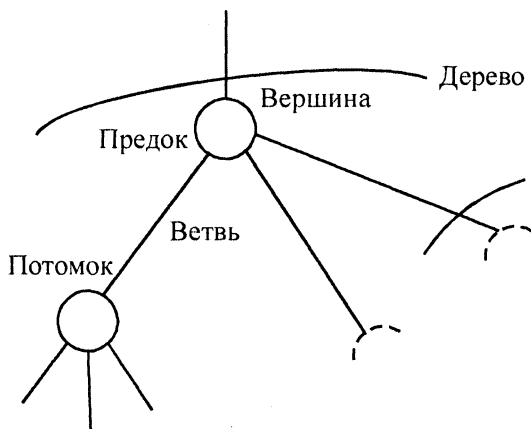


Рис. 3.15

```
struct tree{
    int val;           // Значение в вершине дерева
    tree *p[4];        // Массив указателей на потомков
};                   // (ограниченное число)
```

Обход дерева. Рекурсивная функция обхода дерева получает в качестве формального параметра указатель на текущую вершину, в теле функции присутствует цикл, в котором производится рекурсивный вызов с параметром – указателем на потомка. Обход ограничивается обнаружением **NULL**-указателя – отсутствием потомка (рис. 3.16).

```
void Scan(xxx *q) {           // Функция обхода дерева
    if (q==NULL) return;      // Потомок отсутствует
    printf("%d\n",q->val);
    for (int i=0; i<4; i++)    // Цикл обхода потомков
        Scan(q->p[i]);        // Рекурсивный вызов для потомков
}
```

Сравнение со списками и с массивами (массивами указателей). Достоинство списка – локальность производимых в нем изменений – при включении/исключении элемента затрагиваются только его соседи, да и то их расположение в памяти не меняется. Массивы (массивы указателей), напротив, требуют в этом случае массового перемещения элементов. Основной порок списка – исключительно последовательный доступ. Древовидная структура данных обеспечивает известный компромисс: изменения в нем обладают свойством локаль-

ности, а доступ хотя и не прямой, но по крайней мере логарифмический (при замене алгоритмов полного обхода дерева выбором одной из его ветвей) (рис. 3.17).

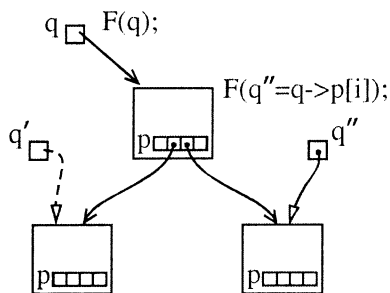


Рис. 3.16

ности, а доступ хотя и не прямой, но по крайней мере логарифмический (при замене алгоритмов полного обхода дерева выбором одной из его ветвей) (рис. 3.17).

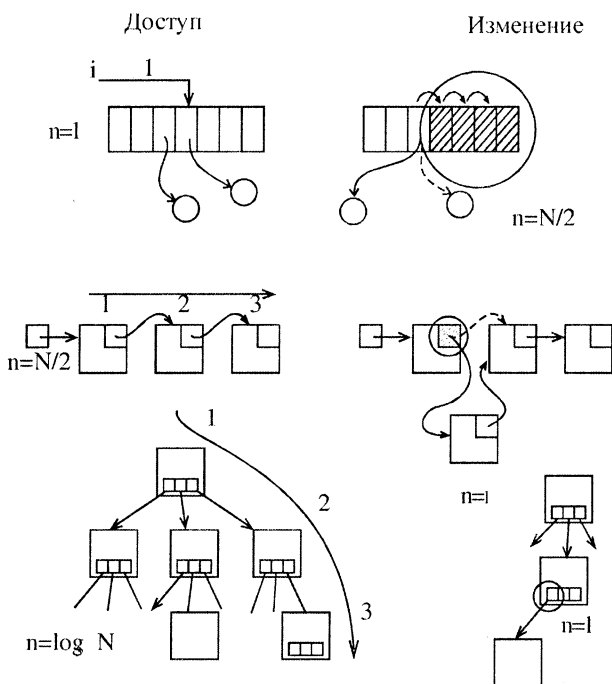


Рис. 3.17

Определение глубины дерева. При определении минимальной (максимальной) длины ветви дерева каждая вершина должна получить значения минимальных длин ветвей от потомков, выбрать из них наименьшую и передать предку результат, увеличив его на 1 – «добавить себя».

```
struct tree{
    int val;
    tree *p[4];
};
//-----37-02.cpp
//---- Определение ветви минимальной длины
int MinLnt(tree *q){
    if (q==NULL) return 0;
    int min= MinLnt(q->p[0]);
    for (int i=1; i<4; i++){
        int x=MinLnt(q->p[i]);
        if (x < min) min=x;}
    return min+1;}

```

Обход дерева на заданную глубину. Для отслеживания процесса «погружения» достаточно дополнительной переменной, которая уменьшает свое значение на 1 при очередном рекурсивном вызове.

```
//-----37-03.cpp
//-----Включение вершины в дерево на заданную глубину
int Insert(tree *ph, int v, int d) { // d - текущая глубина включения
    if (d == 0) return 0;          // Ниже не просматривать
    for ( int i=0; i<4; i++)
        if (ph->p[i] == NULL){
            tree *pn=new tree;
            pn->val=v;
            for (int j=0; j<4; j++) pn ->p[j]=NULL;
            ph->p[i]=pn;
            return 1; }
    else
        if (Insert(ph->p[i], v , d-1)) return 1;    // Вершина включена
    return 0; }

```

Для включения простейшим способом нового значения в дерево в ближайшее к корню место достаточно соединить две указанные функции вместе.

```
void main(){ tree PH={1,{NULL,NULL,NULL,NULL}};
for (int i=0; i<100; i++) Insert(&PH, rand(20), MinLnt(&PH));
}
```

Поиск первого подходящего в дереве на основе полного обхода. При обнаружении в дереве первого значения, удовлетворяющего условию, необходимо прервать не только текущий шаг рекурсии, но и все предыдущие. Поэтому цикл рекурсивного вызова прерывается сразу же, как только потомок возвратит найденное

в поддереве значение. Текущая вершина должна «ретранслировать» полученное от потомка значение к собственному предку («вверх по инстанции»).

```
//-----37-04.cpp
//---- Поиск в дереве строки, длиной больше заданной
struct stree{
char *str;
stree *p[4];};
char *big_str(stree *q){
if (q==NULL) return NULL;
if (strlen(q->str)>5) return q->str;    // Найдена в текущей вершине
    for (int i=0; i<4; i++){
        char *child=big_str(q->p[i]);    // Получение строки от потомка
        if (child!=NULL) return child;    // Вернуть ее " от себя лично"
    }
return NULL;};                        // Нет ни у себя, ни у потомков
```

Поиск в дереве максимального (минимального) значения.
Производится полный обход дерева, в каждой вершине – стандартный контекст выбора минимального из текущего значения в вершине и значений, полученных от потомков при рекурсивном вызове функции.

```
//-----37-05.cpp
//---- Поиск максимального в дереве
int GetMax(tree *q){
if (q==NULL) return -1;
int max=q->val;
    for (int i=0; i<4; i++){
        int x=GetMax(q->p[i]);
        if (x > max) max=x;
    }
return max;}
```

Оптимизация поиска в дереве. Основное свойство дерева соответствует пословице «дальше в лес – больше дров». Точнее, количество просматриваемых вершин от уровня к уровню растет в геометрической прогрессии. Если известен некоторый критерий частоты использования различных элементов данных (например, более короткие строки используются чаще, чем длинные), то в соответствии с ним можно частично упорядочить данные по этому критерию с точки зрения их «близости» к корню: в нашем примере в любом поддереве самая короткая строка находится в его корневой вершине. Алгоритм поиска может ограничить глубину просмотра такого дерева.

```
//-----37-06.cpp
//--- Дерево оптимизацией : короткие ключи ближе к началу
struct dtree{
char *key;                                // Ключевое слово
void *data;                               // Искомая информация
dtree *p[4]; };                          // Потомки
```

```

void *find(dtree *q, char *keystр)    // Поиск по ключу
{ void *s;
if (q==NULL) return NULL;
if (strcmp(q->key,keystр)==0) return q->data; // Ключ найден
if (strlen(keystр)<strlen(q->key))
    return NULL; // Короткие строки - ближе к корню
for (int i=0; i<4; i++)
    if ((s=find(q->p[i],keystр))!=NULL) return s;
return NULL; }

```

Функция включения в такое дерево ради сохранения свойств должна в каждой проходимой ею вершине рекурсивно «вытеснять» более длинную строку в поддереву и заменять ее на текущую (новую), более короткую.

Нумерация вершин. Способы обхода дерева. В массивах и списках каждый элемент имеет свой логический номер в линейной последовательности, соответствующей их размещению в памяти (массив) или направлению последовательного обхода (списки). В деревьях обход вершин возможен только с использованием рекурсии, поэтому и их логическая нумерация производится согласно последовательности их рекурсивного обхода. Рекурсивная функция в этом случае получает ссылку или указатель на счетчик вершин, который она увеличивает на 1 при обходе текущей вершины. В зависимости от того, кто нумеруется раньше – предок или потомки, имеют место различные способы обхода и нумерации (рис. 3.18).

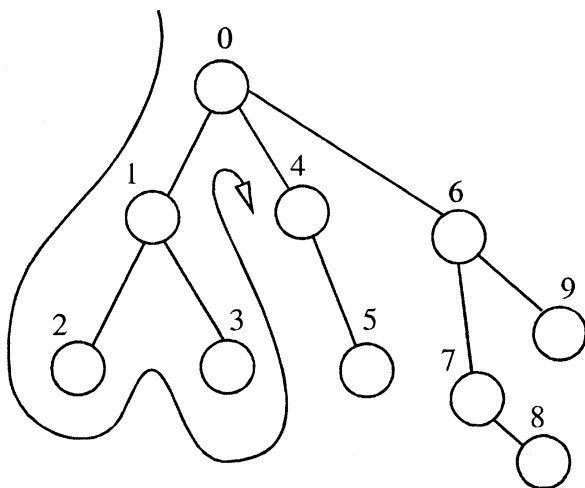


Рис. 3.18


```
//-----37-07.cpp
//---- Обход дерева с нумерацией вершин сверху вниз
void ScanNum (tree *q, int &n ) {
    if (q==NULL) return;
    printf("n=%d val=%d\n",n++,q->val);
    for (int i=0; i<4; i++) ScanNum(q->p[i],n);
}
```

Обход с нумерацией в обычном дереве используется для извлечения вершины по логическому номеру. При достижении вершины с заданным номером обход прекращается (аналогично алгоритму поиска первого подходящего).

```
//-----37-08.cpp
//---- Извлечение по логическому номеру с полным обходом дерева
int GetNum (tree *q, int &n, int num ) {
    if (q==NULL) return -1;
    if (n++ ==num) return q->val;    // Номер текущей совпал с требуемым
    for (int i=0; i<4; i++) {        // Обход потомков,
        int vv=GetNum (q->p[i],n,num ); // пока не превышен номер
        if (n > num) return vv;
    }
    return -1; }
}
```

Если каждая вершина дерева будет содержать дополнительный параметр – количество вершин в связанном с ней поддереве, то извлечение по логическому номеру выполняется с помощью циклического алгоритма либо линейной рекурсии, благодаря тому, что можно сразу же определить, в каком поддереве находится интересующая нас вершина. Счетчики вершин можно корректировать в самом процессе добавления/удаления вершин.

```
//-----37-09.cpp
//---- Извлечение по логическому номеру (счетчик вершин в поддереве)
struct ctree{
    int nodes;           // Счетчик вершин в поддереве
    int val;
    ctree *p[4]; };
int GetNum(ctree *q, int num, int n0){
    if (q==NULL) return 1;    // n0 начальный номер в текущем поддереве
    if (n0++==num) return q->val; // Начальный номер совпал с требуемым
    for (int i=0; i<4; i++){
        if (q->p[i]==NULL) continue;
        int nc= q->p[i]->nodes; // Число вершин у потомка
        if (n0+ nc > num)      // Выбран потомок
            return GetNum(q->p[i],num,n0); // с диапазоном номеров
        else                  // Корректировать начальный номер
            n0+=nc;           // для следующего потомка
    }
}
```

Двоичное дерево. В двоичном дереве каждая вершина имеет не более двух потомков, обозначенных как левый (**left**) и правый (**right**). Кроме того, на данные, хранимые в вершинах дерева, вво-

дится следующее правило упорядочения: значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева – больше значения в текущей вершине (рис. 3.19).

```
struct    btree {
    int    val;
    btree *left,*right; };
```

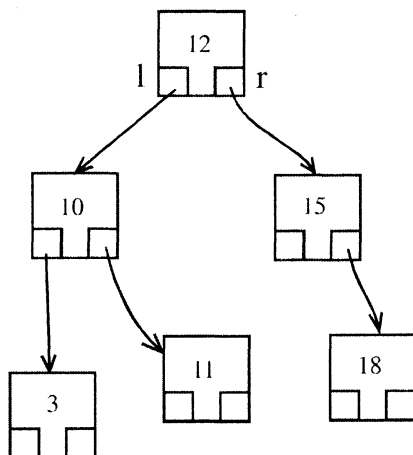


Рис. 3.19

Поиск и включение в двоичное дерево. Свойства двоичного дерева позволяют применить в нем алгоритм поиска, аналогичный двоичному поиску в массиве. Каждое сравнение искомого значения и значения в вершине позволяет выбрать для следующего шага правое или левое поддерево. Алгоритмы включения и исключения вершин дерева не должны нарушать указанное свойство: при включении вершины дерева поиск места ее размещения производится путем аналогичных сравнений. Эти алгоритмы линейно рекурсивные или циклические.

```
//-----37-10.cpp
//---- Обход двоичного дерева
void Scan(btree *p, int level){
if (p==NULL) return;
Scan(p->left,level+1);
printf("l=%d val=%d\n",level,p->val);
Scan(p->right,level+1); }
//---- Поиск в двоичном дереве-----
// Возвращается указатель на найденную вершину
btree *Search(btree *p, int v){
if (p==NULL) return NULL;
if (p->val == v) return p;
if (p->val > v)
// Ветка пустая
// Вершина найдена
// Сравнение с текущим
```

```

        return Search(p->left,v);           // Левое поддерево
    else
        return Search(p->right,v); }        // Правое поддерево
//----- Включение значения в двоичное дерево-----
// Используется ссылка на указатель на текущую вершину
void Insert(btree *&pp, int v){
    if (pp == NULL) {                      // Найдена свободная ветка
        pp = new btree;                   // Создать вершину дерева
        pp->val = v;
        pp->left = pp->right = NULL;
        return;
    }
    if (pp->val > v)                         // Перейти в левое или
        Insert(pp->left,v);                // правое поддерево
    else
        Insert(pp->right,v);
}

```

Обратите внимание, что указатель **pp** ссылается на то место в дереве, где находится указатель на текущую вершину, а потому под указатель можно производить запись (присваивание) при создании новой вершины. При замене рекурсии циклом пришлось бы довольствоваться явным двойным указателем **btree **pp**.

Нумерация вершин в двоичном дереве. В двоичном дереве естественная нумерация вершин соответствует обходу в порядке возрастания их значений, то есть левое поддерево – текущая вершина – правое поддерево.

```

//-----37-11.cpp
//---- Обход двоичного дерева с нумерацией вершин
void ScanNum ( btree *q, int &n ) {
    if (q==NULL) return;
    ScanNum(q->left,n);
    printf("n=%d val=%d\n",n,q->val); n++;
    ScanNum(q->right,n);}

```

Свойства двоичного дерева. Сбалансированность. Поиск в двоичном дереве требует количества сравнений, не превышающего максимальной длины ветви дерева. Условием эффективности поиска в дереве является равенство длин его ветвей (сбалансированность). В наихудшем случае дерево имеет одну ветвь и вырождается в односвязный список, в котором имеет место последовательный (линейный) поиск. В идеальном случае, когда длины ветвей дерева отличаются не более чем на 1 (сбалансированное дерево) и равны **n** или **n-1**, при общем количестве вершин в дереве порядка **2ⁿ** требуется не более **n** сравнений для нахождения требуемой вершины. Это соответствует характеристикам алгоритма двоичного поиска в упорядоченном массиве.

Поддержание сбалансированности при операциях включения/исключения является довольно трудной задачей [1].

Структуры данных с произвольными связями. Граф представляет собой структуру с произвольным характером связей между элементами. С точки зрения программирования наличие в элементе **А** указателя на элемент **В** соответствует наличию в графе дуги, направленной от **А** к **В**. Тогда для неориентированного графа требуется наличие как прямого, так и обратного указателя. Алгоритмы работы с графом также основаны на его рекурсивном обходе. Однако при этом необходимо отмечать уже пройденные вершины для исключения «зацикливания». Для этого достаточно в каждой вершине иметь счетчик обходов, который проверяется каждый раз при входе в вершину.

```
//-----37-12.cpp
//--- Рекурсивный обход графа
struct graph {
    int cnt,val; // Счетчик обходов вершин
    graph **pl; // Динамический массив указателей
    void ScanGraph(graph *p){
        if (p==NULL) return;
        printf("val=%d\n",p->val);
        p->cnt++; // Увеличить счетчик в текущей вершине
        for ( int i=0; p->pl[i] !=NULL; i++) {
            if (p->pl[i]->cnt !=p->cnt) // Вершина не просмотрена
                ScanGraph(p->pl[i]); // Рекурсивный обход
        }
    }
}
```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Включение в дерево по логическому номеру. Вершина дерева содержит два указателя на левое и правое поддеревья и значение. Данные в дереве не упорядочены (дерево не двоичное). Требуется включить новое значение под заданным логическим номером.

За основу возьмем алгоритм полного обхода дерева с нумерацией вершин. Когда будет достигнута вершина с заданным логическим номером, произойдет включение, и обход прекратится. Для этого необходимо выполнить изображенные на рис. 3.20 преобразования, сохраняющие логическую нумерацию вершин во всем остальном дереве. Здесь учитывается тот факт, что относительно текущей вершины нумерация продолжается в левом поддереве. Новое значение включается в текущую вершину. Значение из текущей вершины:

- при отсутствии левого поддерева включается как новая вершина в левое поддерево;
- при отсутствии правого поддерева включается в левое поддерево, а левое поддерево переносится в правое;

– включается иначе – как новая вершина «в разрыв» между текущей вершиной и левым поддеревом.

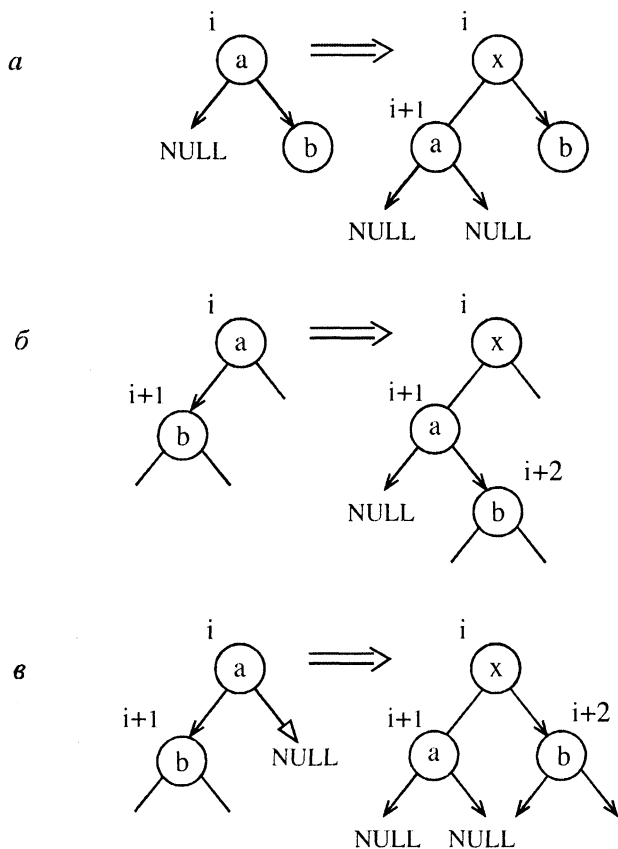


Рис. 3.20

```
//-----37-13.cpp
//---- Включение в дерево по логическому номеру
struct tree2{
    int val;
    tree2 *left,*right; };
int InsertNum ( tree2 *q, int &n, int vv ) {
    if (q==NULL) return 0;
    if (InsertNum(q->left,n,vv))
        return 1; // Если включено в левое при обходе
    if (n-- ==0){ // Включение в текущую вершину
        tree2 *pn=new tree2;
        pn->val = vv; // Новое значение в новую вершину
        pn->left=pn->right=NULL;
        if (q->left==NULL)
```

```

        q->left=pn;          // 1 в свободное левое поддереву
    else
    if (q->right==NULL){     // 2 в правое поддереву с переносом
        pn->val=q->val;      // текущей вершины
        q->val=vv;
        q->right=pn;}
    else {                  // 3 " в разрыв" левого поддерева
        pn->left=q->left;
        pn->val=q->val;
        q->val=vv;
        q->left=pn;
    }
    return 1; }
return InsertNum(q->right,n,vv); // Попытаться в правом
}

```

Представление двоичного дерева в массиве. Двоичное дерево естественным образом располагается в массиве. Если текущая вершина имеет в нем индекс **n**, то левый и правый потомки – **2*n** и **2*n+1** соответственно. Корень дерева имеет **n=1**.

```

//-----37-14.cpp
//--- Двоичное дерево в динамическом массиве
void scan(int v[],int n,int sz){
    if (n>=sz || v[n]==-1) return;
    scan(v,2*n,sz);
    printf("%d\n",v[n]);
    scan(v,2*n+1,sz); }
void insert(int *v, int &sz, int n, int val)
{ if (n>=sz){          // Удвоить размерность динамического массива
    v=(int*)realloc((void*)v, 2*sz*sizeof(int));
    for (int i=sz; i<2*sz; i++) v[i]=-1;
    sz*=2;             // Отметить новые вершины как свободные
}
if (v[n]==-1){         // Вершина свободна
    v[n]=val; return;
}
if (val<v[n]) insert(v,sz,2*n,val);
else insert(v,sz,2*n+1,val);
}

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Программа должна содержать функцию обхода дерева с выводом его содержимого, функцию добавления вершины дерева (ввод), а также указанную в варианте функцию.

1. Вершина дерева содержит указатель на строку. Строки в дереве не упорядочены. Функция включает вершину в дерево с новой строкой в ближайшее свободное к корню дерева место (в результате дерево будет сбалансированным). Для исключения полного обхода в каждую вершину дерева поместить длину его минимальной

ветви и корректировать его в процессе включения во всех проходимых вершинах.

2. Вершина двоичного дерева содержит массив целых и два указателя на правое и левое поддеревья. Массив целых в каждом элементе упорядочен, дерево в целом также упорядочено. Функция включает в дерево целую переменную с сохранением упорядоченности (рис. 3.21).

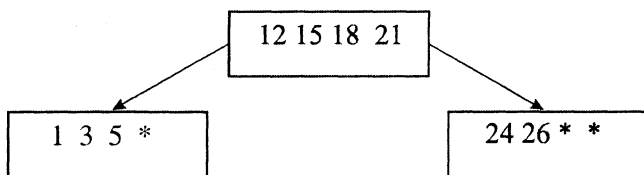


Рис. 3.21

3. Вершина двоичного дерева содержит указатель на строку и указатели на правое и левое поддеревья. Строки в дереве упорядочены по возрастанию. Написать функции включения строки и получения указателя на строку по заданному номеру, который строка имеет в упорядоченной последовательности обхода дерева.

4. Элемент дерева содержит либо данные (строка ограниченной длины), либо указатели на правое и левое поддеревья. Строки в дереве упорядочены. Написать функцию включения новой строки. Обратит внимание на то, что элемент с указателями не содержит данных, и при включении новой вершины вершину с данными следует заменить на вершину с указателями (рис. 3.22).

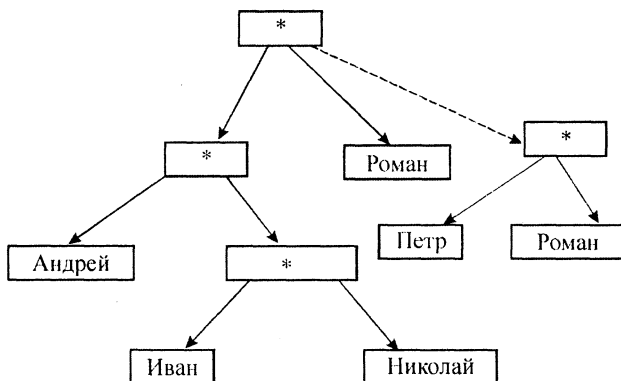


Рис. 3.22

5. Вершина дерева содержит целое число и массив указателей на поддеревья. Целые в дереве не упорядочены. Функция включает вершину в дерево с новой целой переменной в ближайшее свободное к корню дерева место, то есть дерево должно иметь ветви, отличающиеся не более чем на 1 (рис. 3.23).

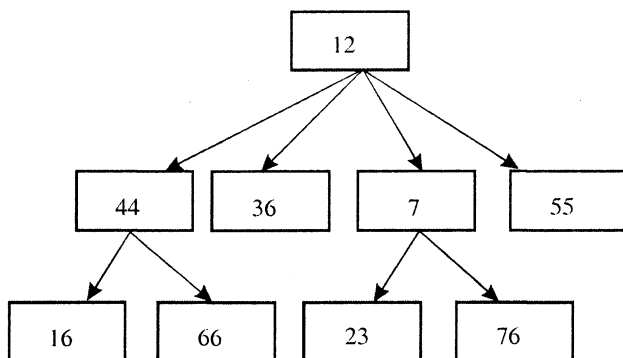


Рис. 3.23

6. Вершина дерева содержит два целых числа и три указателя на поддеревья. Данные в дереве упорядочены. Написать функцию включения нового значения в дерево с сохранением упорядоченности (рис. 3.24).

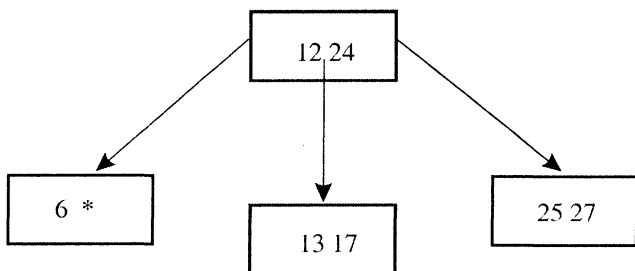


Рис. 3.24

7. Вершина дерева содержит указатель на строку и N указателей на потомков. Функция помещает строки в дерево так, что строки с меньшей длиной располагаются ближе к корню. Если новая строка «проходит» через вершину, в которой находится более длинная строка, то новая занимает место старой, а алгоритм включения продолжается для старой строки. Функция включения выбирает потомка с минимальным количеством вершин в поддереве.

8. Вершина дерева содержит либо четыре целых значения, либо два указателя на потомков, причем концевые вершины содержат данные, а промежуточные – указатели на потомков. Естественная нумерация значений производится при обходе концевых вершин слева направо. Разработать функции получения и включения значения в дерево по логическому номеру (рис. 3.25).

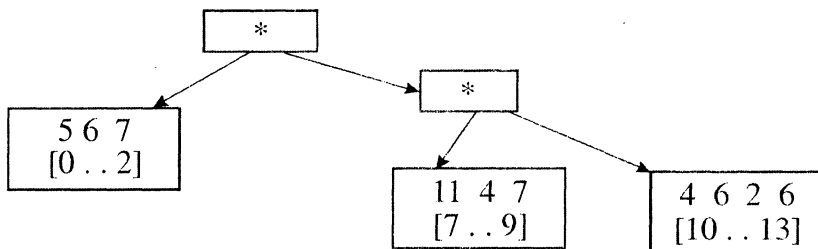


Рис. 3.25

9. Двоичное дерево представлено в массиве «естественным образом»: если вершина-предок имеет номер (индекс) n , то потомки – соответственно $2*n$ и $2*n+1$. Нумерация начинается с $n=1$. Ячейка со значением 0 (или NULL) обозначает отсутствие вершины. Разработать функцию сортировки строк с использованием способа представления такого дерева в массиве указателей на строки.

10. Вершина дерева содержит N целых значений и два указателя на потомков. Запись значений производится таким образом, что меньшие значения оказываются ближе к корню дерева (то есть все значения в поддеревьях больше самого большого значения у предка). Разработать функции включения и поиска данных в таком дереве. Если новое значение «проходит» через вершину, в которой находится большее, то оно замещает большее значение, а для последнего – алгоритм продолжается. Функция включения выбирает потомка с максимальным значением в поддереве.

11. Выражение, содержащее целые константы, арифметические операции и скобки, может быть представлено в виде двоичного дерева. Концевая вершина дерева должна содержать значение константы, промежуточная – код операции и указатели на правый и левый операнды – вершины дерева. Функция получает строку, содержащую выражение, и строит по ней дерево. Другая функция производит вычисления по полученному дереву.

12. Вершина дерева содержит указатель на строку и динамический массив указателей на потомков. Размерность динамического

массива в корневой вершине – N, на каждом следующем уровне – в два раза больше. Функция при включении строки создает вершину, наиболее близкую к корню.

13. Вершина дерева содержит динамический массив целых значений и два указателя на потомков. Значения в дереве не упорядочены и не нумеруются. Размерность динамического массива в корневой вершине – N, на каждом следующем уровне – в два раза больше. Функция включает новое значение в свободное место в массиве ближайшей к корню вершины.

14. Вершина дерева содержит массив целых и два указателя на правое и левое поддеревья. Значения в дереве не упорядочены. Естественная нумерация значений производится путем обхода дерева по принципу «левое поддерево – вершина – правое поддерево». Разработать функции включения и получения значения элемента по заданному логическому номеру.

15. Код Хаффмана, учитывающий частоты появления символов, строится следующим образом. Для каждого символа подсчитывается частота его появления и создается вершина двоичного дерева. Затем из множества вершин выбираются две с минимальными частотами появления и создается новая – с суммарной частотой, к которой выбранные подключаются как правое и левое поддерева. Созданная вершина включается в исходное множество, а выбранные – удаляются. Затем процесс повторяется до тех пор, пока не останется единственная вершина. Код каждого символа – это последовательность движения к его вершине от корня (левое поддерево – 0, правое – 1). Функция строит код Хаффмана для символов заданной строки.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определите вид дерева и выполняемое над ним действие. Напишите вызов функции для статического дерева, составленного из инициализированных переменных.

Пример оформления тестового задания

```
//-----37-15.cpp
struct tree { char *s; tree *p[4]; };
int F( tree *q) {
    if (q==NULL) return 0;
    for (int v=strlen(q->s), i=0; i<4; i++)
        v+=F(q->p[i]);
    return v; }
```

То, что речь идет о дереве, подтверждается наличием рекурсии для указателей на «соседние» элементы структуры данных. Вершина дерева содержит указатель на строку. В каждой вершине производится суммирование длины содержащейся в ней строки и результатов рекурсивного вызова потомков, очевидно, тоже суммарных длин строк, находящихся в поддеревьях. Итог: функция возвращает суммарную длину строк в вершинах дерева.

```
tree A1={"aaa",NULL,NULL,NULL,NULL};
tree A2={"bb",NULL,NULL,NULL,NULL};
tree A3={"cccc",&A1,&A2,NULL,NULL};
tree A4={"dd",NULL,NULL,NULL,NULL};
tree A5={"aaa",NULL,NULL,NULL,NULL};
tree A6={"fff",&A3,&A4,&A5,NULL};
void main()
{ printf("F=%d\n",F(&A6)); }    // Вызов для статического дерева

//-----37-16.cpp
//-----1
struct xxx { int v; xxx *p[4]; };
int F1(xxx *q)
{ int i,n,m;
if (q==NULL) return 0;
for (n=F1(q->p[0]),i=1; i<4; i++)
    if ((m=F1(q->p[i])) >n) n=m;
return n+1; }
//-----2
struct zzz { int v; zzz *l,*r; };
int F2(zzz *p) {
if (p==NULL) return(0);
return (1 + F2(p->r) + F2(p->l)); }
//-----3
int F3(xxx *q)
{ int i,n,m;
if (q==NULL) return 0;
for (n=q->v,i=0; i<4; i++)
    if ((m=F3(q->p[i])) >n) n=m;
return n; }
//-----4
void F4(int a[], int n, int v) {
if (a[n]==-1) { a[n]=v; return; }
if (a[n]==v) return;
if (a[n]>v) F4(a,2*n,v);
else F4(a,2*n+1,v);
}
void z3() {
int B[256],i;
for (i=0; i<256; i++) B[i]=-1; F4(B,1,5); F4(B,1,3); }
//-----5
int F5(xxx *q)
{ int i,n;
if (q==NULL) return 0;
for (n=q->v,i=0; i<4; i++)
    n+=F5(q->p[i]);
return n; }
```

```

//-----6
struct yyy { int k; int v[10]; yyy *l,*r; };
int F6(yyy *q)
{ int i,n;
if (q==NULL) return 0;
for (n=0,i=0; i<q->k; i++) n+=q->v[i];
return n+F6(q->l)+F6(q->r); }
//-----7
int F7(yyy *q)
{ int i,n,m;
if (q==NULL) return 0;
for (n=1,i=0; i<4; i++)
    n+=F7(q->p[i]);
return n; }
//-----8
int F8(yyy *p) {
if (p==NULL) return(0);
int nr=F8(p->r)+1;
int nl=F8(p->l)+1;
return nr>nl ? nr : nl; }
//-----9
int F9(yyy *q)
{ int i,n,m;
if (q==NULL) return -1;
if (q->v >=0) return q->v;
for (i=0; i<4; i++)
    if ((m=F9(q->p[i]))!=-1) return m;
return -1; }

```

3.8. ИЕРАРХИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Лучшее – враг хорошего.

Поговорка

Иерархические структуры данных. При возрастании объема хранимых данных затраты на перемещения отдельных элементов (если они имеют место) сильно возрастают. То же самое можно сказать о поиске (особенно последовательном, как, например, в списках). Уменьшить их можно, введя в структуру данных иерархию. Для этого можно вложить в элемент одной структуры данных заголовок другой структуры. Соответственно, вложенными будут определения используемых типов данных, а алгоритмы работы будут содержать вложенные циклы для работы с каждым уровнем. Приведем некоторые примеры.

Список, элемент которого содержит массив указателей:

```

struct elem      {          // Элемент односвязного списка
elem *next;
void **pp[20]; }; // Массив указателей на элементы данных
// Подсчет количества элементов в структуре данных
int      count(elem *p) {
elem *q; int cnt;          // Цикл по списку

```

```

for (cnt=0, q=p; q!=NULL; q=q->next) {
    int i; // Цикл по массиву указателей
    for (i=0; q->pp[i]!=NULL; i++)
        cnt++;
    } return cnt; }

```

Массив, каждый элемент которого является заголовком списка:

```

struct list {
    list *next;
    void *data; };
int count(list *p[]) {
    int k,cnt; // Цикл по массиву заголовков списков
    for (cnt=0, k=0; p[k]!=NULL; k++) {
        list *q; // Цикл по списку
        for (q=p[k]; q!=NULL; q=q->next)
            cnt++;
        } return cnt; }

```

Двухуровневый массив указателей:

```

void **p[20]; // Массив указателей на массивы указателей
int count(void **p[]) {
    int k,cnt; // Цикл по массиву верхнего уровня
    for (cnt=0, k=0; p[k]!=NULL; k++) {
        int i; // Цикл по массиву нижнего уровня
        for (i=0; p[k][i]!=NULL; i++)
            cnt++;
        } return cnt; }

```

Логическая нумерация элементов. Логическая нумерация в иерархической структуре данных, как и везде, определяется последовательностью обхода хранимых в ней элементов. Обратите внимание, что внутренние индексы и номера (элементов массивов, списков, вершин деревьев) не имеют к этому никакого отношения. Это тем более важно, что резервирование памяти для массивов производится однократно с учетом последующего их заполнения (то есть любой массив заполнен всегда «частично»).

Локальность изменений. Любая иерархия хороша тем, что изменения отдельных ее частей в большинстве случаев происходят локально, то есть не затрагивают системы в целом. Применительно к иерархической структуре данных это означает, что возможные перемещения объектов или перераспределение памяти должны осуществляться в компонентах нижнего уровня, не затрагивая ни соседей, ни вышележащей структуры данных. Например, в двухуровневом массиве указателей указатель на новый объект включается в массив нижнего уровня, размерность которого ограничена. При отсутствии переполнения указатели будут перемещаться только в границах этого массива. При переполнении же должна быть выполнена более сложная процедура, сохраняющая логиче-

скую организацию структуры данных и ее возможность к расширению. Для двухуровневого массива указателей в качестве одного из вариантов возможно создание динамического массива указателей нижнего уровня и перенесение в него половины указателей из заполненного. Естественно, что новый массив указателей должен быть связан со структурой данных верхнего уровня – его адрес помещается в массив указателей вслед за адресом переполнившегося.

Сбалансированность структур данных. Необходимой платой за перечисленные достоинства является поддержка необходимой сбалансированности – размерности структур данных нижнего уровня должны быть примерно одинаковы. Алгоритмы, выполняющие эту процедуру при каждой операции включения/исключения, могут быть достаточно громоздкими. Альтернатива – периодическое «утрясание» всей структуры данных (например, переписыванием всех ее элементов в аналогичную новую структуру) при значительном нарушении сбалансированности.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Двухуровневый массив указателей. Массив указателей верхнего уровня – статический; массивы указателей нижнего уровня – динамические уже потому, что создаются они в процессе заполнения структуры данных. Однако размерность их фиксирована и при переполнении память под них не перераспределяется.

```
//-----38-01.cpp
//----- Двухуровневый массив указателей на целые
#define N 4
int **p[20]={NULL}; // Исходное состояние – структура данных пуста
//--- Вспомогательные функции для нижнего уровня
int size(int *p[]) // Количество элементов в массиве указателей
{ for (int i=0; p[i]!=NULL; i++); return i; }
//--- Включение в массив указателей нижнего уровня по номеру
int F3(int *p[], int *q, int n) {
int i,m=size(p);
for (i=m; i>=n; i--) p[i+1] = p[i];
p[n] = q;
return m+1==N; } // Результат – проверка на переполнение
```

В структуре данных применяется сквозная нумерация элементов, то есть логический номер определяется в процессе обхода структуры данных. При этом индексы элемента в массивах верхнего и нижнего уровней значения не имеют (рис. 3.26).

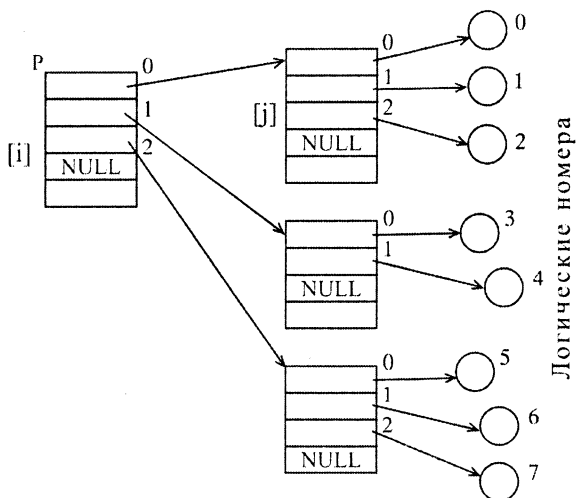


Рис. 3.26

```
//-----38-02.cpp
//----- Обход структуры данных со сквозной нумерацией
void show(int **p[])
{ int i,j,k;
  for (i=0,k=0; p[i] != NULL; i++)
    for (j =0; p[i][j] != NULL; j++,k++)
      printf("A[%d(%d,%d)]=%d\n",k,i,j,*p[i][j]);
}
```

В алгоритме включения по логическому номеру из логического номера вычитается количество указателей в текущем массиве указателей нижнего уровня, пока не будет найден тот, в который попадает новый указатель. При включении указателя в массив нижнего уровня соседние массивы не меняются, то есть структура данных модифицируется локально. Только при его переполнении создается дополнительный массив указателей, в который переписывается половина указателей из исходного. Указатель на новый массив также включается в массив верхнего уровня.

```
//-----38-03.cpp
//----- Включение по логическому номеру
void Insert_Num(int **p[],int *q, int n)
{ int i,j,l,sz;
  if (p[0]==NULL){ // Отдельно для пустой структуры данных
    p[0]=new int*[N+1];
    p[0][0]=q; p[0][1]=NULL; return;
  } // Поиск места включения
  for (i =0; p[i] != NULL; i++,n-=sz) {
    sz=size(p[i]); // Количество указателей в массиве
```

```

    if (n<=sz) break;           // Номер попадает в текущий массив
    }
    if (p[i]==NULL)             // Не найден – включить последним
    { i--; n=size(p[i]); }
    if (F3(p[i],q,n))           // Вызов функции включения для нижнего уровня
    {                             // Переполнение – создание нового массива
        for (int ii=0; p[ii] != NULL; ii++);
        for(int h=ii;h>i;h--)    // Раздвижка в массиве указателей
            p[h+1]=p[h];         // верхнего уровня
        p[i+1]=new int*[N+1];    // Создание массива нижнего уровня
        for(j=0;j<N/2;j++)      // Перенос указателей
            p[i+1][j]=p[i][j+N/2];
        p[i][N/2]=NULL;
        p[i+1][N/2]=NULL;
    }
}

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Разработать заданные функции для иерархической (двухуровневой) структуры данных.

1. Список – элемент содержит статический массив указателей на упорядоченные строки. Включение с сохранением упорядоченности. Если после включения строки массив заполняется полностью, то создается еще один элемент списка с массивом указателей, в который переписывается половина указателей из старого.

2. Список, каждый элемент которого содержит динамический массив указателей на строки, упорядоченные по длине. Размерность массива в каждом следующем элементе в два раза больше, чем в предыдущем. Строка включается в структуру данных с сохранением упорядоченности. Если строка включается в заполненный массив, то последний указатель перемещается в следующий элемент и так далее.

3. Двухуровневый массив указателей на упорядоченные строки. Массив верхнего уровня – статический, массивы нижнего уровня – динамические. Включение строки с сохранением упорядоченности. Если после включения строки массив заполняется полностью, то создается еще один массив указателей, в который переписывается половина указателей из старого.

4. Двухуровневый массив указателей на строки, упорядоченные по длине. Размерность каждого следующего массива нижнего уровня в два раза больше предыдущего. Строка включается в структуру данных с сохранением упорядоченности. Если строка включается в заполненный массив, то последний указатель перемещается в следующий элемент и так далее.

5. Список – элемент содержит статический массив указателей на строки. Включение новой строки последней. Сортировка выбором: в старой структуре данных выбирается минимальная строка и включается последней в новую структуру данных.

6. Двухуровневый массив указателей на строки. Массив верхнего уровня – статический, массивы нижнего уровня – динамические. Новая строка включается последней. Сортировка выбором: в старой структуре данных выбирается минимальная строка и включается последней в новую структуру данных.

7. Дерево, вершина которого содержит статический массив указателей на строки и N указателей на потомков. Если вершина не заполнена, то строка помещается в текущую вершину, если заполнена – то в поддереву с минимальным количеством включенных строк.

8. Список – элемент содержит статический массив указателей на строки. Включение и удаление строки по логическому номеру. Если после включения строки массив заполняется полностью, то создается еще один элемент списка с массивом указателей, в который переписывается половина указателей из старого.

9. Двухуровневый массив указателей на строки. Массив верхнего уровня – статический, массивы нижнего уровня – динамические. Включение и удаление строки по логическому номеру. Если после включения строки массив заполняется полностью, то создается еще один массив указателей, в который переписывается половина указателей из старого.

10. Массив указателей на заголовки списков. Элемент списка содержит указатель на строку. Включение и удаление строки по заданному логическому номеру и включение последней. При включении строки последней предусмотреть ограничение длины текущего списка и переход к следующему.

11. Массив указателей на заголовки списков. Элемент списка содержит указатель на строку. Строки упорядочены в порядке возрастания. Включение с сохранением упорядоченности и ускоренный поиск с проверкой только первого элемента списка.

12. Массив указателей на заголовки списков. Элемент списка содержит указатель на строку. Включение нового элемента последним. Предусмотреть ограничение длины текущего списка и переход к следующему. Сортировка выбором: выбирается минимальная строка, исключается и включается последней в новую структуру данных.

13. Список – каждый элемент является заголовком односвязного списка. Элемент списка второго уровня содержит указатель на строку. Строки упорядочены. Включение с сохранением упорядоченности. Включение элемента последним в список производить с учетом выравнивания длины текущего и следующего списков.

14. Список – каждый элемент является заголовком односвязного списка. Элемент списка второго уровня содержит указатель на строку. Включение и удаление по логическому номеру. Включение элемента последним в список производить с учетом выравнивания длины текущего и следующего списков.

15. Список – каждый элемент является заголовком односвязного списка. Элемент списка второго уровня содержит указатель на строку. Включение элемента последним в список производить с учетом выравнивания длины текущего и следующего списков. Сортировка выбором: выбирается минимальная строка, исключается и включается последней в новую структуру данных.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Пример оформления тестового задания

```
//-----38-04.cpp
struct list { void *data; list *next; };
void *F( list *p[], int (*pf)(void*,void*))
{ list *q; void *pmin=p[0]->data;
for ( int i=0; p[i] != NULL; i++)
    for ( q = p[i]; q != NULL; q = q->next)
        if ((*pf)(q->data ,pmin) < 0) pmin=q->data;
return pmin; }
```

Анализ этого теста производится по формальным признакам. Наличие вложенных циклов говорит о двухуровневой структуре данных. Структурированный тип **list** с единственным указателем на переменную такого же типа и внешний вид внутреннего цикла говорят о том, что нижний уровень представляет собой односвязный список. Структура данных верхнего уровня – массив указателей на переменные типа **list**, каждый элемент его является заголовком списка – указателем на первый элемент. Соответственно, внешний цикл перемещается по массиву заголовков, выбирая эти указатели.

Элемент списка содержит указатель на переменную неопределенного типа **void***. Сама функция является итератором, тип объектов, хранимых в структуре данных, ей неизвестен. В качестве второго параметра он получает указатель на ту функцию, которая

учитывает конкретный вид этих объектов. Выберем в качестве хранимых объектов строки, тогда при инициализации элементов списка их можно заполнить строковыми константами – указателями на эти строки, размещенные в памяти самим транслятором. Сначала определяются переменные – элементы списка (список задается «хвостом вперед»). Затем массив указателей инициализируется указателями на переменные – начальные элементы списков.

```
list a1={"this",NULL}, a2={"is",&a1}, a3={"the",&a2};
list b1={"array",NULL}, b2={"of lists",&b1}, b3={"of strings",&b2};
list *pp[]={&a3,&b3,NULL};
```

Алгоритм итератора содержит стандартный контекст поиска минимума. Естественно, это производится итератором во всей структуре данных, однако сам способ сравнения элементов определяется внешней функцией, которая получает два указателя на текущий минимальный объект и объект, извлеченный из структуры данных. Для строк выберем функцию сравнения их по длине, которая возвращает разность этих длин. Сама функция получает указатели на объекты типа **void***, но поскольку она «знает, что это строки», то преобразует их к типу **char***.

```
int cmp(void *p1,void *p2){
    return strlen((char*)p1) - strlen((char*)p2); }
```

Таким образом, итератор возвратит указатель на строку минимальной длины. Последнее, что нужно сделать, преобразовать результат функции от типа **void*** к типу **char***, опять-таки потому, что итератор возвращает указатель на объект «вообще», то есть произвольного типа, а мы «знаем», что он является строкой.

```
void main(){ puts((char*)F(pp,cmp)); }
```

Определите вид итератора и структуру данных, с которой он работает. Напишите вызов функции для статической структуры данных. Обратите внимание, что при инициализации сначала определяются переменные – элементы структур данных нижнего уровня, указатели на которые помещаются в поля переменных, составляющих структуру данных верхнего уровня.

```
//-----38-08.cpp
//----- 1
struct x1 { void *data; x1 *next; };
void F1( x1 **p, void (*pf)(void*))
{ x1 *q;
  for (; *p != NULL; p++)
    for (q = *p; q != NULL; q = q->next)
      (*pf)(q->data);}
```

```

//----- 2
struct x2 { void *data; x2 *next; };
struct sxxx { x2 *ph; sxxx *next; };
void F2( sxxx *p, void (*pf)(void*)) {
x2 *q;
for (; p != NULL; p = p->next)
    for ( q = p->ph; q!=NULL; q=q->next)
        (*pf)(q->data); }

//----- 3
struct x3 { void **data; x3 *next; };
void F3( x3 *p, void (*pf)(void*)) {
void **q;
for (; p != NULL; p = p->next)
    for (q = p->data; *q != NULL; q++)
        (*pf)(*q); }

//----- 4
void F4(void ***p, void (*pf)(void*))
{ void **q;
for (; *p != NULL; p++)
    for (q = *p; *q != NULL; q++)
        (*pf)(*q); }

//----- 5
void F5(void *p, int sz, int n, void (*pf)(void*))
{ char *q;
for (q = (char*)p; n > 0; n--, q+=sz) (*pf)(q);
}

//----- 6
struct x6 { void *data; x6 **link; };
void F6( x6 *p, void (*pf)(void*)) {
x6 **q;
if (p==NULL) return;
(*pf)(p->data);
for (q = p->link; q!=NULL && *q != NULL; q++)
    F6(*q,pf); }

//----- 7
struct x7 { void **data; x7 *r, *l; };
void F7( x7 *p, void (*pf)(void*))
{ void **q;
if (p==NULL) return;
F7(p->r, pf);
for (q = p->data; q!=NULL && *q != NULL; q++)
    (*pf)(*q);
F7(p->l, pf); }

//----- 8
struct x8 { void *data; x8 *next; };
struct zzz { x8 *ph; zzz *r, *l; };
void F8( zzz *p, void (*pf)(void*)) {
x8 *q;
if (p==NULL) return;
F8(p->r, pf);
for (q = p->ph; q != NULL; q = q->next)
    (*pf)(q->data);
F8(p->l, pf); }

//----- 9
struct x9 { void *data; x9 *next,*pred; };
void F9( x9 *p, void (*pf)(void*))
{ x9 *q; if (p==NULL) return;

```

```

q = p;
do {
    (*pf)(q->data);
    q = q->next;
} while (q != p); }
//----- 10
void *F10(void **p, int (*pf)(void*))
{ int i,j;
  for (i=0; p[i] != NULL; i++)
    for (j=0; p[i][j] != NULL; j++)
      if ((*pf)(p[i][j])) return p[i][j];
  return NULL; }
//----- 11
void F11(void **p, void (*pf)(void*))
{ int i,j;
  for (i=0; p[i] != NULL; i++)
    for (j=0; p[i][j] != NULL; j++) (*pf)(p[i][j]); }
//----- 12
typedef int (*PCMP)(void*, void*);
void F12(void **p, PCMP pf, void *q)
{ int n,i;
  for (n=0; p[n]!=NULL; n++)
    if ((*pf)(q,p[n]) >0) break;
  for (i=n; p[i] !=NULL; i++);
  for (; i >=n; i--) p[i+1]=p[i];
  p[n]=q; }
//----- 13
typedef int (*PCMP)(void*, void*);
void *F13(void *p[], PCMP pf, void *q)
{ int h,l,m ,rr;
  for (h=0; p[h]!=NULL; h++);
  for (h--, l=0; l <=h;) {
    m = (h+l)/2;
    if ((rr=(*pf)(q, p[m]) )==0) return(p[m]);
    else
      if (rr<0) h = m-1;
      else l = m+1; }
  return(NULL); }
//----- 14
typedef int (*PTEST)(void*);
void *F14(void *p, int sz, int n, PTEST pf)
{ char *q;
  for (q=(char*)p; n!=0; q +=sz, n--)
    if ((*pf)(q)) return(q);
  return(NULL); }
//----- 15
typedef int (*PCMP)(void*, void*);
struct x15 { void *data; x15 *next; };
void *F15( x15 **p, PCMP pf)
{ x15 *q; void *s;
  for (s=p[0]->data; *p != NULL; p++)
    for (q = *p; q != NULL; q = q->next)
      if ((*pf)(s,q->data)<0) s=q->data;
  return s; }
//----- 16
typedef int (*PCMP)(void*, void*);
struct x16 { void *data; x16 *next; };

```

```

struct s16 { x16 *ph; s16 *next; };
void *F16( s16 *p, PCMP pf)
{ x16 *q; void *s;
for (s=p->ph->data; p != NULL; p = p->next)
    for ( q = p->ph; q!=NULL; q=q->next)
        if ((*pf)(s,q->data)<0) s=q->data;
return s; }
//----- 17
typedef int (*PCMP)(void*, void*);
struct x17 { void **data; x17 *next; };
void *F17( x17 *p, PCMP pf)
{ void **q; void *s;
for (s=p->data[0]; p != NULL; p = p->next)
    for ( q = p->data; q!=NULL && *q != NULL; q++)
        if ((*pf)(s,*q)<0) s = *q;
return s; }
//----- 18
typedef int (*PCMP)(void*, void*);
void *F18(void ***p, PCMP pf)
{ void **q, *s;
for (s=p[0][0]; *p != NULL; p++)
    for (q = *p; *q != NULL; q++)
        if ((*pf)(s,*q)<0) s = *q;
return s; }
//----- 19
typedef int (*PCMP)(void*, void*);
void *F19(void *p, int sz, int n, PCMP pf)
{ char *q; void *s;
for (q = (char*)p, s = p; n > 0; n--, q+=sz)
    if ((*pf)(s,q)<0) s=q;
return s; }
//----- 20
typedef int (*PCMP)(void*, void*);
struct x20 { void *data; x20 **link; };
void *F20( x20 *p, PCMP pf)
{ x20 **q; void *s,*r;
if (p==NULL) return NULL;
s = p->data;
for (q = p->link; q!=NULL && *q != NULL; q++) {
    r=F20(*q,pf);
    if (r!=NULL && (*pf)(s,r)<0) s=r;
}
return s; }
//----- 21
typedef int (*PCMP)(void*, void*);
struct x21 { void *data; x21 *next,*pred; };
void *F21( x21 *p, PCMP pf)
{ x21 *q; void *s;
if (p==NULL) return NULL;
q = p; s = p->data;
do {
    if ((*pf)(s,q->data)<0) s=q->data;
    q = q->next;
} while (q != p);
return s; }
//----- 22
typedef int (*PCMP)(void*, void*);

```

```

void F22(void *p[], PCMP pf)
{ int i,k;
  do { k=0;
    for (i=1;p[i]!=NULL; i++)
      if ((*pf)(p[i-1],p[i]) > 0)
        { void *s = p[i-1];
          p[i-1]=p[i];
          p[i]=s; k++; }
    } while(k); }
//----- 23
typedef int (*PCMP)(void*, void*);
void F23(void *p[], PCMP pf, void *q)
{ int i,j;
  for (i=0; p[i]!=NULL && (*pf)(p[i],q) < 0; i++);
  for (j=0; p[j]!=NULL; j++);
  for (; j>=i; j--) p[j+1]=p[j];
  p[i]=q; }
//----- 24
typedef int (*PCMP)(void*, void*);
struct x24 { void *data; x24 *next; };
void F( x24 **p, PCMP pf, void *q)
{ x24 *s;
  for (; *p!=NULL && (*pf)((*p)->data,q) <0; p = &(*p)->next);
  s = new x24;
  s->data = q;
  s->next = (*p)->next;
  *p = s; }

```

3.9. БИТЫ, БАЙТЫ, МАШИННЫЕ СЛОВА

Машинное слово. Основа представления любых данных – машинное слово. Машинное слово – это упорядоченное множество двоичных разрядов, используемое для хранения команд программы и обрабатываемых данных. Каждый разряд, называемый **битом**, – это двоичное число, принимающее значения только 0 или 1. Разряды в слове нумеруются справа налево, начиная с 0. Количество разрядов в слове называется **размерностью** машинного слова, или **разрядностью** машинного слова. **Байт** – машинное слово минимальной размерности (8 бит), адресуемое компьютером. Размерность байта – 8 бит – принята не только для представления данных в большинстве компьютеров, но и в качестве стандарта для хранения данных на внешних носителях, для передачи данных по каналам связи, для представления текстовой информации. Кроме того, байт является универсальным «измерительным инструментом» – размерность всех форм представления данных устанавливается кратной байту. При этом машинное слово считается разбитым на байты, которые нумеруются, начиная с младших разрядов (рис. 3.27).

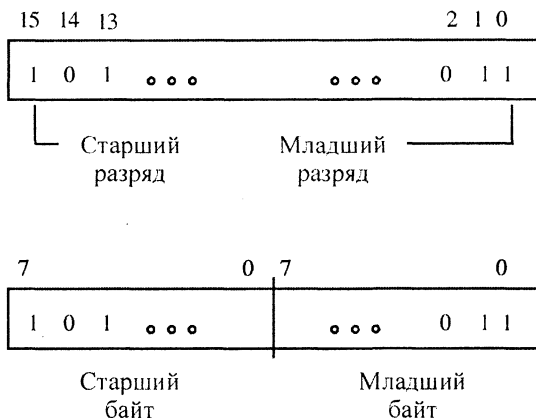


Рис. 3.27

Машинные слова в Си. Базовые типы данных целых чисел реализованы в машинных словах различной размерности, поэтому для задания в программе машинных слов нужно просто определить ту или иную целую переменную. Тип данных **char** всегда соответствует байту, **int** – стандартной размерности машинного слова, обрабатываемого процессором, **long** – машинному слову увеличенной размерности по отношению к стандартному (обычно двойной). Операция **sizeof**, определяющая размерность любого типа данных в байтах, может быть использована и для «измерения» машинных слов.

```
long vv; // Машинное слово двойной длины
for(int i=0; i<8*sizeof(long); i++) // Количество битов в long
{ ... vv ... } // Цикл побитовой обработки слова
```

Представление машинных слов в программе. На практике вместо двоичной системы используются восьмеричная и шестнадцатеричная системы счисления. Это объясняется тем, что одна восьмеричная цифра принимает значения от 0 до 7 и занимает три двоичных разряда. Аналогично шестнадцатеричная цифра принимает значения от 0 до 15, что соответствует четырем двоичным разрядам (**тетрада**). Поскольку обычных цифр для представления значений от 0 до 15 не хватает, то для недостающих используются прописные или строчные латинские буквы:

A - 10, B - 11, C - 12, D - 13, E - 14, F - 15

При необходимости представить машинное слово с заданным значением в его «натуральном» виде – как последовательность

двоичных разрядов, используются шестнадцатеричные и восьмеричные константы. Для этого каждую цифру такой константы нужно разложить в ее двоичное представление.

0x1B8C = 0001 1011 1000 1100
 1 В 8 С

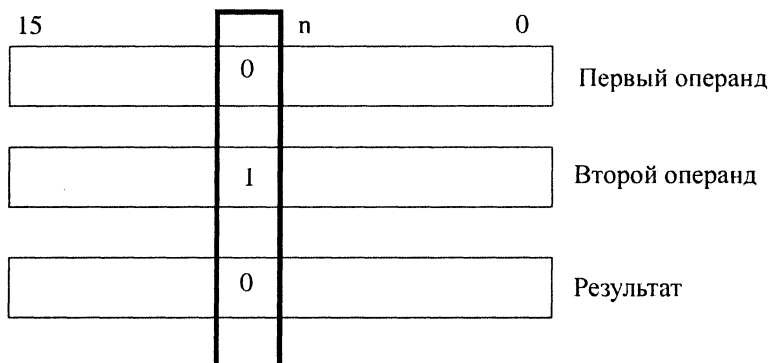
И наоборот, представить в программе машинное слово с заданным сочетанием битов можно, переведя его из двоичного представления в шестнадцатеричную константу, разбив на тетрады и заменив значение каждой из них соответствующей цифрой **0..9 – А..F**.

Но на самом деле программиста обычно не интересует представление всего слова в виде последовательности битов. По условию поставленной задачи ему требуется иметь установленными в 0 или 1 отдельные разряды или их группы. Для этого нужно принять к сведению очевидные вещи: цифре 0 соответствует тетрада с четырьмя нулевыми битами, цифре F – с четырьмя единичными, каждому байту соответствуют две шестнадцатеричные цифры, разряды и байты в машинном слове нумеруются справа налево (парабски), начиная с 0. Например, если в константе требуется установить в 1 девятый разряд машинного слова, то он будет находиться в третьей справа цифре, содержащей разряды с номерами 8..11. Все остальные цифры будут нулевыми. Значение же этой цифры с установленным девятым разрядом будет равно 2. В результате получим константу 0x0200.

Аналогичным образом используются восьмеричные константы. В Си любая константа, содержащая цифры от 0 до 7 и начинающаяся с 0, считается восьмеричной, например 0177556.

Технология работы с машинными словами. Особых секретов в технике работы с машинными словами не существует, если руководствоваться правилом, что машинное слово – это массив битов. То есть алгоритмы работы с машинными словами в первом приближении аналогичны алгоритмам, работающим с массивами. Единственная разница состоит в том, что в «джентльменском наборе» команд процессора отсутствуют команды прямой адресации битов. Взамен их используются **поразрядные** операции, выполняющие одну и ту же логическую операцию или операцию перемещения над всеми разрядами машинного слова одновременно (рис. 3.28). Другое их название – **машинно-ориентированные операции** – отражает тот факт, что они поддерживаются в любой системе команд и любом языке Ассемблера. К ним относятся:

- «|» – поразрядная операция **ИЛИ**
- «&» – поразрядная операция **И**;
- «^» – поразрядная операция **исключающее ИЛИ**;
- «~» – поразрядная операция **инверсии**;
- «>>» – операция **сдвига вправо**;
- «<<» – операция **сдвига влево**.



Логическая операция над n-м разрядом

Рис. 3.28

Формальная сторона логических операций всем известна. Однако программиста интересует содержательная интерпретация поразрядных операций, которая позволяет выполнять различные действия с отдельными битами и их диапазонами – **битовыми полями**: устанавливать, очищать, выделять, инвертировать. Для этого используют поразрядные операции, в которых первый операнд является обрабатываемым машинным словом. Второй операнд, как правило, определяет те биты в первом операнде, которые изменяются при выполнении операции, и в этом случае называется **маской**. Если маска жестко задана в программе, то является просто битовой константой в шестнадцатеричной системе.

0x1F // 0000 0000 0001 1111 – в маске установлены биты 0...4
 0x3C0 // 0000 0011 1100 0000 – в маске установлены биты 6...9
 0x1 // 0000 0000 0000 0001 – в маске установлен младший бит

Часто требуется, чтобы маска была «программируемой», например, занимала заданную последовательность разрядов. В этом случае нужно организовать процесс «пробегания» единичного бита по заданному полю.

```
//-----39-01.cpp
//--- Формирование маски в заданном диапазоне разрядов
long set_mask(int r0, int dn){
long m,v;          // m " бегущий" единичного бита, v маска
m = 1 << r0;       // Сдвинуть единичный бит на r0 разрядов влево
for (v=0; dn!=0; dn--){          // Повторять dn раз
v |= m;           // Установить очередной разряд из m в v
m <<=1;}          // Переместить единичный бит в следующий разряд
return v;}
```

Поразрядная операция И. По отношению ко второму операнду (маске) логическая операция **И** сохраняет (выделяет) те биты первого операнда, которые соответствуют единичным битам маски, и безусловно сбрасывает в 0 те биты результата, которые соответствуют нулевым битам маски. Операция так и называется – **выделение битов по маске**.

```
x x x x x x x x – операнд
0 0 1 1 1 0 0 0 – маска
0 0 x x x 0 0 0 – результат
```

```
b = a & 0x0861; // Выделить биты 0,5,6,11
b = a & 0x00F0; // Выделить биты с 4 по 7
                // (биты второй цифры справа)
```

Выделение битов по маске может сопровождаться проверкой их значений.

```
if ((a & 0x100)!=0) ... // Установлен ли 8-й бит –
                    // (младший бит второго по счету байта)
```

Поразрядная операция ИЛИ. По отношению ко второму операнду (маске) логическая операция **ИЛИ** сохраняет те биты первого операнда, которые соответствуют нулевым битам маски, и безусловно устанавливает в 1 те биты результата, которые соответствуют единичным битам маски. Операция так и называется – **установка битов по маске**.

```
x x x x x x x x - операнд
0 0 1 1 1 0 0 0 - маска
x x 1 1 1 x x x - результат
```

```
a |= 0x0861; // Установить в 1 биты 0,5,6,11
a |= 0x00F0; // Установить в 1 биты с 4 по 7
                // (биты второй цифры справа)
```

Операция **ИЛИ** используется также для объединения значений непересекающихся битовых полей (логическое сложение), которые предварительно выделяются с помощью операции **И**.

```
int a=0x5555,b=0x4444,c;
c = a & 0xFFFF0 | b & 0xF; // c = «aaab»
```

В переменной **c** объединяются битовые поля, выделенные из **a**, – три старшие шестнадцатеричные цифры (12 разрядов), и выделенные из **b**, – младшая шестнадцатеричная цифра (4 разряда).

```
c <=<=1; c |= b & 1; b >=>=1;
```

Содержимое слова **c** сдвигается влево, в результате чего «освобождается место» в самом правом его разряде. Затем операция **И** выделяет младший разряд из машинного слова **b**, который затем переносится в освободившийся разряд **b** с помощью операции **ИЛИ**.

Операция поразрядной инверсии. Поразрядная инверсия меняет значение каждого бита машинного слова на противоположное (инвертирует). Операция **И** в сочетании с инвертированной маской-константой производит **очистку битов по маске**.

```
a &= ~0x0861;    // Очистить биты 0,5,6,11, остальные сохранить
a &= ~0x00F0;    // Очистить биты с 4 по 7, остальные сохранить
                // (биты второй цифры справа)
```

Поразрядная операция исключающее ИЛИ. Поразрядная операция **исключающее ИЛИ** выполняет над парами битов в операндах логическую операцию **исключающее ИЛИ**, называемую также **неравнозначность**, или **сложение по модулю 2**, – результат равен 1 при несовпадении значений битов. По отношению ко второму операнду (маске) логическая операция **исключающее ИЛИ** сохраняет те биты первого операнда, которые соответствуют нулевым битам маски, и инвертирует те биты результата, которые соответствуют единичным битам маски. Операция так и называется – **инвертирование битов по маске**.

```
a ^= 0x0861;    // Инвертировать биты 0,5,6,11
a ^= 0x00F0;    // Инвертировать биты с 4 по 7
                // (биты второй цифры справа)
```

Операция сдвиг влево. Поразрядная операция **сдвиг влево** переносит каждый бит первого операнда на то количество разрядов влево, которое задано вторым операндом, освобождающиеся разряды справа заполняются нулями. Результат операции содержит сдвинутое машинное слово, а сами операнды не изменяются.

Естественно, что от программиста не требуется вручную интерпретировать перемещение разрядов машинного слова. Каждое перемещение имеет свою содержательную интерпретацию.

```
a <<= 4;        // Сдвиг влево на одну шестнадцатеричную цифру
a = 1<<n;       // Установить 1 в n-й разряд машинного слова
```

Операции сдвига часто используются для «подгонки» групп двоичных разрядов к требуемому их местоположению в машинном слове. После чего в дело вступают операции **И**, **ИЛИ** для выделения и изменения значений полей.

```
long a=0x12345678; // Поменять местами две младшие цифры
long b = a & ~0xFF | (a >>4) & 0xF | (a <<4) & 0xF0;
```

Первая операция **И** очищает две младшие шестнадцатеричные цифры (8 разрядов), вторая операция перемещает первую цифру на место нулевой (и выделяет), третья операция перемещает нулевую цифру на место первой, после чего все поля объединяются по **ИЛИ**.

У операции сдвига влево есть еще одна интерпретация. Если рассматривать машинное слово как целое без знака, то однократный сдвиг увеличивает его значение в два раза, двукратный – в четыре раза, **n**-кратный – в 2^n раз. В таком виде, например, умножение числа на 10 можно представить так:

$$a * 10 \dots a * (8 + 2) \dots 8 * a + 2 * a \dots (a < < 3) + (a < < 1)$$

Операция сдвиг вправо. Поразрядная операция **сдвиг вправо** имеет некоторые особенности выполнения. По аналогии со сдвигом влево операция сдвига вправо на **n** разрядов интерпретируется как целочисленное деление на 2^n . При этом заполнение освобождающихся старших разрядов производится таким образом, чтобы сдвиг соответствовал операции деления с учетом формы представления целого. Для беззнакового целого заполнение должно производиться нулями (**логический сдвиг**), а для целого со знаком – сопровождаться дублированием значения старшего знакового разряда (**арифметический сдвиг**). В последнем случае отрицательное число при сдвиге останется отрицательным:

```
int          n=0xFF00;      n>>=4; // n=0xFFFF0;
unsigned     u=0xFF00;      u>>=4; // u=0x0FF0;
```

Формы представления числовых данных. Целое без знака. Содержимое машинного слова используется для представления целых положительных значений без знака. Каждый разряд машинного слова имеет вес, в два раза больший, чем вес соседнего правого, то есть 1, 2, 4, 8, 16 и т.д., или последовательные степени 2. Тогда значение числа в машинном слове равно сумме произведений значений разрядов на их веса:

$$R_0 * 1 + R_1 * 2 + R_2 * 4 + \dots + R_{15} * 32768 \text{ или } R_0 * 2^0 + R_1 * 2^1 + \dots + R_{15} * 2^{16}$$

Например, машинное слово 000000000010001001 имеет значение $1+8+128 = 137$. Получить значение восьмеричной или шестнадцатеричной константы в десятичной системе можно также путем умножения цифр числа на веса разрядов – последовательные степени 8 или 16:

$$0x6DCC = 12(C) \cdot 16^0 + 12(C) \cdot 16^1 + 13(D) \cdot 16^2 + 6 \cdot 16^3 = 12 + 12 \cdot 16 + 13 \cdot 256 + 6 \cdot 4096$$

Представление отрицательных чисел. Дополнительный код. Открытый характер языка Си, его близость к архитектуре компьютера позволяют наблюдать, а при необходимости и использовать особенности представления целых чисел со знаком на уровне машинных слов. Идея заключается в том, что область отрицательных отображается на область беззнаковых положительных таким образом, что для них можно использовать часть команд для целых без знака (сложение).

Дополнительный код – беззнаковая форма представления чисел со знаком.

Такой «фокус» может быть произведен в любой системе счисления. Продемонстрируем его для начала в десятичной.

Пусть имеется трехразрядное десятичное число со знаком. Представим его в следующем виде:

- добавим слева еще одну цифру – знак числа, принимающую всего два значения: 0 – плюс, 9 – минус;
- положительные числа представим обычным образом;
- каждую цифру отрицательного числа заменим на дополнение ее до $n-1$, где n – основание системы счисления. Для десятичной системы – это дополнение до 9, то есть цифра, которая в сумме с исходной дает 9;
- к полученному числу добавим 1.

Такое представление отрицательных чисел называется **дополнительным кодом**. Он обладает одним замечательным свойством: сложение чисел в дополнительном коде по правилам сложения целых без знака дает корректный результат, который также получается в дополнительном коде.

- 3 8 6 - отрицательное число
 9 6 1 3 - дополнение каждой цифры до 9
 9 6 1 4 - добавление 1

$$512 - 386 =$$

```

  0 5 1 2
+ 9 6 1 4

```

```

1 0 1 2 6
  0 1 2 6

```

- для знака используется 0 или 9
(переполнение игнорируется)

```

119 - 386 =
  0 1 1 9
+ 9 6 1 4

```

```

  9 7 3 3
- 2 6 6
- 2 6 7

```

- результат в дополнительном коде
- дополнение каждой цифры до 9
- добавление 1

Если внимательно присмотреться, то дополнение каждой цифры до 9 имеет некоторую аналогию с вычитанием. Как бы там ни было, получаем следующий результат: вычитание или же сложение чисел со знаком заменяется операцией сложения, выполненной для беззнаковых чисел.

В двоичной системе счисления дополнение каждой цифры до основания системы счисления без единицы ($n-1 = 1$) выглядит как инвертирование двоичного разряда. Если же знак числа представляется старшим разрядом машинного слова, то получается простой способ представления отрицательного числа:

- взять абсолютное значение числа в двоичной системе;
- инвертировать все разряды, включая знаковый;
- добавить к результату 1.

Используя поразрядные операции, можно «превратить» положительное число в отрицательное:

```
int a=125; a = ~a + 1; // Эквивалентно a=-a;
```

Все эти нюансы, вообще-то, не важны для программиста, поскольку ему нет нужды вручную выполнять сложение или вычитание ни в двоичной, ни в шестнадцатеричной системах, за него это сделает компьютер. На самом деле от программиста, даже при работе на уровне внутреннего представления данных, достаточно знать правила отображения диапазонов положительных и отрицательных значений знаковых чисел на диапазон беззнаковых. Используемая форма преобразования приводит к тому, что отрицательные числа отображаются на «вторую половину» диапазона беззнаковых целых, причем таким образом, что значение -1 соответствует максимальному беззнаковому (то есть **0xFFFF** во внутреннем представлении), а минимальное отрицательное – середине интервала (то есть **0x8001**). Значение **0x8000** является «водоразделом» положительных и отрицательных и называется **минус 0**. Все

отрицательные числа имеют старший (знаковый) бит, установленный в 1 (рис. 3.29).

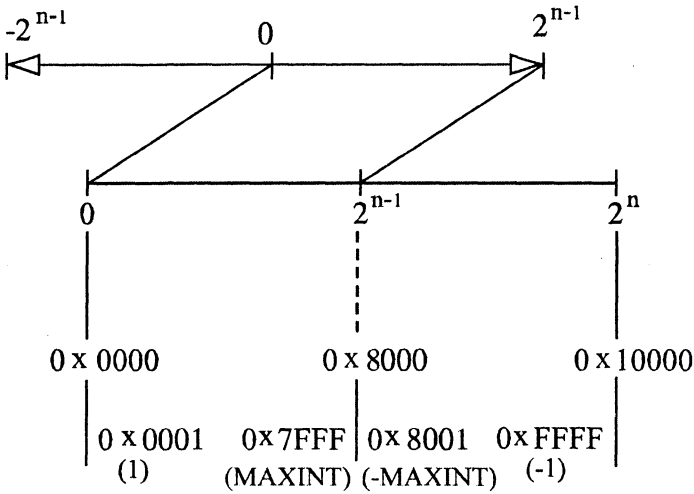


Рис. 3.29

Целое со знаком	Значение в дополнительном коде
0	0
1	1
...	...
+32766	0x7FFE
+32767 (+MAXINT)	0x7FFF
-1	0xFFFF
-2	0xFFFE
-16	0xFFF0
-32767 (-MAXINT)	0x8001
не определено (минус0)	0x8000

Как видим, положительные числа представлены аналогично беззнаковым. Машинное слово со всеми разрядами, установленными в 1, соответствует значению -1 , а затем по убыванию: -2 , -3 и т.д.

Преобразование типов операндов в выражения. Преобразования базовых типов данных, соответствующих целым числам, не всегда сохраняют значения переменных и могут приводить к трудно обнаруживаемым ошибкам. В ряде случаев необходимо апеллировать к внутренним формам представления и действиям над ними. Преобразование формы представления может включать в себя:

- преобразование целой переменной в переменную с плавающей точкой, и наоборот;
- увеличение или уменьшение разрядности машинного слова, то есть «растягивание» или «усечение» целой переменной;
- преобразование знаковой формы представления целого в беззнаковую, и наоборот.

Уменьшение разрядности машинного слова всегда происходит путем отсечения старших разрядов числа, что может привести к ошибкам потери значащих цифр и разрядов:

```
int      n=0x7654;
char     c; c = n;           // Потеря значащих цифр (0x54)
```

Увеличение разрядности приводит к появлению дополнительных старших разрядов числа. При этом способ их заполнения зависит от формы представления целого и обеспечивает сохранение значения переменной в данной форме представления:

- для беззнаковых целых заполнение производится нулями;
- для целых со знаком дополнительные разряды заполняются одним и тем же значением знакового (старшего) разряда.

```
int      n; unsigned u;
char     c=0x84;      n = c;           // Значение n=0xFF84
unsigned char uc=0x84; u = uc;         // Значение u=0x0084
```

При преобразовании вещественного к целому происходит потеря дробной части, при этом возможно возникновение ошибок переполнения и потери значащих цифр, когда полученное целое имеет слишком большое значение.

```
double d1=855.666, d2=0.5E16;
int     n; n = d1;           // Отбрасывание дробной части
n = d2;                          // Потеря значимости
```

Преобразование знаковой формы в беззнаковую и обратно не сопровождается изменением значения целого числа и вообще не приводит к выполнению каких-либо действий в программе. В таких случаях транслятор «запоминает», что форма представления целого изменилась, и только.

```
int      n=-1;
unsigned d; d = n;           // Значение d=0xFFFF (-1)
```

Самое главное для программиста, что в языке не предусмотрены средства автоматической реакции на ошибки преобразования типов данных, поэтому «отлавливать» их должна сама программа.

Преобразования типов данных операндов происходят в программе в трех случаях:

– при выполнении операции присваивания, когда значение переменной или выражения из правой части запоминается в переменной в левой части;

– при прямом указании на необходимость изменения типа данных переменной или выражения, для чего используется операция явного преобразования типа;

– при выполнении бинарных операций над операндами различных типов, когда более «длинный» операнд превалирует над более «коротким», вещественное – над целым, а беззнаковое – над знаковым.

В последнем случае неявные преобразования выполняются в такой последовательности: короткие типы данных (знаковые и беззнаковые) удлиняются до **int** и **double**, а выполнение любой бинарной операции с одним **long double**, **double**, **long**, **unsigned** ведет к преобразованию другого операнда в тот же тип. Это может сопровождаться перечисленными выше действиями: увеличением разрядности операнда путем его «удлинения», преобразованием в форму с плавающей точкой и изменением беззнаковой формы представления на знаковую, и наоборот.

Следует обратить внимание на одну тонкость: если в процессе преобразования требуется увеличение разрядности переменной, то на способ ее «удлинения» влияет только наличие или отсутствие знака у самой переменной. Второй операнд, к типу которого осуществляется приведение, на этот процесс не влияет:

```
long    l=0x21;
unsigned d=0xFF00;
l + d ...
// 0x00000021 + 0xFF00 = 0x00000021 + 0x0000FF00 = 0x0000FF21
```

В данном случае производится преобразование целого обычной точности без знака (**unsigned**) в длинное целое со знаком (**long**). В процессе преобразования «удлинение» переменной **d** производится как беззнаковое (разряды заполняются нулями), хотя второй операнд и имеет знак. Рассмотрим еще несколько примеров.

```
int i;    i = 0xFFFF;
```

Целая переменная со знаком получает значение **FFFF**, что соответствует **-1** для знаковой формы в дополнительном коде. Изменение формы представления с беззнаковой на знаковую не сопровождается никакими действиями.

```
int i = 0xFFFF;
long l; l = i;
```

Преобразование **int** в **long** сопровождается «удлинением» переменной, что с учетом представления **i** со знаком дает **FFFFFFFF**, то есть длинное целое со значением **-1**.

```
unsigned n = 0xFF00;  
long l;    l = n;
```

Переменная **n** «удлиняется» как целое без знака, то есть переменная **l** получит значение **0000FF00**.

```
int i; unsigned u;  
i = u = 0xFFFF;  
if (i > 5) ...           // "Ложь"  
if (u > 5) ...           // "Истина"
```

Значения переменных без знака и со знаком равны **FFFF** или **-1**. Но результаты сравнения противоположны, так как во втором случае сравнение проводится для беззнаковых целых по их абсолютной величине, а в первом случае – путем проверки знака результата вычитания, то есть с учетом знаковой формы представления чисел.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Определение разрядности целого числа. Простейший способ – сдвигать беззнаковое целое вправо, пока оно не станет равным 0. Количество сдвигов и есть его разрядность.

```
//-----39-02.cpp  
//----- Определение разрядности числа  
int wordlen(unsigned long vv){  
    for (int i=0; vv!=0; i++, vv>>=1);  
    return i;}  

```

Подсчет количества единичных битов. При сдвиге вправо все биты числа будут последовательно находиться в младшем разряде, из которого их нужно выделять с использованием операции **И** с единичной маской.

```
//-----39-03.cpp  
//----- Подсчет количества единичных битов  
int what_is_1( unsigned long n)  
{ int i,s;  
  for (i=0,s=0; i < sizeof(long) * 8; i++)  
    { if (n & 1) s++; n>>=1; }      // Проверить младший бит и сдвинуть  
  return s; }  

```

Упаковка данных фиксированными полями. В простейших случаях упаковки битовых полей происходит привязка к границам машинных слов, то есть отдельные битовые поля не пересекаются с этими границами. В следующем примере большие латинские бу-

квы и цифры кодируются пятибитовым кодом, уложенным по три в целую 16-разрядную переменную. Остальные символы задаются в виде последовательности из трех таких кодов – идентификатора (**other**) и групп разрядов самого символа (0...4 и 5...7).

```
//-----39-04.cpp
//-----Упаковка символов 5-битным кодом
void put_5(int A[], int &n, int vv){ // Запись очередного 5-битного поля
if (n%3==0) A[n/3]=0; // Очистить очередное слово
A[n/3] |= vv << ((n%3)*5); // Сдвинуть на 0,5,10 битов
n++; }
#define other 40
int pack(int A[], char c[]){ // Упаковка строки
int i=0,m=0;
do {
if (c[i]>='A' && c[i]<='Z') put_5(A,m,c[i]-'A'+1);
else
if (c[i]>='0' && c[i]<='9') put_5(A,m,c[i]-'0'+27);
else {
put_5(A,m,other); // Идентификатор остальных символов
put_5(A,m,c[i]&0x1F); // 5 младших разрядов символа
put_5(A,m,(c[i]>>5) &0x7);} // 3 старших разряда символа
} while (c[i++]!=0);
return (m+1)/3;}

```

Упаковка данных полями переменной длины. Наибольшую плотность упаковки можно достичь, если сделать границы слов (байтов) «прозрачными», представив упакованные данные в виде неограниченной последовательности битов, «плотно» уложенных в массиве машинных слов (байтов). Для того чтобы «вынести за скобки» часть программы, работающую с отдельными битами и их полями, и учитывая тот факт, что биты записываются и извлекаются только последовательно, разработаем две функции добавления и выделения очередного бита по заданному номеру. Он будет передаваться ссылкой на переменную – счетчик очередного бита, увеличиваемый при каждом вызове функции.

```
//-----39-05.cpp
//----- Извлечение и запись бита
long getbit(char c[], int &n) { // c[] - массив байтов, n - номер бита
int nb = n/8; // Номер байта
int ni = n%8; // Номер бита в байте
n++;
return (c[nb]>>ni) & 1; } // Сдвинуть к младшему и выделить
void putbit(char c[], int &n, int v ){
int nb = n/8;
int ni = n%8;
n++;
c[nb] = c[nb] & ~(1<<ni) | ( (v&1) << ni);}

```

Функция, извлекающая последовательность битов числа младшими разрядами вперед, производит повторную сборку их в машинное слово заданной размерности с использованием операций сдвига и поразрядного **ИЛИ**. Функция упаковки слова выделяет последовательность битов, начиная с младшего, использует операцию сдвига и вызывает функцию записи.

```
//----- Извлечение слова заданной размерности
// sz - количество битов
unsigned long getword(char c[], int &n, int sz) {
    unsigned long v = 0;
    for(int i = 0; i<sz; i++) v |= getbit(c, n)<<i;
    return v; }
void putword(char c[], int &n, int sz, long v){
    while(sz--!=0) //Пока количество битов не равно нулю
        { putbit(c, n, v&1); v>>=1;}}
```

Последующие действия связаны уже с форматом представления данных – наличием управляющих полей и полей данных взаимосвязанной размерности. В них поразрядные операции могут вообще отсутствовать. Это видно на примере функций упаковки и распаковки целых переменных различной размерности **char**, **int**, **long**: перед каждым числом размещаются 2 бита, определяющие размерность числа: 00 – конец последовательности, 01 – **char**, 10 – **int**, 11 – **long**. После них размещаются разряды самого числа.

```
//-----39-06.cpp
//---- Упаковка и распаковка переменных различной размерности
void unpack(char c[]){
    int n=0; long vv;
    while(1){
        int mode=getword(c,n,2); // Извлечение 2-разрядного кода
        switch(mode){ // Переключение по типу переменной
            case 0: return;
            case 1: vv=getword(c,n,8); break; // 01 извлечь байт (char)
            case 2: vv=getword(c,n,16);break; // 10 извлечь int
            case 3: vv=getword(c,n,32);break; // 11 извлечь long
        } printf("%ld\n",vv);
    }
}
void pack(char c[]){
    int n=0; long vv;
    do { scanf("%ld",&vv);
        if(vv==0) putword(c,n,2,0); // Запись 2-разрядного кода 00
        else
            if (vv < 256) {
                putword(c,n,2,1); // Запись 2-разрядного кода 01
                putword(c,n,8,vv); // Запись 8-разрядного кода числа
            }
            else
                if (vv < 32768) {
                    putword(c,n,2,2); // Запись 2-разрядного кода 10
                    putword(c,n,16,vv); // Запись 16-разрядного кода числа
                }
    } while(vv != 0);
}
```

```

else {
    putword(c,n,2,3); // Запись 2-разрядного кода 11
    putword(c,n,32,vv);} // Запись 32-разрядного кода числа
} while (vv!=0); }

```

Поразрядная сортировка разделением. Одним из вариантов сортировки разделением является поразрядная сортировка разделением. Напомним, что сущность алгоритма состоит в разделении исходного массива на две части по принципу «меньше-больше» относительно некоторого среднего значения, именуемого медианой. Тогда полученные части могут быть отсортированы независимо, в том числе с помощью того же разделения, то есть рекурсивно. Таким «водоразделом» в процессе разделения может выступать очередной бит машинного слова. Массив сортируемых значений делится на две части так, что в левой оказываются значения с очередным значащим битом, равным 0, а в правой – со значением 1. Затем обе части массива делятся на две части по значению следующего бита и т.д. В результате, когда мы доберемся до младшего бита, массив окажется упорядоченным.

13	6	11	15	8	14	Исходный
1101	0110	1011	1111	1000	1110	массив
6	15	13	11	8	14	Разделение
6	11	8	15	13	14	по биту 3
6	8	11	13	15	14	Разделение
6	8	11	13	14	15	по биту 2
6	8	11	13	15	14	Разделение
6	8	11	13	14	15	по биту 1
6	8	11	13	14	15	Разделение
6	8	11	13	14	15	по биту 0

В нашем примере после выполнения разделения по третьему биту массив делится на две части, границей которых является значение 8. Затем обе части делятся пополам со значениями границ, определяемых вторым битом, то есть 4 и 12, и т.д.

```

void bitsort(int A[],int a,int b, unsigned m){
int i;
if (a+1 >= b) return; // Интервал сжался в точку
if (m == 0) return; // Проверяемые биты закончились
// Маска после сдвига стала 0
// Разделить массив на две части по значению бита,
// установленного в m, i - граница разделенных частей
bitsort(A,a,i,m >>1);
bitsort(A,i+1,b,m >>1); }

```

Приведенная функция выполняет поразрядную сортировку части массива **A**, ограниченного индексами **a** и **b**. Этот интервал разделяется на две части (интервалы **a...i**, **i+1...b**), в которые попадают значения элементов массива соответственно с нулевым и единич-

ным значением проверяемого бита. Сам бит задается маской **m** – переменной, в которой его значение установлено в 1. Затем функция рекурсивно вызывает самое себя для обработки полученных частей, для которых выполняется разделение по следующему правому биту. Для этого текущая маска **m** сдвигается на один разряд вправо.

Для самого первого вызова рекурсивной функции для всего исходного массива необходимо определить старший значащий бит в его элементах. Для этого ищется максимальный элемент, и для него определяется маска **m**, пробегающая последовательно все разряды справа налево до тех пор, пока она не превысит значение этого максимума.

```
void mainsort(int B[], int n){
int max,i; unsigned m;
for(max = 0, i=0; i< n; i++)
    if (B[i] > max) max = B[i];
for (m=1; m < max; m <= 1);
m >=1;
bitsort(B,0,n-1,m); }
```

Разделение интервала массива по заданному биту происходит по принципу «сжигания свечи с двух концов», аналогично алгоритму «быстрой сортировки» (см. раздел 2.5). Два индекса (**i** и **j**) движутся от концов интервала к середине, оставляя после себя слева и справа разделенные элементы. На каждом шаге производится сравнение битов по маске **m** в элементах массива, находящихся по указанным индексам (границы неразделенной части массива). В зависимости от комбинации битов (четыре варианта) производится перестановка элементов и перемещение одного или обоих индексов к середине:

Состояние пары элементов		Сдвиг границ
Оба на месте	0 1	Сдвинуть обе
Размещены наоборот	1 0	Переставить элементы, сдвинуть обе
Левый на месте	0 0	Сдвинуть левую
Правый на месте	1 1	Сдвинуть правую

```
//-----38-05.cpp
//----- Поразрядная сортировка разделением
void bitsort(int A[],int a,int b, unsigned m){
int i;
if (a+1 >= b) return; // Интервал сжался в точку
if (m == 0) return; // Проверяемые биты закончились
// Маска после сдвига стала 0
// Разделить массив на две части по значению бита,
// установленного в m, i - граница разделенных частей
int j,vv; // Цикл разделения массива
```

```

for (i=a, j=b; i<j; ) // в поразрядной сортировке
    if ((A[i] & m) ==0){
        if ((A[j] & m) !=0) i++,j--; // Вариант 0,1
        else i++; // Вариант 0,0
    }
    else {
        if ((A[j] & m) !=0) j--; // Вариант 1,1
        else { // Вариант 1,0
            vv = A[i]; A[i]=A[j]; A[j]=vv;
            i++, j--;
        }
    }
}
if ((A[i] & m)!=0) i--; // Уточнить границу разделения
bitsort(A,a,i,m >>1);
bitsort(A,i+1,b,m >>1); }

```

Поразрядная распределяющая сортировка. Идея циклического слияния (см. раздел 2.6) имеет свое воплощение и в варианте, работающем с отдельными разрядами. В этом варианте в явном виде отсутствуют сливаемые группы и разделение производится на две части по значению очередного разряда – от старшего к младшему. Слияние происходит обычным образом – сравнением значений. Невероятно, но факт: программа работает.

```

//-----39-07.cpp
//----- Поразрядная распределяющая сортировка
#define MAXINT 0x7FFF
void sort(int in[], int n)
{int m,i,max,i0,i1;
 int *v0=new int[n];
 int *v1=new int[n];
 for (i=0, max=0; i<n; i++)
     if (in[i] > max) max=in[i]; // Определение максимального
 for (m=1; m <=max; m <=<=1); // значащего разряда
 for (m >>=1; m !=0; m >>=1){ // По всем разрядам от старшего
     for (i0=i1=0; i0+i1 < n; ) // Распределение по значению
         if ((in[i0+i1] & m) ==0) // очередного разряда
             v0[i0] = in[i0+i1], i0++;
         else
             v1[i1] = in[i0+i1], i1++;
     v0[i0] = v1[i1] = MAXINT;
     for (i0=i1=0; i0+i1 < n; ) // Слияние по обычному сравнению
         if (v0[i0] < v1[i1]) // значений в последовательностях
             in[i0+i1] = v0[i0], i0++;
         else
             in[i0+i1] = v1[i1], i1++;
     } delete v0; delete v1;}

```

Машинная арифметика – целые произвольной точности. Другим важным приложением поразрядных операций является моделирование программными средствами процессов аппаратной обработки данных на уровне отдельных битов, полей, слов различной размерности. Прежде всего это относится к моделированию (эмуляции) машинных операций с различными формами представ-

ления данных (машинной арифметики). В предлагаемом ниже примере программно моделируются переменные произвольной размерности, представленные массивами беззнаковых байтов в форме, соответствующей их внутреннему представлению в памяти. В такой форме можно реализовать все арифметические операции по аналогичным алгоритмам, которые имеют место на аппаратном уровне в компьютере для базовых типов данных.

Операция сложения выполняется побайтно. Возникающий при сложении двух байтов перенос (8-й бит, выделяемый маской 0x0100) используется в операции сложения следующих двух байтов.

```
//-----39-08.cpp
//-----Сложение целых произвольной разрядности
typedef unsigned char uchar;
void add(uchar out[], uchar in1[], uchar in2[], int n)
{int i, carry; // Бит переноса
 unsigned w; // Рабочая переменная для сложения двух байтов
  for (i=0, carry=0; i<n; i++){
    out[i] = w = in1[i]+in2[i]+carry;
    carry = (w & 0x0100) >>8;
  }}
```

Для того чтобы продемонстрировать работоспособность алгоритма и соответствие его принятым формам представления данных, необходимо взять в качестве параметров функции любой базовый тип (например, **long**), сформировать на него указатель как на область памяти, заполненную беззнаковыми байтами.

```
void main(){
long a=125000, b=30000, c;
add((uchar*)&c, (uchar*)&a, (uchar*)&b, sizeof(long));
printf("c=%ld\n", c);}
```

Для эмуляции *операции вычитания* необходимо сформировать дополнительный код числа, произвести побайтную инверсию и добавить 1 к результату. Последнее можно сделать, установив первоначально в 1 перенос и распространив его по массиву байтов, аналогично сложению.

```
//-----39-09.cpp
//----- Получение отрицательного числа в дополнительном коде
typedef unsigned char uchar;
void neg(uchar in[], int n)
{int i, carry; // Бит переноса
 unsigned w; // Рабочая переменная для сложения двух байтов
  for (i=0; i<n; i++) in[i]=~in[i];
  for (i=0, carry=1; i<n; i++){
    in[i] = w = in[i]+carry;
    carry = (w & 0x0100) >>8;
  }}
```

Для моделирования операции умножения необходимо реализовать операции сдвига на один разряд влево и вправо.

```
//-----39-10.cpp
//-----Сдвиг целых произвольной разрядности
typedef unsigned char uchar;
void lshift(uchar in[], int n)
{ int carry; // Бит переноса
  int i,z;
  for (carry=0, i=0; i<n; i++){
    z=(in[i] & 0x80)>>7; // Выделить старший бит (перенос)
    in[i] <= 1; // Сдвинуть влево и установить
    in[i] |= carry; // старый перенос в младший бит
    carry = z; // Запомнить новый перенос
  }
}
void rshift(uchar in[], int n)
{ int carry; // Бит переноса
  int i,z;
  for (carry=0, i=n-1; i>=0; i--) {
    z = in[i] & 1; // Выделить младший бит (перенос)
    in[i] >= 1; // Сдвинуть вправо и установить
    in[i] |= carry <<7; // старый перенос в старший бит
    carry = z; // Запомнить новый перенос
  }
}
```

В переменной **carry** запоминается значение старшего (младшего) бита, который переносится в следующий байт на место младшего (старшего).

В операции умножения реализован самый простой алгоритм сложения и сдвига. Он, как и все алгоритмы машинной арифметики для внутреннего представления данных, использует свойства

двоичной системы
$$aa \times bb = aa \times \sum_{i=0}^n bb_i \times 2^i = \sum_{i=0}^n bb_i \times aa \times 2^i.$$

В произведении множитель **bb** раскладывается как сумма произведений двоичных разрядов на степени двойки. Известно, что **n**-я степень двойки эквивалентна сдвигу влево на **n** разрядов. Тогда при наличии 1 в **n**-м разряде множителя **bb** к произведению должен быть добавлен множитель **aa**, сдвинутый на **n** разрядов влево. При наличии 0 добавление не производится.

```
//-----39-11.cpp
//-----Умножение целых произвольной разрядности
void mul(uchar out[], uchar aa[], uchar bb[], int n)
{int i;
  for (i=0; i<n; i++) out[i]=0;
  for (i=0; i< n* 8; i++){ // Цикл по количеству битов
    if (bb[0] & 1) // Разряд множителя равен 1
      add(out,out,aa,n); // Добавить множимое к произведению
    lshift(aa,n); // Множимое - влево
    rshift(bb,n); // Множитель - вправо
  }
}
```

В множителе **bb** подряд просматриваются все разряды, начиная с младшего (путем одноразрядного сдвига его вправо). Множимое **aa** при этом каждый раз сдвигается на один разряд влево (умножается на 2). Если очередной разряд множителя **bb** равен 1, то текущее сдвинутое значение множимого добавляется к произведению. Чтобы не усложнять программу, значения множимого и множителя не сохраняются.

Двоично-десятичная арифметика. Существует другая, весьма удобная, хотя и не столь эффективная форма представления данных, позволяющая выполнять арифметические операции произвольной точности. Она основана на представлении данных в десятичной системе счисления, при этом операции производятся над каждой цифрой числа отдельно с учетом взаимного влияния десятичных разрядов через переносы, заемы, то есть так, как «учат в школе». Возможны два варианта представления десятичных цифр:

- отдельная цифра представлена четырьмя битами (тетрадой, шестнадцатеричной цифрой, так называемый **BDC-код**);
- цифра задана символом во внешней форме представления, а само число – текстовой строкой.

Например, число 17665 выглядит в этих формах представления следующим образом:

```
long ss=0x00017655;
char s[]={0x55,0x76,0x10,0x00};
char s[]="17665";
```

В качестве иллюстрации технологии работы с отдельными цифрами числа в десятичной системе счисления рассмотрим пример функции, добавляющей 1 к числу во внешней форме представления, то есть в виде текстовой строки. Добавление 1 состоит в поиске первой цифры, отличной от 9, к которой добавляется 1. Все встречающиеся «на пути» цифры 9 превращаются в 0. Если процесс «превращения девяток» доходит до конца строки, то строка расширяется следующей цифрой 1.

```
//-----39-12.cpp
//----- Инкремент числа во внешней форме представления
void inc(char s[]){
for (int i=0; s[i]!=0; i++);           // Поиск конца строки
for (int n=i-1; n>=0; n--){           // Младшая цифра - в конце
    if (s[n]=='9' )                   // 9 превращается в 0
        s[n]='0';
    else { s[n]++; return; }          // Добавить 1 к цифре и выйти
for (s[i+1]=0; i>0; i--) s[i]='0' ;  // Записать в строку 1000...
s[0]='1';}
```

Другие арифметические операции также моделируются по принципу «цифра за цифрой». Так, при сложении суммируется очередная пара цифр, переведенных во внутреннее представление, и при получении результирующей суммы, превышающей 9, формируется перенос в следующий разряд. Вычитание производится соответственно, с учетом заема.

```
//-----39-13.cpp
//----- Сложение чисел во внешней форме представления
void add(char out[],char c1[],char c2[]){
int l1=strlen(c1)-1;          // Определение разрядности суммы
int l2=strlen(c2)-1;          // и индексов младших цифр слагаемых
int l=l1; if (l2>l1) l=l2;
l++; out[l+1]=0;              // В сумме на 1 цифру больше
int v,v1,v2,carry;            // Цифры и перенос
for (carry=0;l>=0;l--,l1--,l2--){ // Цикл от младших цифр к старшим
if (l1<0) v1=0; else v1=c1[l1]-'0';
if (l2<0) v2=0; else v2=c2[l2]-'0';
v=v1+v2+carry;                // Сложение с учетом входного
if (v>=10) {carry=1; v-=10;} // и формированием выходного
else carry=0;                 // переноса (во внутренней форме)
out[l]=v+'0';                 // Запись цифры результата
}}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

1. Программа деления целых чисел произвольной длины во внутреннем представлении с использованием операций вычитания, инкремента и проверки на знак результата. Частное определяется как количество вычитаний делителя из делимого до появления отрицательного результата (проверить на переменных типа **long**).

2. Программа деления целых чисел произвольной длины во внутреннем представлении с восстановлением остатка. Очередной разряд частного определяется вычитанием делителя из делимого. Если результат положителен, то разряд равен 1, если отрицателен, то делитель добавляется к делимому (восстановление остатка) и разряд частного считается равным 0. После каждого вычитания делимое и частное сдвигаются на один разряд влево. Перед началом операции делитель выравнивается с делимым путем сдвига на $n/2$ разрядов влево.

3. Умножение чисел произвольной длины, представленных непосредственно строками цифр. Первоначально формируется строка символов произведения с необходимым количеством нулей. Далее для каждой пары цифр сомножителей к нему добавляется частичное произведение: значения цифр переводятся во внутреннюю форму и перемножаются, после чего выделяется младшая и стар-

шая цифры результата, которые суммируются с соответствующими цифрами произведения с учетом переноса и его распространения в старшие цифры.

4. Вариант 3 для двоично-десятичного представления исходных данных: в одном байте – две тетрады, хранящие десятичные цифры числа. Последовательность цифр размещена, начиная с младшей, и ограничена тетрадой с кодом **0xF**.

5. Умножение чисел произвольной длины, представленных непосредственно строками цифр. Произведение формируется через многократное сложение одного из множителей с накапливаемым произведением, количество сложений определяется вторым сомножителем.

6. Вариант 5 для двоично-десятичного представления исходных данных: в одном байте – две тетрады, хранящие десятичные цифры числа. Последовательность цифр размещена, начиная с младшей, и ограничена тетрадой с кодом **0xF**.

7. Вычитание чисел произвольной длины, представленных непосредственно строками цифр с использованием дополнительного кода вычитаемого (в десятичной системе счисления).

8. Вариант 7 для двоично-десятичного представления исходных данных: в одном байте – две тетрады, хранящие десятичные цифры числа. Последовательность цифр размещена, начиная с младшей, и ограничена тетрадой с кодом **0xF**.

9. Кодирование и декодирование строки символов, содержащих цифры, в последовательность битов. Десятичная цифра кодируется четырьмя битами – одной шестнадцатеричной цифрой. Цифра **F** обозначает, что за ней следует байт (две цифры) с кодом символа, отличного от цифры. Разработать функции кодирования и декодирования с определением процента уплотнения.

10. Кодирование и декодирование целых переменных различной размерности. Перед каждым числом размещаются пять битов, определяющие количество битов в следующем за ним целом числе; 00000 – конец последовательности. Разработать функции упаковки и распаковки массива переменных типа **long** с учетом количества значащих битов и с определением коэффициента уплотнения. Пример: **01000 xxxxxxxx 00011 xxx 10000 xxxxxxxxxxxxxxxxx 00000**

11. Кодирование массива, содержащего последовательности одинаковых битов. При обнаружении изменения значения очередного бита по сравнению с предыдущим в последовательность записывается шестиразрядное значение счетчика (**n<6**) длины после-

довательности одинаковых битов; $n=0$ обозначает конец последовательности. Пример (исходная последовательность битов задана справа налево): 0000000011111110000000000000 – 001100 000111 001000 000000

12. Большие латинские буквы упаковываются в виде пятибитных кодов по три символа в 16-разрядное машинное слово (типа **int** или **short**). При этом старший бит устанавливается в 1. Остальные символы упаковываются по одному в целую переменную со значением старшего бита – 0. Разработать функции упаковки и распаковки строки с определением коэффициента уплотнения.

13. Первые 15 наиболее часто встречающихся символов кодируются четырехбитными кодами от 0000 до 1110. Код 1111 означает, что следующие за ним 8 битов кодируют один из остальных символов. Разработать функции упаковки и распаковки строки с определением наиболее часто встречающихся символов и коэффициента уплотнения.

14. Если в последовательности встречается бит 0, то за ним идет трехбитовый код первых 8 наиболее часто встречающихся символов (000...111). За битом 1 следует обычный восьмибитный код остальных символов. Разработать функции упаковки и распаковки строки с определением наиболее часто встречающихся символов и коэффициента уплотнения.

15. Первый наиболее часто встречающийся символ кодируется битом 0. Бит 1 кодирует группу из всех остальных символов. Код 10 кодирует второй по частоте символ, 11 – группу всех остальных и т.д. Разработать функции упаковки и распаковки строки с определением наиболее часто встречающихся символов и коэффициента уплотнения.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определите значения переменных после выполнения поразрядных операций. Учтите, что заданные маски соответствуют восьмеричным или шестнадцатеричным цифрам.

```
//-----39-14.cpp
int i,j; unsigned u1,u2,u; char c;
unsigned char uc; long l;
double d1,d2,d3;
//----- 1
i = 0xFFFF; i ++;
//----- 2
u1 = 5; u2 = -1; u=0;
if (u1 > u2) u++;
```

```

//----- 3
i = 0x01FF; c = i; i = c;
//----- 4
i = 0x01FF; uc = i; i = uc;
//----- 5
d1 = 2.56; d2 = (int)d1 + 1.5;
d3 = (int)(d1 + 1.5);
//----- 6
d1 = 2.56;
i = (d1 - (int)d1) * 10;
//----- 7
int i1=20000,i2=20000,s ; // sizeof(int) равен 2
long s1,s2;
s1 = i1 + i2; s2 = (long)i1 + i2;
if (s1 == s2) s=0; else s=1;
//----- 8
i = 0x5678;
l = (i & ~0x00F0) | 0x0010;
c = (l >> 4) & 0xF + '0';
j = (i & 0xFF0F) | (~i & 0x00F0);
//----- 9
i = 1; j = 2; c = 3;
l = (j > i) + (j==c) << 1 + (i !=c) << 2;
//----- 10
for (l=1,i=0; l >0; l<=1, i++); // sizeof(long)=4
//----- 11
for (l=1,i=0; l !=0; l<=1, i++); // sizeof(long)=4
//----- 12
i = 1; j = 3; c = 2; l = i | (j << 4) | (c << 8 );
c = i << 8; j = j << j;

```

Определите и объясните значение результата операции с объединением (**union**). Объединение (см. раздел 2.8) используется для хранения в одной и той же области памяти элементов в различных формах представления.

```

//-----39-15.cpp
//-----
union x {
char c[4];
unsigned char u[4];
int n[2]; // sizeof(int) = 2
long l; // sizeof(long)= 4
} UNI;
//----- 1
void F1() {
long s; int i;
for (i=0; i<4; i++) UNI.c[i]='0'+i;
s = UNI.l; }
//----- 2
void F2() {
char z; UNI.l = 0x00003130; z = UNI.c[1]; }
//----- 3
void F3() {
long s; char z; UNI.l = 0x0000FFFF;

```

```

z = UNI.c[1]; UNI.c[1]=UNI.c[2]; UNI.c[2]=z;
s = UNI.l; }
//----- 4
void F4() {
long s;
UNI.l = 0x0001FFFF; UNI.n[0] >>=2; s = UNI.l; }
//----- 5
void F5() {
long s;
UNI.l = 0x0001FFFF; UNI.c[1] <=>2; s = UNI.l; }
//----- 6
void F6() {
long s; UNI.l = 0x0001FFFF; UNI.u[1] >>=2;
s = UNI.l; }

```

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Содержательно сформулируйте результат функции, определите, какие свойства машинного слова и какие действия над ним она производит. Вызов функции оформите с формальным параметром – шестнадцатеричной константой и прокомментируйте полученный результат.

Пример оформления тестового задания

```

//-----39-16.cpp
unsigned long F(unsigned long v, int k){
int i; unsigned long s;
for (s=1, i=0; i<k; s<=>1, i++) v = v & ~s;
return v;}
void main(){ printf("F(0x1FFF,5)=%lx\n",F(0x1FFF,5)); }

```

В данном примере необходимо представить процессы перемещения битов по машинному слову. Единственный цикл в программе организует перемещение единичного бита последовательно по разрядам машинного слова *s* справа налево. Число повторений цикла *k* ограничивает этот процесс *k* разрядами. Таким образом, в *s* находится единичная маска, которая пробегает по первым *k* разрядам машинного слова. В теле цикла имеет место поразрядная операция **И** с инвертированной маской, что, как мы знаем, интерпретируется как очистка соответствующего бита. Результат функции – очистка *k* младших битов входной переменной. Для приведенного примера вызова результат легко определить. Естественно, в шестнадцатеричной системе счисления (с раскладкой по битам) – **0x1FE0**. Очищаются первые пять битов – младшая тетрада и еще младший бит второй тетрады.


```

//-----39-17.cpp
//----- 1
int F1(){
int i; long l;
for (l=1,i=0; l >0; l<=1, i++);
return i;}
//----- 2
int F2(){
int i; long l;
for (l=1,i=0; l !=0; l<=1, i++);
return i;}
//----- 3
int F3(long n)
{ int i,s;
for (i=0,s=0; i < sizeof(long) * 8; i++, n >>=1)
if ((n & 0x7)==5) { s++; i+=2; }
return s; }
//----- 4
long F4(long n)
{ int i; long s;
for (i=s=0; i < sizeof(long) * 8; i++)
{ s <=1; s |= n & 1; n >>=1; }
return s; }
//----- 5
long F5(long n, int m1, int m2)
{ long s,x; int i;
for (i=0,x=1,s=n; i < sizeof(long)*8; i++)
{ if (i >=m1 && i <=m2) s |= x; x <=1; }
return s; }
//----- 6
int F6(char c[])
{ int i,s;
for (i=0; c[i] !='\0'; i++)
if (c[i] >='0' && c[i] <='7') break;
for (s=0; c[i] >='0' && c[i] <='7'; i++)
{ s <=3; s |= c[i] & 0x7; }
return s; }
//----- 7
void F7(char c[],long n)
{ int i=sizeof(long)*8/3 +1;
for (c[i--]='\0'; i>=0; i--)
{ c[i] = (n & 0x7) + '0'; n >>=3; }}
//----- 8
// Операция "^" - ИСКЛЮЧАЮЩЕЕ ИЛИ
int F8(long n)
{ int i,m,k;
for (i=m=k=0; i < sizeof(long) * 8; i++, n >>= 1)
if ((n & 1) ^ m)
{ k++; m !=m; }
return k; }
//----- 9
int F9(long n)
{ int i,m,k;
for (i=m=k=0; i < sizeof(long) * 8; i++, n >>= 1)
if (n & 1) k++;
else { if (k > m) m=k; k=0; }

```

```
return m; }
//----- 10
int F10(long v){
for (int i=0; v!=0; i++, v>>=1);
return i;}
```

3.10. ДВОИЧНЫЕ ФАЙЛЫ ПРОИЗВОЛЬНОГО ДОСТУПА

Двоичный файл – неограниченный массив байтов. Двоичный файл назван так по причине того, что данные в нем хранятся в той же (двоичной) форме представления, что и во внутренней памяти компьютера. Подобно тому, как в памяти программы размещаются статические переменные в процессе трансляции и динамические – во время работы программы, так и программа может размещать в таком файле любые переменные и структуры данных. При возможности позиционирования к любой области двоичный файл именуется **двоичным файлом произвольного доступа** и имеет следующие свойства:

- двоичный файл представляет собой неограниченный массив байтов внешней памяти;

- формы представления данных во внутренней памяти компьютера (переменные) и в двоичном файле полностью идентичны;

- программа имеет возможность при помощи функций ввода/вывода копировать любую область файла в любую область памяти без преобразования («байт в байт»). Таким образом, можно разместить в любом месте файла любую переменную из памяти программы в том виде, в каком она присутствует в памяти, и прочитать ее обратно;

- в отличие от памяти программы, которая распределяется частично транслятором (обычные переменные), частично библиотечными функциями (динамические переменные), память в файле распределяется самой программой. Только она определяет способ размещения данных, метод доступа к ним и несет ответственность за корректность этого размещения.

Функции стандартной библиотеки для работы с двоичным файлом. При открытии или создании нового файла необходимо указать режим работы с файлом:

```
// Открыть существующий как двоичный для чтения и записи
FILE *fd; fd = fopen("a.dat","rb+wb");
// Создать новый как двоичный для записи и чтения
fd = fopen("a.dat","wb+");
```

С открытым файлом связано понятие **текущей позиции (позиционера)**. Текущей позицией называется номер байта, начиная с

которого производится очередная операция чтения/записи, что интерпретируется как **адрес переменной в файле**. Другой, часто используемый, термин – **смещение**. При открытии файла текущая позиция устанавливается на начало файла, после чтения/записи порции данных перемещается вперед на размерность этих данных. Для дополнения файла новыми данными необходимо установить текущую позицию на конец файла и выполнить операцию записи. Текущая позиция представляется в программе переменной типа **long**.

Функция **long ftell(FILE *fp)** возвращает текущую позицию в файле. Если по каким-то причинам текущая позиция не определена, функция возвращает **-1L**. Это же самое значение будем использовать в дальнейшем для представления недействительного значения файлового указателя (файловый **NULL**), самостоятельно определив его:

```
#define FNULL -1L
```

Функция **int fseek(FILE *fp, long pos, int mode)** устанавливает текущую позицию в файле на байт с номером **pos**. Параметр **mode** определяет, относительно чего отсчитывается текущая позиция в файле, и имеет символические и числовые значения (установленные в **stdio.h**):

```
#define SEEK_SET 0    // Относительно начала файла
// Начало файла - позиция 0
#define SEEK_CUR 1    // Относительно текущей позиции,
// >0 - вперед, <0 - назад
#define SEEK_END 2    // Относительно конца файла
// (значение pos - отрицательное)
```

Функция **fseek** возвращает значение 0 при успешном позиционировании и **-1 (EOF)** – при ошибке. Получить текущую длину файла можно простым позиционированием:

```
long fsize;
fseek(fd, 0L, SEEK_END);    // Установить позицию на конец файла
fsize = ftell(fd);          // Прочитать значение текущей позиции
```

Функции **fread** и **fwrite** используются для перенесения данных из файла в память программы (чтение) и обратно (запись).

```
int fread (void *buf, int size, int nrec, FILE *fd);
int fwrite (void *buf, int size, int nrec, FILE *fd);
```

Особенность этих функций в том, что для них безразличен (неизвестен) характер структуры данных в той области памяти, в которую осуществляется ввод/вывод (указатель **void* buf**). Функция

fread читает, а функция **fwrite** пишет в файл, начиная с текущей позиции, **nrec** элементов размерностью **size** байтов каждый, возвращая количество успешно прочитанных (записанных) элементов.

Из того, что функции **fread**, **fwrite** копируют данные из памяти в файл без преобразования, «байт в байт», следует естественный способ сохранения в файле переменной любого типа данных, основанный на использовании операции **sizeof** для определения ее размерности (рис. 3.30).

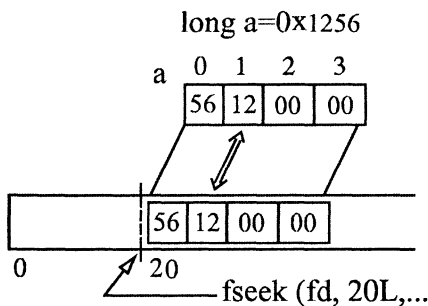


Рис. 3.30

```
// Записать в файл переменную
// типа long, начиная с позиции 20
long    a=0x1256;
fseek (fd, 20L, SEEK_SET);
fwrite ((void*)&a, sizeof(long),1,fd);
// Добавить в файл переменную
// типа man
struct man b;
fseek (fd,0L,SEEK_END);
fwrite ((void*)&b, sizeof b,1,fd);

// Прочитать с начала файла
// динамический массив из n переменных типа double
double *pd = new double[n];
fseek(fd,0L,SEEK_SET);
fread((void*)pd, sizeof(double),n,fd);
```

Позиционирование в текстовом файле. Текстовые файлы являются по своей природе файлами последовательного доступа. Единственное исключение из этого правила – позиционирование (возврат) к уже прочитанному фрагменту текста при помощи функции **fseek**. Но для этого необходимо при первоначальном последовательном просмотре файла определить текущую позицию этого фрагмента в файле функцией **ftell**, вызвав ее перед функцией чтения.

Распределение памяти в двоичном файле. В управлении внутренней памятью на физическом уровне (см. раздел 3.1) и внешней памятью в двоичном файле – много общего. Используя возможности адресной арифметики и преобразования типов указателей, можно произвольным образом планировать память программы, размещая в ней различные переменные. Аналогичная «свобода выбора» имеет место и при работе с файлами: программист произвольно строит в файле любые структуры данных подобно тому, как он это делает в памяти. Но с небольшой разницей: если в памяти программы структуры данных можно организовать, используя обычные переменные языка, динамические переменные, указатели и стандартные операции над ними, то при работе с файлом программист всего этого лишен. Он не может присвоить имя переменной в файле и пользоваться им, не может выполнить над ней никаких операций, кроме как загрузив ее в переменную такого же типа в память программы. Короче говоря, программа вынуждена работать со структурами данных в файле на уровне физических адресов, не имея соответствующей поддержки транслятора.

Способы распределения памяти в файле могут быть довольно сложными. Неиспользуемые (свободные) участки файла должны объединяться в отдельную структуру данных (например, список). Однако существует и простой способ: для размещения переменной в файле достаточно добавить ее в конец файла. Для этого нужно установиться на конец файла и получить значение позиционера, после чего записать в файл саму переменную.

```
int a;  
long pos;  
fseek(fd, 0L, SEEK_END);  
pos = ftell(fd);  
fwrite((void*)&a, sizeof(int), 1, fd);
```

Если в процессе работы с переменной в файле ее размерность не меняется, то можно просто переписывать обновленное значение переменной на то же самое место. В терминологии баз данных такая операция называется обновлением (**Update**).

```
a++;  
fseek(fd, pos, SEEK_SET);  
fwrite((void*)&a, sizeof(int), 1, fd);
```

Если размерность переменной увеличится, то можно еще раз добавить ее в конец файла. Проблема утилизации получающихся свободных мест («сбор мусора») достаточно сложна, чтобы рассматривать ее в простых примерах. В качестве наиболее удобного

решения можно предложить периодическое переписывание всей структуры данных в новый файл (сжатие).

Доступ к данным в файле происходит на физическом уровне, то есть по адресу. Существуют два способа получения адреса:

- адрес вычисляется, исходя из количества и размерности переменных. Простейший случай – файл записей фиксированной длины (массив), адрес записи вычисляется как произведение номера записи на ее размерность;

- адрес содержится в другой части структуры данных, то есть структура данных использует **файловые указатели**.

Терминология, касающаяся двоичных файлов. Двоичные файлы имеют свою историческую терминологию.

Запись – стандартная единица хранения данных в файле «Запись» – это единица хранения, которую получает внешний пользователь, «прикладная» часть структуры данных, находящаяся в файле. Кроме нее, в файле присутствует в том или ином виде «системная» часть, которая обеспечивает упорядоченность, ускоренный поиск и другой необходимый сервис для работы с записями.

Запись фиксированной длины – все записи файла представляют собой переменные одного типа и имеют фиксированную для данного файла размерность. Обычно файл записей фиксированной длины – это массив переменных одного типа.

Запись переменной длины – размерность единицы хранения может меняться от записи к записи. Записями файла могут быть переменные различных типов, либо динамические массивы, либо любые другие структуры данных переменной размерности. Типичной записью переменной длины является строка.

Произвольный доступ – записи файла могут быть прочитаны в любом порядке (вследствие особенностей структуры данных, хранящейся в файле).

Последовательный доступ – файл по своей физической организации (устройство ввода/вывода) или по характеру структуры данных допускает просмотр записей в последовательном порядке. При отсутствии операций позиционирования записи в файле извлекаются в режиме последовательного доступа.

Избыточность в двоичных файлах и защита от ошибок. При работе с файлами возникает специфический род ошибок программы – ошибки формата файла. Дело в том, что при сбое или аварийном завершении программы обычные переменные теряются. Что же касается файлов данных, то в таких ситуациях они остаются в промежуточном состоянии, в котором структура данных в файле

окажется некорректной (например, при выполнении двух последовательных операций программа успевает выполнить только одну из них). Другая причина – программа получает файл не того формата, с которым она работает (вследствие задания неправильного имени файла). Для обнаружения таких ошибок в файл необходимо вносить избыточные данные.

Связанные записи в файле. Файловый указатель. При размещении в файле структур данных с указателями возникает вопрос, каким образом последние будут в нем представлены. Само собой разумеется, что значение указателя, представляющего собой адрес указуемой переменной в памяти программы, не имеет никакого смысла при размещении той же переменной в файле. Тогда каждый указатель, связывающий две переменные в памяти, нужно сопоставить с аналогичным указателем в файле – назовем его **файловым указателем**. Его значением служит позиция (адрес) переменной при ее размещении в файле. Файловый указатель не является типизированным, для всех указуемых объектов он имеет один и тот же тип **long**.

Способы размещения связанных записей в файле. Способ 1. Структура данных записывается в файл «хвостом вперед»: сначала размещаются указуемые переменные с целью получения их адресов в файле, а затем переменные, содержащие указатели. В структурированной переменной, указатель **ptr** продублирован файловым указателем **fptr** (рис. 3.31):

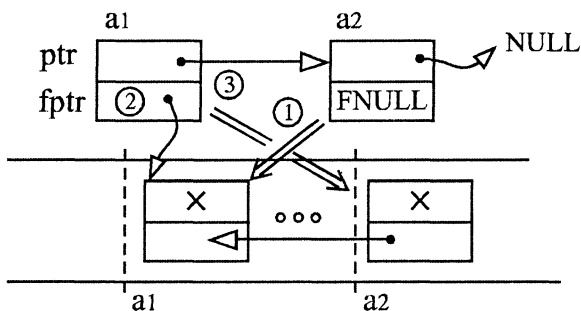


Рис. 3.31

– если указуемая переменная (**a2**) еще не размещена в файле, то необходимо позиционироваться на конец файла, получить текущую позицию как значение ее адреса в файле и записать переменную в файл (1) (рис. 3.31). Если указуемая переменная уже размещена в файле, то просто используется адрес ее размещения;

- полученный адрес указываемой переменной (**a2**) в файле необходимо сохранить как значение файлового указателя (**fptr**) в переменной, содержащей обычный указатель (**ptr** в **a1**) (2);
- сохранить переменную (**a2**) в файле (3).

В принципе цепочка связанных записей может сохраняться вообще без позиционирования (в режиме последовательного доступа).

```
#define FNULL -1L
struct x {
    int    val;
    // Указатель в памяти
    x      *ptr;
    // Файловый указатель
    long   fptr;
a2 = {0,NULL,FNULL},
a1 = {1, &a2, FNULL};

fseek(fd, 0L, SEEK_END);      // Разместить в файле указываемую
a1.fptr = ftell(fd);          // переменную и сохранить ее адрес
fwrite((void*)a1->ptr, sizeof(x), 1, fd);

fseek(fd, 0L, SEEK_END);      // Разместить в файле переменную,
fwrite((void*)&a1, sizeof(x), 1, fd); // содержащую файлового указателя
```

Способ 2. Запись структуры данных естественным образом, «головой вперед» (рис. 3.32):

- разместить в файле все переменные структуры данных и запомнить их адреса в файле (1,2);
- сформировать значения файловых указателей в переменных структуры данных, расположенных в памяти (3);
- «обновить» значения переменных структуры данных в файле, то есть переписать их из памяти по тем же файловым адресам (4).

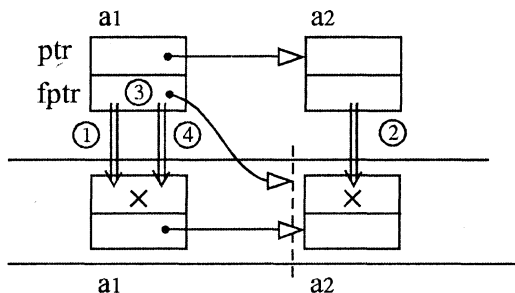


Рис. 3.32

```
fseek(fd, 0L, SEEK_END);
long pp1 = ftell(fd);
fwrite((void*)&a1, sizeof(x), 1, fd);

fseek(fd, 0L, SEEK_END);
long pp2 = ftell(fd);
```



```

fwrite((void*)&a2, sizeof(x), 1, fd);
a1.fprt=pp2;
fseek(fd, pp1, SEEK_SET);
fwrite((void*)&a1, sizeof(x), 1, fd);

```

Такой алгоритм единственно возможен в структуре данных с циклическим ссылками (например, циклический список или граф).

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Повторяющиеся фрагменты в тексте. Исходный текстовый файл может содержать вложенные друг в друга фрагменты вида (12 ...), ограниченные скобками, включающими в себя произвольный текст и счетчик его повторений в виде целой константы (строки цифр). Требуется сгенерировать выходной текст, «раскрыв» повторения.

Вложенные фрагменты определяют рекурсивный характер программы. Каждый фрагмент должен обрабатываться отдельным вызовом рекурсивной функции. Для устранения проблем, связанных с хранением повторяющегося фрагмента произвольной длины, предлагается запомнить начальную позицию фрагмента в файле и перечитывать его при циклическом выводе. Начальной точкой рекурсии удобнее всего считать обнаружение открывающейся скобки в текущем потоке (то есть при вызове она считается уже прочитанной).

```

//-----310-00.cpp
void more(FILE *fd){
long pp;           // Текущая позиция фрагмента повторения
char c; int n=0;   // Количество повторов
while(1){
    pp=ftell(fd);   // Запомнить текущую позицию
    char c=getc(fd); //
    if (!isdigit(c)) break; //
    n=n*10+c-'0';   // Накопление константы
}
if (n==0) n=1;     // Отсутствие константы - повторить 1 раз
while(n--!=0){     // Повторять фрагмент
    fseek(fd,pp,SEEK_SET); // Вернуться на начало
    while ((c=getc(fd)) !=EOF && c!='('){
        if (c=='(') more(fd); // Вложенный фрагмент -
        else                 // рекурсивный вызов после '('
            putchar(c);       // Перечитать фрагмент до '('
    }
}
}

void main(){
FILE *fd=fopen("d310-00.txt","r"); more(fd); fclose(fd);
}

```

Из **main** функция вызывается при установленной начальной позиции файла, что по умолчанию определяет однократный просмотр его содержимого. В этом случае признаком завершения фрагмента служит конец файла (EOF).

Постраничный просмотр текста. Для просмотра текста в произвольном порядке необходимо предварительно последовательно прочитать файл, сделав «закладки» в нужных местах, в нашем случае – запомнить адреса в файле каждой страницы текста, вызвав функцию **ftell** перед чтением очередной двадцатки строк.

```
//-----310-01.cpp
//----- Вывод текста с заданной страницы
FILE *fd; char name[30]="d310.txt" , str[80];
int i,n,NP;           // Количество страниц в файле
long POS[100];       // Массив указателей начала страниц в файле
void main() {
    if ((fd=fopen(name,"r"))==NULL) return;
    for (NP=0; NP<100; NP++){ // Просмотр страниц файла
        POS[NP]=ftell(fd);    // Запомнить начало страницы
        for (i=0; i<20; i++)  // Чтение строк страницы
            if (fgets(str,80,fd)==NULL)
                break;         // Конец файла - выход из цикла
        if (i < 20) break;     // Неполная страница - выход
    }
    while(1){
        printf("page number:");
        scanf("%d",&n);
        if ((n >= NP) || (n <0)) break;
        fseek(fd,POS[n],SEEK_SET); // Позиционироваться на страницу
        for (i=0; i<20; i++) {    // Повторное чтение страницы
            if (fgets(str,80,fd)==NULL) break;
            puts(str);
        }
    }
}
```

Файл записей фиксированной длины. Структура файла записей фиксированной длины – обычный массив переменных одного типа. Соответственно записи в нем последовательно нумеруются, начиная с 0, номер записи является индексом массива, смещение (адрес) записи с номером **n** определяется как **n * sizeof (тип данных записи)**. Количество записей в файле определяется делением его размера на размер записи. Структура файла может быть усложнена ради большей его универсальности и устойчивости к ошибкам выполнения операций ввода-вывода и программирования. Например, в начало файла можно поместить переменные, содержащие размер записи и количество их в файле (рис. 3.33).

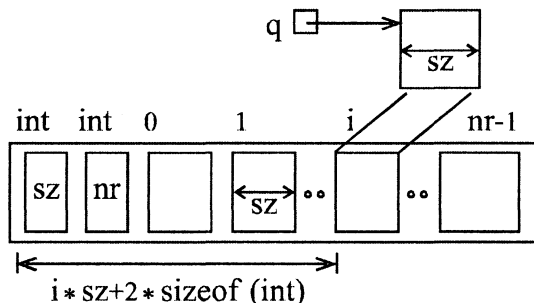


Рис. 3.33

```
//-----310-02.cpp
//----- Функции для работы с файлом записей фиксированной длины
// Создать пустой файл
int Create(char *name, int sz) {
    FILE *fd;
    if ((fd=fopen(name,"wb"))==NULL)    // Создать новый для записи
        return 0;
    int nr=0;
    fwrite((void*)&sz,sizeof(int),1,fd);    // Записать в файл прес и size
    fwrite((void*)&nr,sizeof(int),1,fd);
    fclose(fd); return 1; }
//----- Читать запись с заданным номером из открытого файла
void *Get(FILE *fd, int i){
    int nr,sz;
    fseek(fd,0L,SEEK_SET);
    fread((void*)&sz,sizeof(int),1,fd);    // Читать из файла nr и sz
    fread((void*)&nr,sizeof(int),1,fd);
    fseek(fd,0L,SEEK_END);    // Соответствует ли длина файла
    if (ftell(fd)!=2*sizeof(int)+(long)nr*sz)    // значениям прес и size?
        return NULL;
    if (i >= nr) return NULL;    // Номер записи некорректен
    void *q = ( void*) new char [sz];    // Выделить память
    if (fseek(fd, 2*sizeof(int) + i*sz, SEEK_SET) ==EOF)
        { delete q; return NULL; }    // Ошибка позиционирования
    if (fread(q, sz, 1, fd) !=1)
        { delete q; return NULL; }    // Ошибка чтения
    return q; }
//----- Добавить запись
int Append(FILE *fd, void *pp){
    int nr,sz;
    fseek(fd,0L,SEEK_SET);
    fread((void*)&sz,sizeof(int),1,fd);    // Читать из файла nr и sz
    fread((void*)&nr,sizeof(int),1,fd);
    fseek(fd,0L,SEEK_END);    // Установиться на конец файла
    if (fwrite(pp,sz,1,fd)!=1) return 0;    // Добавить запись
    nr++;
    fseek(fd,sizeof( int),SEEK_SET);    // Обновить переменную nr в файле
    if (fwrite((void*)&nr,sizeof(int),1,fd)!=1)
        return 0;    // Ошибка
    return 1; }
```

Файл записей переменной длины. Запись переменной длины (ЗПД) – единица хранения, меняющая свою размерность в различных экземплярах. В формате записей переменной длины могут храниться:

- динамические массивы. Типичный пример – строка – запись переменной длины;
- последовательности переменных различных типов, определяемые форматом (см. раздел 3.1).

Имеется два способа хранения записей переменной длины в файле:

- используется специальное значение или код – ограничитель записи. Типичным примером является строка текста в памяти, имеющая в качестве ограничителя символ '\0', который в файле превращается в последовательность символов '\r' и '\n'. В этом смысле обычный текстовый файл при работе с ним построчно – это файл записей переменной длины (рис. 3.34);

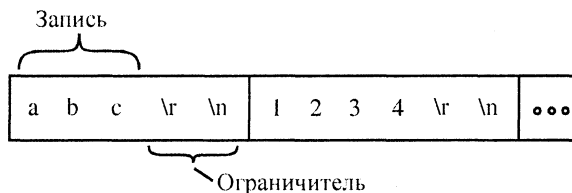


Рис. 3.34

- запись предваряется переменной-счетчиком, который содержит длину этой записи. Содержимое записи может быть любым (прозрачность), поскольку явно выделенные коды-ограничители отсутствуют (рис. 3.35).

Интерпретация содержимого записи никак не связана со способом ее хранения. В следующем примере структурированная переменная (фиксированная размерность) и связанная с ней строка (переменная размерность) хранятся в единой записи (рис. 3.36).

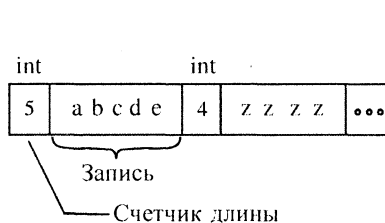


Рис. 3.35

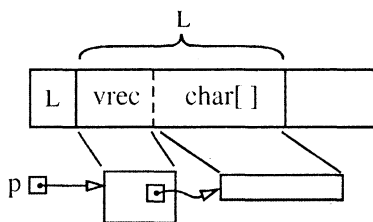


Рис. 3.36

```
//-----310-03.cpp
//----- Структура + строка = запись переменной длины
struct vrec {
    int dd,mm,yy;
    char name[20];           // Строка фиксированной длины
    char *addr;              // Строка переменной длины

    vrec *get(FILE *fd)
    { int size; vrec *p;
      fread(&size,sizeof(int),1,fd);           // Чтение счетчика длины
      if (size == 0) return NULL;              // EOF
      if (size < sizeof(vrec)) return NULL;     // Короткая запись
      if ((p = new vrec) == NULL) return NULL;
      fread((void*)p, sizeof(vrec),1,fd);       // Постоянная часть записи
      size -= sizeof( vrec);                     // Остаток записи - строка
      p->addr = new char[size];
      fread((void*)p->addr,size,1,fd);          // Переменная часть записи
      return p; }
}
```

По своей природе такой файл является файлом последовательного доступа, поскольку определить адрес любой записи без знания размерности всех предыдущих не представляется возможным.

Параметризованные файлы записей фиксированной длины. Структуру файла записей фиксированной длины можно усложнить, если сделать размерности записей зависимыми от параметров, которые можно хранить в том же файле, но «ближе к началу». Тогда получаем структуру данных с варьируемой от файла к файлу размерностью. Работа с таким файлом происходит по принципу «раскрутки»: читаются параметры, определяющие размерность следующего компонента, в котором находятся новые параметры, и т.д. В качестве примера приведем фрагмент программы, работающей с файлом-таблицей с произвольным количеством и типами столбцов. Файл содержит целые переменные – количество столбцов и строк, затем соответствующее количество структурированных переменных – описателей столбцов, а уж потом сами строки таблицы (рис. 3.37).

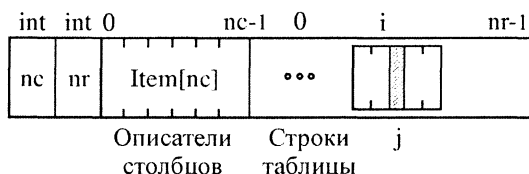


Рис. 3.37

Описатели столбцов – записи фиксированной длины. То же самое представляют собой строки таблицы. Последовательность «раскрутки» структур данных файла:

- читаются размерности таблицы – количество столбцов **nc** и строк **nr**;
- создается и читается динамический массив описателей столбцов;
- вычисляется начальный адрес области строк в файле **adata=2*sizeof(int)+nc*sizeof(cDef)**;
- вычисляется размерность строки (**lr**) как сумма размерностей столбцов;
- вычисляется адрес **i**-й строки для записей фиксированной длины **adata+lr*i**;
- вычисляется смещение **j**-го столбца в строке как сумма размерностей столбцов от 0 до **j-1**-го.

```
//-----310-04.cpp
//---- Файл - таблица произвольной размерности
struct cDef {           // Описатель столбца
int type;               // Тип столбца
int size;               // Размерность столбца в байтах
char name[30]; };       // Имя столбца

FILE *fd;               // Дескриптор файла
int nc;                 // Количество столбцов
int nr;                 // Количество строк
int lr;                 // Размер строки таблицы
long adata;             // Начальный адрес области строк
cDef *ST;               // Динамический массив описателей

//----- Открыть файл и прочитать описатели столбцов
int OpenTable(char *name)
{ int i;
if ((fd=fopen(name,"rb"))==NULL) return 0;
fread(&nc,sizeof(int),1,fd);      // Чтение nc
fread(&nr,sizeof(int),1,fd);      // Чтение nr
ST= new cDef[nc];                 // Память под массив описателей
fread(ST,sizeof(cDef),nc,fd);     // Чтение массива описателей
adata=sizeof(int)*2 + sizeof(cDef) * nc; // Определение adata
for (i=0,lr=0; i<nc; i++)         // Определение длины строки
lr += ST[i].size;
return 1;}

//----- Чтение элемента таблицы из столбца j строки i
void *GetItem( char *name, int i, int j)
{
if (!OpenTable(name)) return NULL;
if (nc <=j || nr <=i) return NULL;
for ( int k=0,lr=0; k<j; k++)      // Смещение j-го столбца в строке
lr += ST[k].size;
void *data = (void*)new char[ST[j].size]; // Память под ячейку таблицы
fseek(fd, adata + i*lr + lr, SEEK_SET);  // Адрес ячейки таблицы в файле
fread(data,ST[j].size,1,fd);
return data;}
```

Файловые указатели. Создание структуры данных в файле.
 Чтобы начать работать со структурой данных в файле, нужен не

просто «пустой» файл, а файл, содержащий некоторое начальное состояние структуры данных. Двоичный файл, содержащий массив указателей на строки – записи переменной длины, создается и первоначально заполняется строками из заданного текстового файла. В начале файла размещаются размерность массива указателей и его смещение (начальный адрес). Это сделано для того, чтобы при последующем добавлении строк в файл и переполнении массива указателей его можно было перезаписывать в конец файла с увеличением размерности. Строки хранятся в формате записей переменной длины со счетчиком (рис. 3.38).

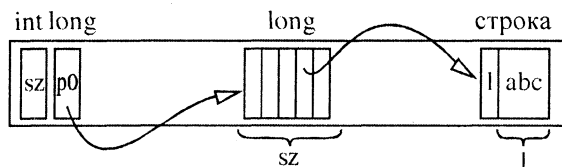


Рис. 3.38

Массиву указателей в файле соответствует аналогичный динамический массив этих же указателей в памяти, который сначала формируется, а затем уже записывается в файл. Входной текстовый файл прочитывается два раза, первый раз – для определения количества строк и размерности массива указателей, второй раз – для формирования структуры данных. Массив файловых указателей пишется в двоичный файл также два раза: в первый раз – чтобы «занять место», а во второй раз – чтобы записать сформированные адреса строк (обновление).

```
//-----310-05.cpp
//---- Создание файла с массивом указателей из текстового файла
void save(char *in, char *out)
{ FILE *fdi,*fdo; char c[80];
  if ((fdi=fopen(in,"r"))==NULL) return;
  if ((fdo=fopen(out,"wb"))==NULL) return;
  for (int ns=0; fgetc(c,80,fdi)!=NULL; ns++); // Количество строк
  fseek(fdi,0L,SEEK_SET); // Вернуться к началу
  long *pp = new long[ns]; // Массив файловых указателей
  long p0=sizeof(int)+sizeof(long); // Начальное смещение МУ
  fwrite(&ns,sizeof(int),1,fdo); // Записать размерность МУ
  fwrite(&p0,sizeof(long),1,fdo); // Записать смещение МУ
  fwrite(pp,sizeof(long),ns,fdo); // Записать "пустой" МУ
  for (int i=0; i<ns; i++) { // Повторное чтение строк
    pp[i]=ftell(fdo); // Получить адрес i-й строки
    fgetc(c,80,fdi);
    int sz=strlen(c)+1; // Переписать в формате ЗПД
    fwrite(&sz,sizeof(int),1,fdo);
    fwrite(c,sz,1,fdo);
  }
}
```

```

    }
    fseek(fdo,p0,SEEK_SET);           // Обновить в файле массив
    fwrite(pp,sizeof(long),ns,fdo);   // файловых указателей
    fclose(fdo);}

```

В программе присутствует единственное позиционирование – перед обновлением массива файловых указателей. Все остальные данные (размерность, смещение, массив указателей, сами строки) сохраняются последовательно.

Файловые указатели. Загрузка массива указателей на строки. Функция загрузки иллюстрирует тот факт, что при переменной размерности она должна полностью создаваться в динамической памяти. Происходит это в два этапа. Сначала создается и загружается массив файловых указателей, для которого создается аналогичный массив указателей на строки, но уже в памяти. Затем читаются сами строки.

```

//-----310-06.cpp
//----- Загрузка массива указателей на строки из двоичного файла
char **load(char *name)           // Функция возвращает динамический
{ FILE *fd; int i,n;              // массив указателей на строки
  long *pp,p0;                   // Динамический массив файловых указателей
  char **p;                      // Динамический МУ на строки
  if ((fd=fopen(name,"rb"))==NULL) return NULL;
  fread(&n,sizeof(int),1,fd);     // Прочитать размерность
  fread(&p0,sizeof(long),1,fd);   // и смещение МУ
  pp=new long[n];                // Создать динамический массив
  p=new char*[n+1];              // файловых указателей и указателей
  fseek(fd,p0,SEEK_SET);         // на строки.
  fread(pp,sizeof(long),n,fd);   // Читать массив файловых указателей
  for (i=0; i<n; i++)
  { int sz;
    fseek(fd,pp[i],SEEK_SET);     // Установиться по i-му файлово-
    fread(&sz,sizeof(int),1,fd); // указателю и прочитать запись
    p[i]=new char[sz];           // переменной длины - строку
    fread(p[i],sz,1,fd);
  }
  p[n]=NULL; fclose(fd); return p;}

```

Связанные записи в файле. Сохранение дерева в файле. Вершина дерева в памяти и в файле представлена одной и той же структурированной переменной, которая содержит указатели на потомки как в памяти, так и в файле. В памяти с вершиной дерева связана строка, хранящаяся в динамическом массиве, которая в файле представлена записью переменной длины. В структурированной переменной находится счетчик записи, сама запись размещается непосредственно за вершиной. Дерево записывается в файл «потомками вперед», причем в режиме последовательного доступа без позиционирования. В начале файла размещается указатель на корневую вершину.


```
//-----310-07.cpp
//---- Сохранение дерева в файле "хвостом вперед"
struct ftree {
    char *str;                // Строка в памяти
    ftree *p[4];              // Указатели на потомков в памяти
    long fp[4];               // Указатели на потомков в файле
    int sz; };                // Длина строки в файле (ЗПД)

#define FNULL -1L
#define TSZ sizeof(ftree)

//--- Функция записи возвращает адрес размещенной вершины в
// файле
long PutTree(ftree *q, FILE *fd)
{ long pos;
  if (q == NULL) return FNULL;
  for (int i=0; i<4; i++)    // Рекурсивное сохранение потомков
    q->fp[i] = PutTree(q->p[i],fd);
  pos = ftell(fd);          // Адрес вершины
  q->sz=strlen(q->str)+1;    // Длина строки (ЗПД)
  fwrite(q, TSZ, 1, fd);    // Сохранить вершину
  fwrite(q->str, q->sz, 1, fd); // Сохранить строку
  return pos;
}
// В начало файла записывается указатель на головную
// вершину дерева
void SaveTree(ftree *p, char *name)
{ FILE *fd; long pos0;      // Указатель на корневую вершину
  if ((fd=fopen(name,"wb")) ==NULL) return;
  fwrite(&pos0, sizeof(long), 1, fd); // Резервировать место под указатель
  pos0 = PutTree(p,fd);      // Сохранить дерево
  fseek(fd, 0L, SEEK_SET);   // Обновить указатель
  fwrite( (void*)&pos0, sizeof(long), 1, fd); fclose(fd); }
```

Более естествен вариант, когда вершина дерева в памяти и в файле представлена различными структурированными переменными. Тогда дерево в файле и в памяти не будет содержать лишних данных. Для формирования текущей вершины в файле рекурсивной функции достаточно иметь локальную переменную (в каждом вызове – свою). Рекурсивная функция записи сначала размещает текущую вершину и запоминает ее адрес в файле. Затем вызывает самое себя для размещения потомков. Полученные после размещения файловые указатели запоминаются в текущей вершине, после чего вершина «обновляется» в файле.

```
//-----310-08.cpp
//---- Сохранение дерева в файле "головой вперед"
struct tree {
    char *str;                // Строка в памяти
    tree *p[4];              // Указатели на потомков в памяти
    struct ftree {            // Вершина дерева в файле
        long fp[4];          // Указатели на потомков в файле
        int sz; };           // Длина строки в файле (ЗПД)
};
#define FNULL -1L
```

```

long PutTree(tree * q, FILE *fd)
{ long cpos; // Адрес в файле текущей вершины
  ftree A; // Текущая вершина в файле
  if ( q==NULL) return FNULL;
  fseek(fd, 0L, SEEK_END);
  cpos = ftell(fd); // Сохранить адрес текущей вершины
  A.sz=strlen(q->str)+1; // Записать в файл текущую
  fwrite(&A, sizeof(ftree), 1, fd);
  fwrite(q->str, A.sz, 1, fd); // вершину и строку
  for (int i=0; i<4; i++) // Рекурсивное сохранение потомков
    A.fp[i] = PutTree(q->p[i],fd);
  fseek(fd, cpos, SEEK_SET); // Обновить текущую вершину
  fwrite((void*)&A, sizeof( ftree), 1, fd);
  return cpos; }

void SaveTree(tree *p, char *name)
{ FILE *fd; long pos0; // Указатель на корневую вершину
  if ((fd=fopen(name,"wb")) ==NULL) return;
  fwrite(&pos0, sizeof(long), 1, fd); // Резервировать место под указатель
  pos0 = PutTree(p,fd); // Сохранить дерево
  fseek(fd, 0L, SEEK_SET); // Обновить указатель
  fwrite( (void*)&pos0, sizeof(long), 1, fd); fclose(fd); }

```

Связанные записи в файле. Загрузка дерева из файла. Последовательность действий по чтению структуры данных со связанными записями более простая: по имеющемуся адресу из файла читается переменная, из которой берутся значения файловых указателей на другие переменные, и процесс повторяется. Структура данных в памяти формируется, как правило, с использованием динамических переменных. В качестве примера рассмотрим загрузку дерева из файла, сформированного вторым способом.

```

//-----310-09.cpp
//----- Загрузка вершины дерева и потомков из файла
tree *GetTree(long pos, FILE *fd) // Вход - адрес вершины в файле
{ if (pos == FNULL) return NULL; // Результат - указатель на
  tree *q=new tree; // вершину поддерева в памяти
  ftree A; // Текущая вершина из файла -
  fseek(fd,pos,SEEK_SET); // в локальной переменной
  fread((void *) &A, sizeof( ftree), 1, fd);
  q->str=new char[A.sz]; // Загрузка строки - ЗПД
  fread(q->str, A.sz, 1, fd);
  for (int i=0; i<4; i++) // Рекурсивная загрузка потомков
    q->p[i]=GetTree(A.fp[i],fd); // и сохранение указателей
  return q; }
// В начале файла читается файловый указатель
// на головную вершину дерева
tree *LoadTree(char *name)
{ FILE *fd; long phead;
  if ((fd = fopen(name,"rb")) ==NULL) return NULL;
  fread((void*)&phead, sizeof(long), 1, fd);
  return GetTree(phead, fd); }

```

Поэлементная загрузка структур данных. Двоичное дерево. Довольно часто требуется не вся структура данных, а лишь отдельные ее переменные, либо структура данных настолько велика,

что не может быть размещена в памяти. Тогда используется более «изысканный» способ работы: в локальные или динамические переменные загружаются только те элементы, которые используются в процессе поиска или просто «движения» по структуре данных. Сложность этого способа состоит в том, что структура данных в памяти уже не соответствует структуре данных в файле (или соответствует фрагментарно). В качестве примера рассмотрим функцию поиска в двоичном дереве элемента с указанным значением. В процессе работы рекурсивной функции происходит загрузка только той цепочки вершин дерева, по которой производится поиск. Текущая вершина загружается в локальную переменную, строка – в динамический массив.

```
//-----310-10.cpp
//----- Поиск в двоичном дереве по образцу с поэлементной загрузкой
struct ftree {                                // Вершина дерева в файле
long fl,fr;                                  // Указатели на потомков в файле
int sz; };                                   // Длина строки в файле (ЗПД)

char *FindTree(long pos, char *key, FILE *fd)
{ ftree A; char *str;
if (pos == FNULL) return NULL;
fseek(fd, pos, SEEK_SET);
fread(&A, sizeof( ftree), 1, fd);
str = new char[A.sz];                        // Чтение строки в динамический массив
fread(str, A.sz, 1, fd);
if ( strncmp(str,key,strlen(key))==0)
    return str;                             // Совпадение с образцом
char *pNext;                                 // Найденная строка от потомка
if ( strcmp(str,key) > 0)
    pNext = FindTree(A.fl, key, fd);
else
    pNext = FindTree(A.fr, key, fd);
delete str;                                 // Уничтожить текущую строку
return pNext; }                             // и вернуть строку потомка
```

Аналогичная схема имеет место, когда изменяется структура данных. Если это связано с изменением связей между элементами структуры, то переменные, в которых изменяются значения файловых указателей, необходимо «обновлять» в файле. *Замечание:* функция добавления вершины в дерево не работает с пустым деревом, поэтому этот случай необходимо рассматривать отдельно перед ее вызовом.

```
//-----310-11.cpp
//---- Добавление новой вершины в дерево в двоичном файле
long AppendOne( char *str, FILE *fd)
{ long pos;                                 // Добавить в файл новую вершину дерева
ftree Elem;                                // В памяти - автоматическая переменная
Elem.fl = Elem.fr = FNULL;
```

```

Elem.sz = strlen( str ) + 1;
fseek(fd, 0L, SEEK_END);
pos = ftell(fd);
fwrite(&Elem, sizeof( ftree), 1, fd);
fwrite(str, Elem.sz, 1, fd);
return pos; }

void AppendTree(long pos, char *newstr, FILE *fd)
{ ftree A; char *str;
fseek(fd, pos, SEEK_SET);
fread(&A, sizeof(ftree), 1, fd);
str = new char[A.sz]; // Чтение строки в динамический массив
fread(str, A.sz, 1, fd);
    if ( strcmp(str,newstr)>0) {
        if (A.fl !=FNULL)
            { AppendTree(A.fl,newstr,fd); delete str; return; }
        else
            A.fl = AppendOne(newstr,fd);
    }
    else {
        if (A.fr !=FNULL)
            { AppendTree(A.fr,newstr,fd); delete str; return; }
        else
            A.fr = AppendOne(newstr,fd);
    }
}
fseek(fd, pos, SEEK_SET); // Обновить текущую вершину дерева
fwrite(&A, sizeof( ftree), 1, fd);
delete str; }

```

Поэлементная загрузка структур данных. Массив указателей на строки. Строки как записи переменной длины позволяют работать с файлом только в режиме последовательного доступа. Наличие в файле массива указателей позволяет извлекать их в произвольном порядке. В начале файла размещаются размерность массива указателей и его смещение (начальный адрес). Это сделано для того, чтобы при переполнении массива указателей его можно было перезаписывать в конец файла с увеличением размерности. Строки хранятся в формате записей переменной длины со счетчиком. Из файла читаются только данные, необходимые для выполнения текущей операции.

```

//-----310-12.cpp
//---- Массив указателей на строки, чтение по логическому номеру
char *load(char *name, int num) // Возвращается строка =
{ FILE *fd; int i,n,sz; long p0,pp; // динамический массив
if ((fd=fopen(name,"rb"))==NULL)
    return NULL; // Режим чтения двоичного файла
fread(&n,sizeof(int),1,fd); // Считать размерность МУ
fread(&p0,sizeof(long),1,fd); // и его смещение (адрес)
if (num>=n) return NULL; // Нет записи с таким номером
fseek(fd,sizeof(int) + p0+sizeof(long)*num,SEEK_SET);
fread((void*)&pp,sizeof(long),1,fd); // Прочитать указатель с номером n
fseek(fd,pp,SEEK_SET); // Установиться на запись
fread((void*)&sz,sizeof(int),1,fd); // Прочитать длину записи

```

```
char *p=new char[sz];           // Создать динамический массив
fread((void*)p,sz,1,fd);       // Прочитать запись - строку
fclose(fd); return p; }        // Возвратить указатель на строку
```

Обратите внимание на то, что операция позиционирования по переменной **pp** функционально эквивалентна косвенному обращению по указателю (операция «*») при работе с аналогичными структурами данных в памяти.

Позлементная загрузка структур данных. Односвязный список. Наиболее показателен приazoleментной загрузке в память односвязный список. Обратите внимание на полную функциональную аналогию алгоритма работы с односвязным списком в памяти и в файле. Особенности работы с файлом заключаются в том, что для каждого активизируемого элемента структуры данных необходим аналогичный элемент в памяти, а для указателя на него – соответствующий файловый указатель. Так, если для включения в односвязный список с сохранением упорядоченности используется текущий и предыдущий элементы списка, то необходимы две локальные структурированные переменные – текущий и предыдущий элементы списка **cur** и **prev**, а также два файловых указателя, определяющих их расположение в файле, – **fcur** и **fprev**. В начале файла размещается заголовок списка – файловый указатель на первый элемент.

```
//-----310-13.cpp
//----- Односвязный список в файле. Позлементная загрузка.
#define FNULL -1L
struct flist { // Определение элемента списка в файле
int val;      // Значение элемента списка
long fnext;   // Файловый указатель на следующий элемент
};           // Приazoleментной работе flist *next не нужен

void show(FILE *fd) // Просмотр списка
{ flist cur;        // Файловый указатель текущего элемента
long fcur;         // Текущий элемент
fseek(fd,0L,SEEK_SET);
fread(&fcur,sizeof(long),1,fd); // Загрузить указатель на первый
for (; fcur!=FNULL; fcur=cur.fnext) {
fseek(fd,fcur,SEEK_SET); // Загрузка текущего элемента
fread(&cur,sizeof(flist),1,fd);
printf("%d ",cur.val);
} puts(""); }

// Включение с сохранением упорядоченности
void ins_sort(FILE *fd, int vv) {
flist cur,prev,lnew; // Текущий и предыдущий и новый элементы списка
long fnew,fcur,fprev; // Файловые указатели элементов списка
fseek(fd,0L,SEEK_SET);
fread(&fcur,sizeof(long),1,fd);
for (fprev=FNULL; fcur!=FNULL;
fprev=fcur, prev=cur, fcur=cur.fnext)
{ // Переход к следующему
```

```

fseek(fd,fcurl,SEEK_SET); // с запоминанием предыдущего
fread(&cur,sizeof(flist),1,fd); // элемента и его адреса
if (cur.val > vv) break; // Поиск места - текущий > нового
}
lnew.val = vv;
lnew.fnext=fcurl;
fseek(fd,0L,SEEK_END); // Заполнение нового элемента списка
fnew=ftell(fd); // Запись в файл и получение адреса
fwrite(&lnew,sizeof(flist),1,fd);
if (fprev==FNULL) { // Включение первым -
fseek(fd,0L,SEEK_SET); // обновить заголовок
fwrite(&fnew,sizeof(long),1,fd);
}
else { // Включение после предыдущего -
prev.fnext=fnew; // обновить предыдущий
fseek(fd,fprev,SEEK_SET);
fwrite(&prev,sizeof(flist),1,fd);
}}

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ (ТЕКСТОВЫЕ ФАЙЛЫ)

Указанные варианты заданий реализовать с использованием позиционирования указателя в текстовом файле и массива указателей, без загрузки самого текстового файла в память.

1. Сортировка строк файла по длине и по алфавиту и вывод результата в отдельный файл.

2. Программа-интерпретатор текста. Текстовый файл разбит на именованные модули. Каждый модуль может иметь вызовы других текстовых модулей. Требуется вывести текст модуля **main** с включением текстов других модулей в порядке вызова:

```

#aaa
{
Произвольные строки модуля текста aaa
}
#ппп
{
Произвольные строки текста
#aaa // Вызов модуля текста с именем aaa
Произвольные строки текста
}
#main
Основной текст с вызовами других модулей

```

3. Программа – редактор текста с командами удаления, копирования и перестановки строк, с прокруткой текста в обоих направлениях (исходный файл при редактировании не меняется).

4. Программа – интерпретатор текста, включающего фрагменты следующего вида:

```

#repeat 5
Произвольный текст
#end

```

При просмотре файла программа выводит его текст, текст фрагментов «**#repeat – #end**» выводится указанное количество раз. Фрагменты могут быть вложенными.

5. Программа просмотра блочной структуры Си-программы с командами вывода текущего блока, входа в **n**-й по счету вложенный блок и выхода в блок верхнего уровня.

6. Программа построчного сравнения двух файлов с выводом групп строк, вставленных или удаленных из второго файла относительно первого.

7. Программа просмотра текстового файла по предложениям. Предложением считается любая последовательность слов, ограниченная точкой, после которой идет большая буква или конец строки. Программа выводит на экран любой блок с **n**-го по **m**-е предложение.

8. Программа просмотра текстового файла по абзацам. Абзацем считается любая последовательность строк, ограниченная пустой строкой. Программа выводит на экран любой абзац по номеру.

9. Программа составляет словарь терминов. Каждый термин – слово, записанное большими (прописными) буквами. Программа запоминает каждый термин и указатель на строку, в которой он встречается. Кроме того, программа позволяет просматривать текст в обоих направлениях построчно и при выборе текущей строки ищет в ней термин и позиционируется к нему.

10. Программа составляет словарь идентификаторов и служебных слов Си-программы путем запоминания каждого идентификатора и указателя на строку, в которой он встречается. Кроме того, программа позволяет просматривать текст в обоих направлениях построчно и при выборе текущей строки ищет первый идентификатор и позиционируется к строке, где он встречается в первый раз.

11. Программа составляет «оглавление» текстового файла путем поиска и запоминания позиций строк вида «5.7.6. Позиционирование в текстовом файле». Затем программа составляет меню, с помощью которого позиционируется в начало соответствующих разделов и пунктов с прокруткой текста в обоих направлениях.

12. Программа составляет словарь функций Си-программы. Затем программа составляет меню, с помощью которого позиционируется в начало соответствующих функций. Функцию достаточно идентифицировать по фрагменту вида «идентификатор (...», расположенному вне фигурных скобок).

13. Программа – редактор текста с командами изменения (редактирования) строки и прокруткой текста в обоих направлениях (измененные строки добавляются в конец исходного файла, начало файла не меняется).

14. Программа ищет в тексте Си-программы самый внутренний блок (для простоты начало и конец блока располагаются в отдельных строчках), присваивает ему номер и «выкусывает» его из основного текста, заменяя его ссылкой на этот номер. Затем по заданному номеру блока производится его вывод на экран, в тексте блока при этом должна присутствовать строка вида «#БЛОК **nnn**» при наличии вложенного блока. (Процедуру «выкусывания» блоков рекомендуется реализовать при помощи «выкусывания» файловых указателей на строки вложенного блока и замены их на отрицательное число $-n$, где n – номер, присвоенный блоку.)

15. Программа сортировки файла по длине предложений и вывода результата в отдельный файл. При выводе каждое предложение следует переформатировать так, чтобы оно начиналось с отдельной строки и располагалось в строках размером не более 60 символов.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ (ДВОИЧНЫЕ ФАЙЛЫ)

1. Файл записей переменной длины перед каждой записью содержит целое, определяющее ее длину. Написать функции ввода и вывода записи в такой файл. Функция ввода (чтения) должна возвращать размер очередной прочитанной записи. Использовать функции для работы с двумя файлами – строк и динамических массивов целых чисел.

2. Программа создает в файле массив указателей фиксированной размерности на строки текста. Размерность массива находится в начале файла, сами строки также хранятся в файле в виде записей переменной длины. Написать функции чтения/записи строки из файла по заданному номеру.

3. Программа переписывает дерево с ограниченным количеством потомков из памяти в файл записей фиксированной длины, заменяя указатели на вершины номерами записей в файле. Затем выполняет обратную операцию.

4. Дерево представлено в файле записей фиксированной длины естественным образом: если вершина дерева в файле находится в записи под номером N , то ее потомки – под номерами $2N$ и $2N+1$. Корень дерева – запись с номером 1. Написать функции включения в дерево с сохранением упорядоченности и обхода дерева (вывод

упорядоченных записей). (Необходимо учесть, что несуществующие потомки должны быть записями специального вида, например, пустой строкой.)

5. Упорядоченные по возрастанию строки хранятся в файле в виде массива указателей. Написать функции включения строки в файл и вывода упорядоченной последовательности строк (просмотр файла).

6. Для произвольного текстового файла программа составляет файл записей фиксированной длины, содержащий файловые указатели на строки текстового файла. Программа производит логическое удаление, перестановку и сортировку строк, не меняя самого текстового файла.

7. Выполнить вариант 3 применительно к графу, представленному списковой структурой.

8. Составить файл записей фиксированной длины, в котором группы записей связаны в односвязные списки (например, списочный состав студентов различных групп). В начале файла предусмотреть таблицу заголовков списков. Написать функции дополнения и просмотра списка с заданным номером.

9. Создать файл, содержащий массив указателей на строки, представленные записями переменной длины. В начале файла – целая переменная – размерность массива указателей. Последовательность указателей ограничена **NULL**-указателем. Реализовать функции загрузки строки по логическому номеру и добавления строки по логическому номеру.

10. Создать файл, содержащий массив указателей на упорядоченные в алфавитном порядке строки, представленные записями переменной длины. Реализовать функцию двоичного поиска строки по строке-образцу, начало которой совпадает с искомой строкой.

11. В файле записей фиксированной длины содержится двоичное дерево. Вершина содержит переменную типа **int**, а также номера соответствующих записей для правого и левого потомков. Реализовать функцию включения нового значения в существующий файл в виде новой вершины двоичного дерева.

12. Вершина двоичного дерева содержит указатель на строку. Написать функции сохранения и загрузки дерева из файла. Вершина дерева должна содержать файловые указатели на потомков, а также файловый указатель на строку – запись переменной длины.

13. Вершина двоичного дерева содержит указатель на строку. Написать функции сохранения и загрузки дерева из файла. Вершина дерева должна быть записью переменной длины, содержать файловые указатели на потомков и строку.

14. Файл содержит односвязный список. Элемент списка содержит файловый указатель на следующий элемент и строку – запись переменной длины. В начале файла – указатель на первый элемент списка. Реализовать функции просмотра списка и включения строки по номеру.

15. Файл содержит односвязный список. Элемент списка содержит файловый указатель на следующий элемент и строку – запись переменной длины. В начале файла – указатель на первый элемент списка. Реализовать функции просмотра списка и включения строки с сохранением упорядоченности.

ИНДИВИДУАЛЬНЫЕ ПРОЕКТЫ (ТЕКСТОВЫЙ ФАЙЛ)

1. Сортировка текстового файла простым разделением (по длине строк). Файл читается группами по **n** строк в динамический массив указателей на строки, группа сортируется и записывается в промежуточный файл. Имя промежуточного файла генерируется в виде **Fnnnn.txt**, где **nnnn** – номер группы. Затем файлы сливаются по «олимпийской» системе – по два файла в один.

2. Сортировка текстового файла циклическим слиянием/разделением (по длине строк).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определите структуру данных в двоичном файле произвольного доступа анализом последовательности операций ввода/вывода.

Пример оформления тестового задания

Ввиду того, что для вызова большинства функций необходимо иметь двоичный файл соответствующего формата, можно ограничиться анализом текста программы.

```
//-----310-15.cpp
//-----
#define FNULL -1L
struct tree { tree *p[4]; char *s; }; // Вершина дерева в памяти
struct ftree { long fp[4]; int sz; }; // Вершина дерева в файле

tree *F(FILE *fd, long pos)
{ ftree A; // Текущая вершина из файла
  if (pos==FNULL) return NULL; // NULL-указатель - выйти
  tree *q = new tree; // Создать вершину в динам. памяти
  fseek(fd,pos,SEEK_SET); // Позиционироваться на вершину
  fread( (void*)&A, sizeof(ftree),1,fd); // в файле и читать ее
  q->s= new char[A.sz+1]; // Строка - запись переменной длины
```

```

fread(q->s,A.sz,1,fd);           // следует сразу за вершиной
q->s[A.sz]=0;                     // Добавить ограничитель конца строки
for (int i=0; i<4; i++)          // Рекурсивная загрузка потомков
q->p[i]=F(fd,A .fp[i]);
return q; }                      // В начале файла корневая вершина
void main() { FILE *fd=fopen("a.dat","rb"); tree *head = F(fd,0L); }

```

Несомненно, речь идет о дереве, которое строится в памяти, поскольку функция является рекурсивной и возвращает указатель на вершину дерева в памяти. Вершина дерева в памяти содержит до четырех указателей на потомков и указатель на связанную с ней строку. Функция создает в динамической памяти вершину дерева, получая в качестве параметров открытый файл и файловый указатель на место расположения текущей вершины. В файле есть также дерево, фиксированная часть вершины которого представлена структурой **ftree**. Текущая вершина в файле загружается в локальную переменную **A**. Вершина в файле содержит массив файловых указателей на потомков – **fp** и строку, представленную записью переменной длины. В самой структуре **ftree** имеется счетчик длины строки – **sz**. То, что строка читается без позиционирования, говорит о том, что сама строка непосредственно следует за вершиной. В памяти программы строка размещается в отдельном динамическом массиве. В начале файла находится корневая вершина дерева (следует из **main**).

```

//-----310-16.cpp
struct man { int dd,mm,yy; char *addr; };
//----- 1
man *F1(int n, FILE *fd)
{ man *p = new man;
fseek (fd, (long)sizeof( man)*n, SEEK_SET);
fread (p, sizeof( man),1,fd);
return(p); }
//----- 2
void *F2(FILE *fd)
{ int n; void *p;
fread(&n,sizeof(int),1,fd);
if (n==0) return(NULL);
p = ( void*) new char[n];
fread(p,n,1,fd);
return p; }
//----- 3
double *F3(FILE *fd)
{ int n; double *p;
fread(&n,sizeof(int),1,fd);
if (n==0) return(NULL);
p = new double[ n+1];
fread(p,sizeof(double),n,fd);
p[n]=0.0;
return p; }

```

```

//----- 4
#define FNULL -1L
struct xxx { long fnext; /*. . .*/ };
xxx *F4(int n, FILE *fd)
{ xxx *p; long p0;
  p = new xxx;
  fseek(fd, 0L, SEEK_SET);
  fread(&p0, sizeof(long), 1, fd);
  for (; p0!=FNULL && n!=0; n--, p0 = p->fnext) {
    fseek(fd, p0, SEEK_SET);
    fread(p, sizeof( xxx), 1, fd); }
  return p; }
//----- 5
man *F5(int n, FILE *fd)
{ long fp; man *p = new man;
  fseek(fd, sizeof(long)*n, SEEK_SET);
  fread(&fp, sizeof(long), 1, fd);
  fseek(fd, fp, SEEK_SET);
  fread(p, sizeof(man), 1, fd);
  return p; }
//----- 6
void *F6(int n, FILE *fd)
{ int sz; void *p;
  fseek(fd, 0L, SEEK_SET);
  fread(&sz, sizeof(int), 1, fd);
  p = ( void*) new char[sz];
  fseek (fd, (long)sz * n + sizeof(int), SEEK_SET);
  fread (p, sz, 1, fd);
  return p; }
//----- 7
void *F7(int n, FILE *fd)
{ int sz; void *p; long p0;
  fseek(fd, 0L, SEEK_SET);
  fread(&sz, sizeof(int), 1, fd);
  fread(&p0, sizeof(long), 1, fd);
  p = (void*)new char[sz];
  fseek (fd, p0 + sizeof(long)*n, SEEK_SET);
  fread (&p0, sizeof(long), 1, fd);
  fseek(fd, p0, SEEK_SET);
  fread(p, sz, 1, fd);
  return p; }
//----- 8
char *F8(int n, FILE *fd)
{ char *p; long fp; int i;
  fseek(fd, sizeof(long)*n, SEEK_SET);
  fread(&fp, sizeof(long), 1, fd);
  fseek(fd, fp, SEEK_SET);
  n = 80; p = new char [n];
  for (i=0;; i++) {
    if (i==n) p = (char*)realloc(p, n=n*2);
    fread(p+i, 1, 1, fd);
    if (p[i]=='\0') return p;
  }
  return p; }
//----- 9
#define FNULL -1L

```

```

char *F9(int n, FILE *fd)
{ long p0; int sz; char *p;
fseek(fd,0L,SEEK_SET);
fread(&p0,sizeof(long),1,fd);
for (; p0!=FNULL && n!=0; n--) {
    fseek(fd,p0,SEEK_SET);
    fread(&p0,sizeof(long),1,fd);
}
if (p0==FNULL) return(NULL);
fread(&sz,sizeof(int),1,fd);
p = new char[sz+1];
fread(p,sz,1,fd);
p[sz]='\0'; return p;}
//----- 10

char *F10(FILE *fd)
{ int n; char *p;
fread(&n,sizeof(int),1,fd);
if (n==0) return(NULL);
p = new char[n];
fread(p,n,1,fd);
return p; }
//----- 11

void F11(FILE *fd, char *s)
{ int n;
fseek(fd,0L,SEEK_END);
n = strlen(s)+1;
fwrite(&n,sizeof(int),1,fd);
fwrite(s,n,1,fd); }
//----- 12

double *F12(FILE *fd)
{ int n, dn; double *p;
fread(&n,sizeof(int),1,fd);
if (n==0) return(NULL);
dn = n / sizeof(double);
p = new double[ dn+1];
p[0]=dn;
fread(p+1,sizeof(double), dn, fd);
return p; }
//----- 13

void F13(FILE *fd, double *s, int dn)
{ int n;
n = dn * sizeof(double);
fseek(fd,0L,SEEK_END);
fwrite(&n,sizeof(int),1,fd);
fwrite(s,sizeof(double),dn,fd); }
//----- 14

void F14(FILE *fd)
{ int n; void *p;
fread(&n,sizeof(int),1,fd);
if (n==0) return;
p = ( void*) new char[n];
fread(p,n,1,fd);
switch (n) {
case sizeof(int):      printf("%d ",*(int*)p); break;
case sizeof(double):   printf("%lf ",*(double*)p); break;
default:                printf("%s ",(char*)p); break;
} delete p; }

```

```

//----- 15
char *F15(int n, FILE *fd)
{ int m; char *p; long fp;
  fseek(fd, sizeof(long)*n,SEEK_SET);
  fread(&fp,sizeof(long),1,fd);
  fseek(fd,fp,SEEK_SET);
  fread(&m,sizeof(int),1,fd);
  p=new char [m];
  fread(p,m,1,fd);
  return p ; }
//----- 16
char *F16(int n, FILE *fd1, FILE *fd2)
{ long pp; char *q;
  fseek(fd1,n*sizeof(long),SEEK_SET);
  fread(&pp,sizeof(long),1,fd1);
  q = new char[80];
  fseek(fd2,pp,SEEK_SET);
  fgets(q,80,fd2);
  return q; }
//----- 17
char **F17(FILE *fd)
{ int n,m,i; char **p; long *fp;
  fseek(fd, 0L,SEEK_SET);
  fread(&n,sizeof(int),1,fd);
  p = new char *[n+1];
  fp = new long[n];
  fread(fp,sizeof(long),n,fd);
  for (i=0; i<n; i++) {
    fseek(fd, fp[i],SEEK_SET);
    fread(&m,sizeof(int),1,fd);
    p[i]= new char[m];
    fread(p[i],m,1,fd);
  }
  p[n]=NULL; return p ; }
//----- 18
#define FNULL -1L
struct ooo { ooo *p[20]; char *s; long fs; long fp[20]; };
ooo *F18(FILE *fd, long pos)
{ int i,m; ooo *q;
  if (pos==FNULL) return NULL;
  q = new ooo;
  fseek(fd,pos,SEEK_SET);
  fread(q, sizeof(ooo),1,fd);
  fseek(fd,q->fs,SEEK_SET);
  fread(&m,sizeof(int),1,fd);
  q->s= new char[m];
  fread(q->s,m,1,fd);
  for (i=0; i<20; i++) q->p[i]=F18(fd,q->fp[i]);
  return q; }
void main() { FILE *fd; ooo *head = F18(fd,0L); }
//----- 19
man *F19(FILE *fd)
{ man *p; int n;
  fread(&n,sizeof(int),1,fd);
  p = new man;
  fread (p, sizeof(man),1,fd);
  n = n - sizeof(man);

```

```

p->addr = new char[n];
fread(p->addr,n,1,fd);
return p; }
//----- 20
void F20(FILE *fd, man *p)
{ int n = sizeof(man)+strlen(p->addr)+1;
fseek(fd,0L,SEEK_END);
fwrite(&n,sizeof(int),1,fd);
fwrite (p, sizeof(man),1,fd);
n = n - sizeof(man);
fwrite (p->addr, n,1,fd ); }

```



4. ПРОГРАММИСТ «ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ»

ООП – Организация Освобождения
Палестины.

Аббревиатура

Некоторые вещи нам непонятны не потому, что понятия наши слабы, а потому что они не входят в круг наших понятий.

Козьма Прутков

Технология объектно-ориентированного программирования (ООП) по большому счету ставит программиста «с головы на ноги» (или, наоборот, с ног на голову). И дело тут не в замысловатом синтаксисе (чем особенно страдает Си++). Сравнительно легко освоить «эпизодическое ООП», элементы которого присутствуют в любом Бейсике, где популярно объясняется, что такое классы, объекты, свойства и методы на примере стакана молока. Значительно труднее отказаться от уже приобретенной технологии проектирования программы «от функции к функции» и перейти к «тотальному ООП» по принципу «от класса к классу». В психологическом плане самое сложное состоит в том, что программа как бы «расплывается»: вместо стройной конструкции вызывающих друг друга функций появляется множество классов, которые при выполнении отдельных действий (методов) порождают и используют переданные объекты других классов, для которых, в свою очередь, выполняются другие методы, и так до бесконечности. Возникает резонный вопрос: а где все это начинается и чем заканчивается?

Почему же не учить правильной технологии «с пеленок»? Объектно-ориентированное программирование проявляет себя только

в довольно сложных проектах и касается более высокого уровня организации программы. Поэтому для ее понимания нужно иметь опыт восприятия и написания не совсем элементарных программ. На простых примерах здесь нельзя показать, о чем собственно идет речь. Кроме того, чтобы разрабатывать классы, необходимо уметь расписывать методы – обычные функции с использованием обычного структурного программирования. И наконец знание структур данных и базовых алгоритмов необходимо при любой технологии.

В этой главе рассматриваются особенности синтаксиса и механизмы его реализации в той части, которая помогает пониманию технологии ООП. Такие вещи, как ограничение доступа, права доступа при наследовании, дружелюбность, множественное наследование, виртуальные базовые классы упоминаются только по мере необходимости.

4.1. ПРОГРАММИРОВАНИЕ ОБЪЕКТОВ. КОНСТРУКТОРЫ

Методологическое определение класса и объекта. Любая технология – это совокупность знаний, приемов, навыков и инструментов для повышения эффективности работы. Поэтому она опирается не только на достижения науки, но и на практический опыт и здравый смысл. Технология программирования – не исключение: она показывает, как разрабатывать программы быстро, качественно, избегая крупных ошибок, как обеспечить их универсальность и совместимость. Объектно-ориентированное программирование это совокупность понятий (класс, объект, инкапсуляция, полиморфизм, наследование), приемов их использования при проектировании программ, а Си++ – инструмент этой технологии.

Технология ООП прежде всего накладывает ограничения на способы представления данных в программе. Любая программа отражает в них состояние физических предметов либо абстрактных понятий (назовем их **объектами программирования**), для работы с которыми она предназначена. В традиционной технологии варианты представления данных могут быть разными. В худшем случае программист может «равномерно размазать» данные об объекте программирования по всей программе. В противоположность этому все данные об объекте программирования и о его связях с другими объектами можно объединить в одну структурированную переменную. В первом приближении ее можно назвать **объектом**, составляющие ее элементы данных – **свойствами**. Кроме того, с объектом связывается набор действий, иначе назы-

ваемых методами. С точки зрения языка программирования это функции, получающие в качестве обязательного параметра указатель на объект. Технология ООП запрещает работать с объектом иначе, чем через методы, то есть внутренняя структура объекта скрыта от внешнего пользователя. Описание множества однотипных объектов называется **классом**.

Объект – структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии. **Класс** – описание элементов данных (свойств) таких объектов и выполняемых над ними действий (методов).

Технологическое определение класса и объекта. По крайней мере половина содержательного определения класса заключается в структурированном типе. Структурированная переменная – это объект, а ее элементы – свойства объекта. Вторая часть класса – методы, представлена в Си++ элементами-функциями, вводимыми в структурированный тип. Это функции в обычном их понимании, но синтаксически связанные со структурированным типом.

```
//----- 41-01.cpp
struct date {      // Заголовок - определение структурированного типа
    int day,month,year; // Обычные элементы данных
    void NextData();    // Элемент-функция добавления дня
    void PlusData(int); // Элемент-функция добавления n дней
    int TestData();     // Элемент-функция проверки даты
};

static int mm[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
void date::PlusData(int n){ // Элемент-функция добавления n дней
while(n-- !=0) NextData(); }
int date::TestData(){ // Проверка на корректность
if (month ==2 && day==29 && year %4 ==0) return 1;
if (month ==0 || month >12 || day ==0 || day >mm[month])
    return 0;
return 1; }

//----- Следующая дата -----
void date::NextData(){
day++;
if (day <= mm[month]) return;
if (month ==2 && day==29 && year %4 ==0) return;
day=1; month++;
if (month !=13) return;
month=1; year++; }

//----- Основная программа -----
void main(){
date a;
do scanf("%d%d%d",&a.day,&a.month,&a.year);
while(a.TestData() ==0);
a.PlusData(17);
printf("%d-%d-%d\n",a.day,a.month,a.year);
}
```

В определении структуры (**date**) дается прототип функции (заголовков с перечислением типов формальных параметров, например, **void NextData()**).

Определение самой функции дается отдельно, функция имеет полное имя **имя_структуры::имя_функции**, состоящее из имени функции (**NextData**) и имени структурированного типа (**date**). Говорят, что функция (или метод) **NextData** определяется в классе **date**.

Тело функции может быть включено также и в определение структурированного типа (функция **PlusData**), тогда оно следует непосредственно за заголовком функции.

В теле функции неявно вводится формальный параметр с именем **this** – указатель на структурированную переменную, для которой вызывается функция (в нашем примере это будет **date *this**). Эту переменную называют также указателем на текущий объект. Элементы данных и элементы-функции этой структуры доступны через явное или неявное использование этого указателя.

```
this->month = 5;
this->day++;
month = 5;
day++;
```

Элемент-функция вызывается только в паре с некоторым объектом (например, **a.PlusData(17)**). При вызове указателю **this** для структурированной переменной присваивается адрес того объекта, с которым она сейчас работает.

В целом механизм связи объектов и методов довольно «прозрачен». Вот так выглядит тот же самый фрагмент в виде эквивалента на «классическом» Си.

```
// Добавить n дней
void date_PlusData(date *this, int n){
while(n-- !=0) date_NextData(this);
}
// Следующая дата
void date_NextData(date *this){
this->day++;
...
this->month=1;
this->year++;
}
// Основная программа
void main()
{ date a;
...
date_PlusData(&a,17);
}
```

Обратите внимание, что элемент-функция остается алгоритмической частью программы (программного кода), то есть присутствует в одном экземпляре на все объекты класса. Объект как переменная содержит только элементы данных структурированного типа.

Класс как структурированный тип с ограниченным доступом. В отличие от структуры, класс имеет «приватную» (личную) часть, элементы которой доступны только в функциях-элементах класса, и «публичную» (общую) часть, на элементы которой ограничения доступа не накладываются.

Стандартным является размещение элементов данных в личной части, а функций-элементов – в общей части класса. Тогда закрытая личная часть определяет данные объекта, а функции-элементы общей части образуют интерфейс объекта «к внешнему миру» (методы).

Другие варианты размещения элементов данных и функций-элементов в личной и общей части класса встречаются реже:

- элемент данных в общей части класса открыт для внешнего использования как любой элемент обычной структуры;
- функция-элемент личной части класса может быть вызвана только функциями-элементами самого класса и закрыта для внешнего использования.

Таким образом, в первом приближении класс отличается от структуры четко определенным интерфейсом доступа к его элементам. И наоборот, структура – это класс без личной части.

Иногда требуется ввести исключения из правил доступа, когда некоторой функции или классу требуется разрешить доступ к личной части объекта класса. Тогда в определении класса, к объектам которого разрешается такой доступ, должно быть объявление функции или другого класса как «дружественных». Это согласуется с тем принципом, что сам класс определяет права доступа к своим объектам «со стороны».

Объявление дружественной функции представляет собой прототип функции, объявление переопределяемой операции или имя класса, которым разрешается доступ, с ключевым словом **friend** впереди.

```
// Классы и функции, дружественные классу A
class A {
    int x; // Личная часть класса
    ...    // Все «друзья» имеют доступ к x
    friend class B;
```

```
friend void C::fun(A&);
friend void xxx(A&,int);
friend void C::operator+(A&);
};
```

Виды объектов в программе. Объекты класса обладают всеми свойствами переменных, в том числе такими, как область действия и время жизни. Соответственно в программе возможно определение внешних, статических, автоматических и динамических объектов одного класса.

```
class date { ..... };
date a,b;           // Внешние объекты
date *p;            // Указатель на объект
void main(){
    date c,d;        // Автоматические объекты
    p = new date;     // Динамический объект
    delete p;         // Уничтожение динамического объекта
}                    // Уничтожение автоматических объектов
```

Создание и уничтожение объектов. Конструкторы и деструкторы. Создание и уничтожение объектов класса обычно сопровождаются некоторыми действиями (инициализация данных, резервирование памяти, ресурсов и т.д.), которые производятся функциями-элементами специального вида. Элементы-функции, неявно вызываемые при создании и уничтожении объектов класса, называются **конструкторами** и **деструкторами**. Они определяются как элементы-функции с именами, совпадающими с именем класса. Конструкторов для данного класса может быть сколь угодно много, они отличаются формальными параметрами, деструктор же всегда один, он имеет имя, предваренное символом «~». Если конструктор имеет формальные параметры, то в определении переменной-объекта после ее имени должны присутствовать в скобках значения фактических параметров.

Момент вызова конструктора и деструктора определяется временем создания и уничтожения объектов:

- для статических и внешних объектов – конструктор вызывается перед входом в **main**, деструктор – после выхода из **main**. Конструкторы вызываются в порядке определения объектов, деструкторы – в обратном порядке;

- для автоматических объектов – конструктор вызывается при входе в функцию (блок), деструктор – при выходе из него;

- для динамических объектов – конструктор вызывается при выполнении оператора **new**, деструктор – при выполнении оператора **delete**.

В Си++ возможно определение массива объектов класса. При этом конструктор и деструктор автоматически вызываются в цикле для каждого элемента массива и не должны иметь параметров. При выполнении оператора **delete**, кроме указателя на массив объектов, необходимо также указывать его размерность.

```
class date{
    int    day,month,year;
public:
    date(int,int,int); // Конструктор с целыми параметрами
    date();           // Конструктор без параметров
    date(char *);     // Конструктор с параметром-строкой
    ~date();          // Деструктор
};

//-----
dat a("12-12-1990"); // Внешний объект с параметром-строкой
                    // конструктор вызывается перед main()
dat b[10];           // Массив объектов - конструктор без
                    // параметров вызывается перед main()
                    // в цикле для каждого объекта

void xxx(dat &p){
    dat c(12,12);     // Вызывается конструктор dat(int,int,int)
    ...              // для автоматического объекта
}                   // При выходе из функции вызываются
void main()          // деструктор для объекта c
{ int i,n;
  cin << n;
  dat *p = new dat[n]; // Создание массива динамических объектов -
                      // конструктор без параметров неявно
                      // вызывается n раз
  delete [n] p;       // Уничтожение массива динамических объектов -
                      // деструктор неявно вызывается n раз
}                   // Деструкторы для a и b[10] вызываются после
                    // выхода из main()
```

Синтаксическое определение класса и объекта. В Си++ класс обладает синтаксическими свойствами базового типа данных:

- класс определяется как структурированный тип данных (**struct**) и включается в иерархию типов данных программы;
- объекты определяются как переменные класса (локальные, глобальные, динамические);
- возможно переопределение и использование стандартных операций языка, имеющих в качестве операндов объекты класса в виде особых методов в этом классе.

```
struct matrix {
    // Определение структурированного типа matrix и методов,
    // реализующих операции matrix * matrix, matrix * double
};
```

```
matrix a,b;    // Определение переменных -  
double dd;    // объектов класса matrix  
a = a * b;    // Использование переопределенных операций  
b = b * dd * 5.0;
```

Класс – определенный программистом базовый тип данных.
Объект – переменная класса.

Формальное и содержательное использование классов и объектов. Понятно, что можно соблюсти все формальные требования синтаксиса при определении класса и работе с его объектами, но не соответствовать духу технологии ООП. И наоборот, используя «классический» Си, писать почти объектно-ориентированные программы. Перечислим здесь несколько полезных советов:

- данные класса должны быть максимально закрыты, внешний пользователь не должен подозревать, что находится внутри объекта, и, тем более, на него не должны переноситься проблемы, связанные с их размещением, корректностью и т.д.;

- размерность данных объекта должна меняться в максимально возможных пределах, динамические данные предпочтительнее статических, объект сам должен решать проблемы управления динамической памятью;

- интерфейс класса (методы) должен быть максимально универсален, методы должны сочетаться в любых комбинациях, давая широкое разнообразие возможностей работы с объектом;

- содержимое объекта должно быть всегда корректно, за этим прежде всего следят конструкторы и деструктор, другие методы тоже не должны «оставлять после себя» неправильных объектов;

- «друзья» класса должны быть исключением, но не правилом;

- в классе должны решаться проблемы, связанные с возможным копированием объектов или разделением ими общих данных (конструктор копирования).

Объект – замкнутые, логически непротиворечивые, всегда корректные данные с четко определенным универсальным интерфейсом доступа к ним.

Традиционная технология программирования и ООП. Проблема «Что первично – курица или яйцо?» применительно к программированию звучит как «Что первично: алгоритм (процедура, функция) или обрабатываемые им данные». В традиционной технологии программирования взаимоотношения процедуры – данные

имеют более-менее свободный характер, причем процедуры (функции) являются ведущими в этой связке: как правило, функция вызывает функцию, передавая данные друг другу по цепочке. Соответственно, технология структурного проектирования программ прежде всего уделяет внимание разработке алгоритма.

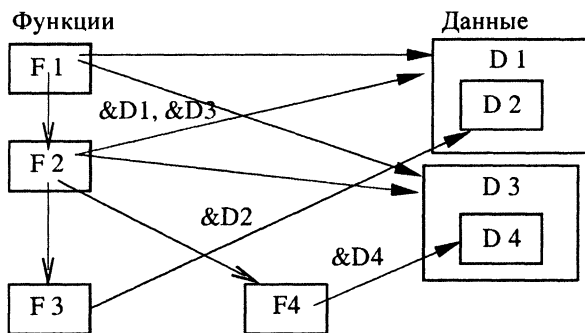


Рис. 4.1

Как видно из рис. 4.1, цепочка вызовов функций – основная в схеме взаимодействия элементов программы, взаимосвязь элементов данных определяется характером передачи параметров между функциями. Поэтому традиционную технологию программирования можно назвать программированием «от функции к функции». В технологии ООП взаимоотношения данных и алгоритма имеют более регулярный характер (рис. 4.2): во-первых, класс объединяет в себе данные (структурированная переменная) и методы (функции). Во-вторых, схема взаимодействия функций и данных принципиально иная. Метод (функция), вызываемый для одного объекта, как правило, не вызывает другую функцию непосредственно. Для начала он должен иметь доступ к другому объекту (создать, получить указатель, использовать внутренний объект в текущем и т.д.), после чего он уже может вызвать для него один из известных методов. Следовательно, структура программы определяется взаимодействием объектов различных классов. Как правило, имеет место иерархия классов, а технология ООП иначе может быть названа как программирование «от класса к классу».

Традиционная технология программирования «от функции к функции» определяет первичность алгоритма (процедур, функций) по отношению к структурам данных. Технология ООП определяет первичность данных (объектов) по отношению к алгоритмам их обработки (методам).

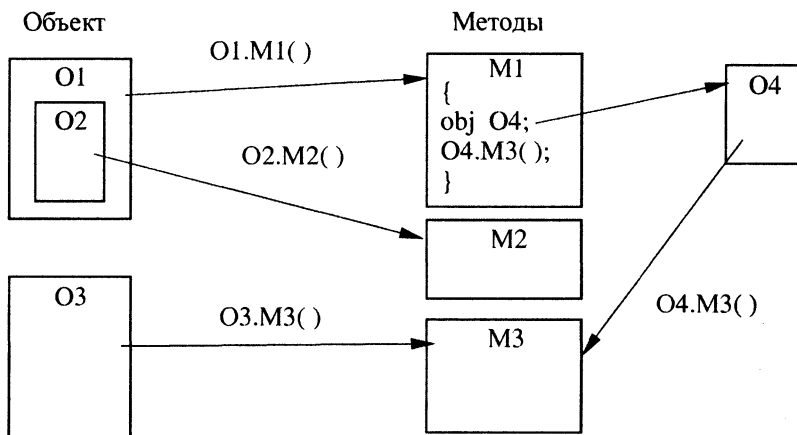


Рис. 4.2

Эпизодическое объектно-ориентированное программирование. Эпизодическое использование технологии ООП заключается в разработке отдельных, не связанных между собой классов и в использовании их как необходимых программисту базовых типов данных, отсутствующих в языке. При этом общая структура программы остается традиционной «от функции к функции». Например, для работы с матрицами программист может определить класс матриц, переопределить для него стандартные арифметические операции и использовать переменные типа «матрица» в обычной программе.

Тотальное программирование «от класса к классу». Строгое следование технологии ООП предполагает, что любая функция в программе представляет собой метод для объекта некоторого класса. Это не означает, что нужно вводить в программу какие попало классы ради того, чтобы написать необходимые для работы функции. Наоборот, класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых физических предметов или абстрактных понятий (объектов программирования). С другой стороны, каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих. В конце концов вся программа в таком виде представляет собой объект некоторого класса с единственным методом **run** (выполнить). Именно этот переход (а не понятия класса и объекта как таковые) создает психологический барьер перед программистом, осваивающим технологию ООП.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Конструктор копирования (КК). При создании объекта всегда вызывается конструктор, за исключением случая, когда объект создается копированием содержимого другого объекта этого же класса, например:

```
date    date2 = date1;
```

Имеется еще два случая, когда объект без вызова конструктора создается неявно, и оба они связаны с вызовом функции:

- формальный параметр – объект, передаваемый по значению, создается в стеке в момент вызова функции и инициализируется копией фактического параметра;

- результат функции – объект, передаваемый по значению, в момент выполнения оператора **return** копируется во временный объект, сохраняющий результат функции.

Во всех трех случаях производится обычная операция копирования структурированных переменных (точнее, элементов данных объектов). Если же объект содержит динамические данные или связанные с ним ресурсы, то такое простое копирование не совсем корректно: оба объекта содержат указатели на динамические данные, для обоих вызываются деструкторы, при разрушении одного из них другой окажется с некорректными данными (будет ссылаться на уже освобожденные динамические переменные).

Конструктор копирования должен выполнять корректное копирование содержимого объекта-параметра в текущий объект (уме-

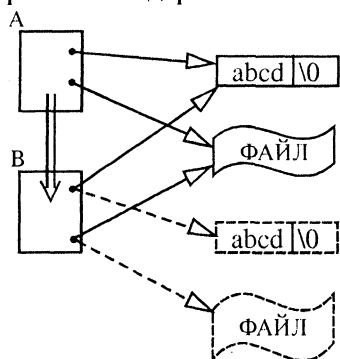


Рис. 4.3

сто назвать его «конструктором клонирования»). Он используется, если объект содержит динамические структуры данных или связанные ресурсы: конструктор должен создать их копию, «независимую от оригинала» (рис. 4.3). При передаче объектов в качестве формальных параметров и результата по значению (в виде копии) транслятор автоматически формирует вызов этого конструктора, и тогда функция получает и возвращает корректную и независимую копию объекта.

Конструктор копирования имеет жесткий синтаксис, по которому его идентифицирует транслятор: он должен иметь параметр – ссылку на объект того же класса.

```
//-----41-02.cpp
//-----Конструктор копирования для класса строк
class string {
    char * str;                // Строка динамический массив
public:
    string(char*);
    string(string&);            // Конструктор копирования
};                               // создает объект " из объекта"
string::string(string&R) {      // Создает копию строки -
    str = new char[ strlen(R.str)+1]; // динамического массива
    strcpy(str, R.str);
    printf("I am %ix\n",this);
}
string::string(char *s){
    str = new char[ strlen(s)+1];
    strcpy(str, s);
}
string copy(string x){          // Функция получает объект string
    return x;                   // и возвращает его по значению
}
void main(){                   // Копирование объектов b=a
    string a("bbbbbb"),b=a;    // string x=a; - для copy(a)
    copy(a);                   // string tmp=x; - для return x;
}
```

Конструктор копирования обязателен, если в программе используются обычные функции, функции-элементы и переопределенные операции, которые получают формальные параметры и возвращают в качестве результата такой объект по значению.

Конструктор копирования для случая разделения данных.

Альтернативный вариант копированию данных объекта – их разделение. Конструктор копирования, в принципе, может дублировать указатель на динамические данные или идентификатор ресурса (например, номер открытого файла или окна), но при этом связанные такими данными объекты должны взаимодействовать. Как минимум, они должны корректно разрушать эти общие данные, для чего обладают общим счетчиком ссылок объектов на эти данные. Счетчик ссылок также может быть динамической переменной, обычный конструктор, создающий новую строку, размещает ее в динамическом массиве и создает динамическую переменную – счетчик (указатель **pcnt**), устанавливая ее в 1. Тогда

конструктор копирования просто дублирует указатели и увеличивает значение общего счетчика на 1. Деструктор, наоборот,

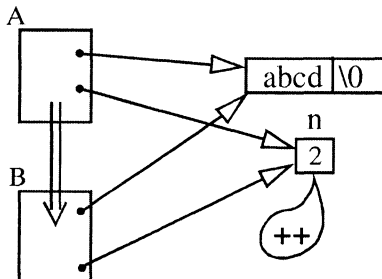


Рис. 4.4

уменьшает значение счетчика на 1 и разрушает строку только тогда, когда этот объект ссылается на строку последним (рис. 4.4).

```
//-----41-03.cpp
//---- Класс "разделяемых" строк
class cstring{
    char *str;                // Указатель на строку
    int *pcnt;                // Указатель на счетчик ссылок
public:
    cstring(char*);           // Конструктор
    cstring(cstring&);         // Конструктор копирования
    ~cstring();               // Деструктор
};
cstring::cstring(char *s){    // Конструктор создает динамический
    str=new char[strlen(s)+1]; // массив для строки и счетчик ссылок
    strcpy(str,s);
    pcnt=new int;              // Количество ссылок 1
    *pcnt=1;
}
cstring::cstring(cstring &R){ // Конструктор копирования -
    str=R.str;                 // копирует указатели
    pcnt=R.pcnt;               // и увеличивает счетчик
    (*pcnt)++;
    printf("+copies=%d %s %lx\n",*pcnt,str,this);
}
cstring::~cstring(){          // Деструктор уменьшает счетчик,
    printf("-copies=%d %s %lx\n",*pcnt,str,this);
    if ((*pcnt)-- ==1 ){      // если последний уничтожает
        delete str;           // динамические данные
        delete pcnt;
    }
}
void main(){
    cstring a("aaaaa"),b=a,c=b,*p[10];
    for (int i=0;i<10; i++) p[i]=new cstring(a);
    for (i=0;i<10; i++) delete p[i];
}
```

Принцип разделения отражается и в других переопределенных операциях: например, присваивание таких объектов должно выполняться аналогично конструктору копирования. Кроме того, хотя такие объекты передаются по значению (копия), копии тем не менее ссылаются на один и тот же оригинал.

Класс матриц произвольной размерности. Конструктор, деструктор и работа с данными объекта. Для объекта-матрицы необходимо прежде всего обеспечить ее неограниченную размерность. Сразу же заметим, что двумерные массивы имеют хотя бы одну статическую размерность (число столбцов матрицы). Поэтому резонно выбрать массив указателей на массивы – строки матрицы (неограниченная вторая размерность). Для неограниченной

первой размерности массив указателей тоже должен быть динамическим. «Самодостаточный» объект включает в себя целые переменные – текущие размерности и массив указателей. Память для структуры данных резервируется в момент конструирования объекта, тогда же задаются и размерности матрицы, в дальнейшем они не должны меняться. Для обеспечения широких возможностей задания матриц необходим набор конструкторов: конструктор, заполняющий матрицу заданным значением, значениями из линейного массива коэффициентов, конструкторы, заполняющие матрицу списком коэффициентов.

```
//-----41-04.cpp
// Матрица с динамическим массивом указателей (ДМУ) на строки
class matrix{
    int n,m; // Размерности матрицы y,x
    double **pd; // Указатель на ДМУ на строки
public: // Конструкторы
    matrix(int,int,double*);
    matrix(int,int,double,...);
    matrix(int,int,int,...);
    matrix(matrix&);
    ~matrix();
    double& Val(int,int);
};
//---- Конструктор, заполняющий матрицу из линейного массива
matrix::matrix(int y,int x,double *q){
    n=y; m=x;
    pd=new double*[n]; // Создать сам ДМУ
    for (int i=0; i<n; i++){ // Создать и заполнить строки матрицы
        pd[i]=new double[m]; // и заполнить их значениями из массива
        for (int j=0; j<m; j++) pd[i][j]=*q++;
    }
}
//---- Конструктор, заполняющий матрицу из списка коэффициентов
matrix::matrix(int y,int x,double a,...){
    double *q=&a; // Указатель на список параметров функции
    n=y; m=x;
    pd=new double*[n]; // Создать сам ДМУ
    for (int i=0; i<n; i++){ // Создать и заполнить строки матрицы
        pd[i]=new double[m]; // и заполнить их из списка параметров
        for (int j=0; j<m; j++) pd[i][j]=*q++;
    }
}
//---- Конструктор, заполняющий матрицу из списка коэффициентов
// Формат : int,int,double координаты и значение коэффициента
matrix::matrix(int y,int x,int a,...){
    int *q=&a; // Указатель на список параметров функции
    n=y; m=x;
    pd=new double*[n]; // Создать сам ДМУ
    for (int i=0; i<n; i++){ // Создать и заполнить строки матрицы
        pd[i]=new double[m]; // и заполнить их 0
        for (int j=0; j<m; j++) pd[i][j]=0;
    }
    while(*q>=0){ // Ограничитель списка значение <0
        int yy=*q++; // Извлечь координаты и коэффициент
    }
}
```

```

int xx=*q++;
double vv=((double*)q); q+=sizeof(double)/sizeof(int);
if (xx>=0 && xx<m && yy>=0 && yy<n) pd[yy][xx]=vv;
}}

```

Поскольку объект содержит динамические данные, ему необходим конструктор копирования.

```

//-----41-05.cpp
//---- Конструктор копирования
matrix::matrix(matrix &R){
n=R.n; m=R.m;          // Копировать размерности матрицы
pd=new double*[n];      // Создать сам ДМУ
for (int i=0; i<n; i++){ // Создать и заполнить строки матрицы
pd[i]=new double[m];    // и заполнить их из объекта-источника
for (int j=0; j<m; j++) pd[i][j]= R.pd[i][j];
}}

```

Деструктор объекта разрушает динамические данные объекта.

```

//-----41-06.cpp
//---- Деструктор
matrix::~matrix(){
for (int i=0; i<n; i++) delete pd[i];
delete pd; }

```

Для обеспечения доступа к содержимому матрицы достаточно реализовать метод, возвращающий ссылку на выбранный коэффициент матрицы, что позволяет работать с ним как по чтению, так и по записи. Хотя это «приоткрывает» доступ к внутренним данным объекта, но реальное использование этой ссылки «во вред объекту», доступ через нее к другим коэффициентам требует большого искусства и не может быть достигнут несознательно (по ошибке). Поэтому такую операцию можно считать безопасной.

```

//-----41-07.cpp
//---- Возвращает ссылку на заданный коэффициент
double &matrix::Val(int yy, int xx){
static double ERROR=0;
if (xx>=0 && xx<m && yy>=0 && yy<n) return pd[yy][xx];
else return ERROR; }
//----- Пример работы с матрицами
void main(){
double dd[6]={1,7,4,5,8,3};
matrix b(2,3,dd);          // Заполнение из массива
matrix c(2,2,3,3,4,4,2,5,3,6); // Заполнение из списка
matrix d(6,6,0,0,2,5,3,5,6,5,-1); // Заполнение " по координатам"
double x=d.Val(3,5); d.Val(2,3)=7; c.Val(1,1)++; d.Val(1,1)=
c.Val(1,1)+5;
for (int i=0; i<6; i++, puts(""))
for (int j=0; j<6; j++) printf("%2.1lf ",d.Val(i,j));
}

```

Статические элементы класса. Общий список объектов класса. Иногда требуется определить данные, которые относятся ко всем объектам класса. Типичные случаи: требуется контроль общего количества объектов класса или одновременный доступ ко всем объектам либо к части их, разделение объектами общих ресурсов. Тогда в определение класса могут вводиться статические элементы – переменные. Такой элемент создается в одном экземпляре, имеет свойства обычной глобальной переменной. Статический элемент в объекты класса не входит, он должен быть явно определен в программе и инициализирован по полному имени **имя_класса::имя_элемента**.

Статическими могут объявляться также и функции-элементы. Их «статичность» определяется тем, что вызов их не связан с конкретным объектом и выполняется по полному имени. Соответственно в них не используется неявный указатель на текущий объект **this**. Статистические функции вводятся, как правило, для выполнения действий, относящихся ко всем объектам класса.

В следующем примере объекты класса строк связаны в односвязный список, что позволяет в любой момент просмотреть их все при помощи статической функции. Заголовок списка – статическая переменная. В момент создания объекта конструктор помещает его в начало общего списка. Деструктор должен найти этот объект в общем списке и исключить его оттуда. Деструктор и статическая функция **show**, имея доступ ко всем объектам, может использовать все их элементы данных и вызывать для них любые методы.

```
//-----41-08.cpp
//-----Все объекты класса string связаны в односвязный список
class string{
    char *str;
    static string *fst;    // Указатель заголовков списка ( статический )
    string *next;         // Указатель на следующий элемент (обычный)
public:
    static void show();    // Просмотр всех объектов статическая функция
    void put(){ puts(str); } // Вывод содержимого объекта обычная функция
    string ( char* );      // Конструктор
    ~ string ();           // Деструктор
};
string * string::fst=NULL; // Определение статического элемента
//-----
void string::show(){
    string *p;
    for (p=fst; p !=NULL; p=p->next) p->put();
}
//----- Конструктор - включение в список объектов
string::string( char *s ){
```

```

str=new char[strlen(s)+1]; strcpy(str,s); // Динамическая копия строки
next = fst; fst = this; } // Включение в начало статического списка
//----- Деструктор - поиск и исключение из списка
string::~string(){
string *p,*pred; // Указатели на текущий и предыдущий
for ( p=fst,pred=NULL; p !=NULL; pred=p, p = p->next)
if (p == this) break; // Найден - выйти
if (p!=NULL){ // Найденный исключить из списка
if (pred==NULL) fst=fst->next;
else pred->next=p->next;
}
delete str; } // Разрушение самого объекта
//----- Вызов статической функции по полному имени
void main()
{ string a("aaa"),b("bbb"), c("ccc"); string::show(); }

```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Разработать класс, объект которого реализует «пользовательский» тип данных. Обеспечить его произвольную размерность за счет использования в объекте динамических структур данных. Разработать необходимые конструкторы, деструктор, конструктор копирования, а также методы, обеспечивающие изменение отдельных составных частей объекта (например, коэффициентов полинома) и вывод его содержимого.

1. Дата, представленная целыми переменными: год, месяц и день.

2. Время, представленное целыми переменными: час, минута, секунда.

3. Несократимая дробь, представленная двумя длинными целыми: числитель и знаменатель.

4. Целое произвольной длины во внешней форме представления в виде строки символов-цифр.

5. Целое произвольной длины в двоично-десятичной форме представления в виде: десятичная цифра – тетрада, по две тетрады в одном байте. Последовательность цифр хранится, начиная с младшей, и ограничена тетрадой с кодом 0xF.

6. Степенной полином, представленный размерностью и динамическим массивом коэффициентов.

7. Степенной полином, представленный односвязным списком ненулевых коэффициентов. Элемент списка содержит показатель степени (индекс) и само значение коэффициента.

8. Матрица произвольной размерности, представленная размерностями и линейным динамическим массивом коэффициентов матрицы, в котором она разложена по строкам.

Вход	Выход	Операция	Действия транслятора
Значение (объект)	Указатель	&	Формирует адрес входного объекта
Значение (объект)	Ссылка	---	Формирует адрес входного объекта в качестве неявного указателя
Ссылка	Указатель	&	Фиктивная операция, превращение неявного указателя в явный
Ссылка	Значение	---	Производит косвенное обращение по неявному указателю, переходит от неявного указателя к объекту-прототипу
Указатель	Значение	*	Производит косвенное обращение по указателю, переходит от указателя к указуемому объекту
Указатель	Ссылка	*	Фиктивная операция, превращение явного указателя в неявный

Перечисленные варианты преобразований и соответствующие им операции можно применять достаточно формально, обращая внимание по необходимости на механизмы их реализации.

```
struct xxx{
    xxx *copy() { xxx *q=new xxx; *q=*this; return q; }
};
void main() { xxx a; xxx *pp=a.copy(); xxx *qq=pp->copy(); }
```

Метод **copy** создает внутри себя динамический объект такого же класса, копирует в него содержимое текущего объекта (присваивание типа «объект – объект») и возвращает указатель на создаваемый объект, который запоминается в **pp**. Для этого объекта в свою очередь вызывается тот же самый метод, создающий еще одну динамическую копию объекта с запоминанием указателя в **qq**.

```
struct xxx{
    xxx &copy() { return *this; }
};
void main() { xxx a,b,c; b=a.copy(); c=a.copy().copy(); }
```

Метод **copy** возвращает ссылку на объект того же класса. В операторе **return** выражение ***this** понимается как «текущий объект». Операция «*» используется потому, что ссылка должна иметь синтаксис объекта, а не указателя. Реально же данная операция перехода от указателя к ссылке фиктивна. Содержательно метод следует понимать как формирование отображения (синонима) на объект, для которого вызывается метод. При этом повторный вызов того же самого метода применительно к результату-ссылке сопровождается преобразованиями «ссылка – объект – указатель на текущий объект», все они фиктивны. Поэтому на самом деле по

цепочке методов будет передаваться указатель на объект вплоть до последнего присваивания результата объекту **c** – присваивание «ссылка – объект» приведет к копированию из-под неявного указателя.

Переопределение операторов (операций) – использование стандартного синтаксиса языка для интерпретации транслятором известной в Си операции, если один или оба операнда в ней являются объектами данного класса:

- синтаксис языка – количество операндов, приоритеты операций и направление их выполнения – сохраняются;

- для каждого сочетания типов операндов требуется отдельное переопределение (функция-оператор), перестановка операндов транслятором не производится, например **date+int**, **int+date** – различные операции;

- способы передачи параметров (по ссылке или по значению) и способ формирования результата (ссылка на объект – операнд или новый объект-значение) определяются программистом.

Переопределение операции в классе. Для переопределения операции **date+x**, где первый операнд является объектом доступного класса, можно использовать специальную функцию-оператор:

- функция определяется в классе первого операнда;

- имя функции – **operator<знак операции>**;

- первый операнд – текущий объект функции;

- второй операнд – формальный параметр, передается как по значению, так и по ссылке. Тип формального параметра должен совпадать с типом второго операнда;

- результат операции может быть произвольного типа, он способен возвращаться как указатель, ссылка или значение (содержательная интерпретация операции);

- на действия, выполняемые в теле функции, ограничений не накладывается (содержательная интерпретация операции);

- если формальный параметр или результат передается по значению, а объект содержит динамические данные, то необходим конструктор копирования.

Рассмотрим простые с точки зрения внутреннего программирования операции инкремента даты, сложения даты и целого и вычитания одной даты из другой. Операция инкремента даты интерпретируется как добавление к ней одного дня. Если моделировать постинкремент, то необходимо возвращать новый объект – значение, причем совпадающий с начальным значением текущего объекта до выполнения операции. Поэтому в самой переопределяемой функ-

ции необходим временный объект. Поскольку объект не содержит динамических данных, все операции копирования объектов и передачи и возврата из функции по значению будут корректны (присваивание структурированных переменных).

```
//-----42-01.cpp
//-----Переопределение операции постинкремента для даты
class date{
int day,month,year;
static int days[13];
void next();           // Скрытая элемент-функция вычисления
public:                // следующего дня
date operator++();     // Операция ++
date operator+(int);   // Операция " дата+целое"
friend date operator+(int, date);
operator int();        // Дружественный оператор " целое+дата"
operator long();       // Преобразование к типу int
long operator-(date&); // Преобразование к типу long
friend ostream &operator<<(ostream& IO, date &t)
{ IO << t.day << "-" << t.month << "-" << t.year; return IO; }
date(int dd=1,int mm=1,int yy=2001) { day=dd; month=mm; year=yy; }
};
int date::days[13]={ 0,31,28,31,30,31,30,31,31,30,31,30,31};
//----- Функция вычисления следующего дня
void date::next() {
day++;
if (day > days[month]) {
if ((month==2) && (day==29) && (year%4==0)) return;
day=1; month++; // К первому числа следующего месяца
if (month==13){ // К первому января следующего года
month=1; year++;
}}}
//----- Операция постинкремента даты
date date::operator++()
{ // Создается временный объект
date x = *this; // В него копируется текущий объект
next(); // Увеличивается значение текущего объекта
return x; } // Возвращается временный объект по значению
```

Добавление к дате целого числа интерпретируется как увеличение ее значения на заданном числе дней. Результат накапливается во временном объекте, а затем возвращается оттуда по значению.

```
//-----42-02.cpp
//-----Переопределение операции " дата+целое"
date date::operator+(int n){
date x = *this; // Копирование текущего объекта в x
while (n-- !=0) x.next(); // Вызов функции next для объекта x в цикле
return(x); } // Возврат объекта x по значению
```

Переопределение операции в виде дружественной функции-оператора. Если первый операнд не может быть передан по указателю (должен быть передан по значению) или класс первого опе-

ранда базовый либо недоступный, то операция переопределяется в виде функции-оператора вне какого-либо класса:

- имя функции – **operator<знак операции>**;
- первый и второй операнд – формальные параметры, передаются как по значению, так и по ссылке. Типы формальных параметров должны совпадать с типами операндов;
- если функция-оператор имеет доступ к закрытым данным операнда, то она должна быть дружественной в классе операнда;
- остальные требования совпадают с предыдущим способом переопределения операции.

Операция «целое+дата» может быть переопределена как функция-оператор вне какого-либо класса, но дружественная в классе **date**, поскольку использует его внутренние данные. Она имеет полный список операндов, второй операнд класса **date** передается по значению и поэтому модифицируется без изменения исходного объекта.

```
//-----42-03.cpp
//----- Операция " целое + дата"
date operator+(int n, date p) {
    while (n-- !=0) p.next();           // Вызов функции next для p
    return p; }                         // Возврат объекта p по значению
```

Тип результата и способ его формирования. Тип результата переопределяемой операции не ограничен синтаксисом и выбирается произвольно, в соответствии с интерпретацией, которую задает для операции программист. Наиболее часто при переопределении операций тип результата совпадает с типом одного из операндов, то есть результат операции может использоваться как операнд в следующей аналогичной операции, образуя конвейер.

Результат операции, не имеющий отношения к операндам. Тип результата и способ его формирования может быть любым, а интерпретация – сколь угодно экзотической. Следить нужно только за соблюдением закрытости данных объекта и за корректностью работы с динамическими данными.

```
class matrix{
    int n,m;
    static double errval;    // Коэффициент «вне матрицы»
    double **pk;            // ДМУ на строки матрицы
public:
    double &operator(int y, int x){
        if (y<0 || y>=n || x<0 || x>=m)
            return errval;    // Вне матрицы – ссылка на «заглушку»
        return pk[y][x];
    };
void main(){ matrix a(5,5);
    a(1,2)=a(3,4)+5;}
```

В классе матриц, заданных динамическим массивом указателей на строки коэффициентов, переопределена операция вызова функции для двух целых операндов **matrix(int,int)**, возвращающая ссылку на выбранный коэффициент матрицы. Внутренние данные матрицы остаются практически закрытыми. Использовать ссылку на отдельный коэффициент в качестве «орудия доступа» к другим данным объекта крайне затруднительно. При этом коэффициент можно читать и записывать по ссылке.

```
class string {
    char *str;
public:
    operator char*(){
        char *q=new char[strlen(str)+1];
        strcpy(q,str);
        return q;
    };
void main(){ string a("12345"); char *s;
s = a; puts(s); delete s; }
```

В классе строк определена операция преобразования объекта к типу указателя на строку **char***. Операция возвращает копию содержимого объекта в создаваемом для этой цели динамическом массиве. Это преобразование будет неявно вызываться транслятором при любом присваивании объекта **string** указателю **char*** и приводить к извлечению внутреннего содержания объекта в виде текстовой строки. Результат операции всегда нужно сохранять для последующего освобождения памяти.

Переопределение операции по типу «конвейер ссылок». Результат операции определяется как ссылка на один из операндов. Интерпретация такой операции: результатом является сам операнд, который переносится со входа на выход операции. Такие операции не дают промежуточных объектов-результатов. Действия, производимые операцией, приводят к изменению содержания одного или обоих операндов. Операнд, ссылка на который возвращается, может участвовать в последующей аналогичным образом переопределенной операции, таким образом получается конвейер ссылок.

```
class xxx{
public:
xxx &operator+(xxx &two)
{ ...действия *this+=two... // Возвращается ссылка
  return *this; }           // на измененный первый операнд
xxx &operator-(xxx &two)
{...действия two+=*this... // Возвращается ссылка
  return two; }             // на измененный второй операнд
};
void main(){
```

```

xxx a,b,c,d;
a + b - c + d; // Эквивалентно a=a+b, c=a-c, c=c+d
a - b - c - d; // Эквивалентно b=a-b, c=b-c, d=c-d
}

```

Наиболее известный пример – стандартные потоки ввода/вывода, в которых результат – ссылка на первый операнд (поток) – позволяет применить следующую операцию по отношению к тому же самому потоку.

```

class ostream{
public:
    ostream &operator<<(int)           { .... return *this; }
    ostream &operator<<(double)        { .... return *this; }
    ostream &operator<<(char*)         { .... return *this; }
};

```

Переопределение операции по типу «конвейер значений».

Результат операции возвращается как объект-значение, то есть как копия операнда или внутреннего объекта. Интерпретация такой операции: результатом является новый объект. Резонно сохранить неизменными значения объектов-операндов, а для этого можно либо использовать внутренний локальный объект, либо передавать объекты-операнды по значению для накопления в них результата. Не следует забывать, что передача объекта-параметра по значению ведет к созданию объекта-копии фактического параметра, заполняемого конструктором копирования. Для объекта-результата транслятором создается временный объект в месте вызова, который также заполняется конструктором копирования при выполнении оператора **return**. Таким образом, реальное количество объектов в программе будет большим, чем число определенных переменных. Объект-результат может участвовать в последующей аналогичным образом переопределенной операции – получается конвейер значений.

```

class xxx{
public:
xxx operator+(xxx two) // Копия или вызов конструктора копирования
{...действия two+=*this...
return two; } // Копия или вызов конструктора копирования
xxx operator-(xxx &two)
{ xxx temp=*this; // Копия или вызов конструктора копирования
...действия temp-=two...
return temp; } // Копия или вызов конструктора копирования
};

void main(){
xxx a,b,c,d;
a + b - c + d; // Эквивалентно x1=b+a, x2=x1-c, x3=x2+d
a - b + c - d; // Эквивалентно x1=a-b, x2=c+x1, x3=x2-d
}

```

Переопределение операции присваивания. Стандартная интерпретация присваивания предполагает соблюдение следующих условий:

- разрушение содержимого текущего объекта – левого операнда (аналогично деструктору);
- копирование содержимого объекта-параметра (правого операнда) в текущий объект (аналогично конструктору копирования);
- возвращение ссылки на текущий объект.

Переопределение операции приведения типа. Особенность операции – отсутствие формальных параметров и спецификации типа результата, поскольку он и так определяется приводимым типом. Переопределенная таким образом операция будет неявно вызываться всякий раз при присваивании целому числу значения объекта либо при явном приведении объекта к этому типу (см. функцию **main** следующего примера). Содержательная интерпретация преобразования – может быть связана с получением какой-либо численной или символьной (строковой) характеристики объекта. Например, для даты это может быть количество дней от начала года или «от Рождества Христова».

```
//-----42-04.cpp
//-----Преобразование к базовым типам данных
//----- Преобразование date в int -----
date::operator int(){           // Количество дней от начала года
int r;                          // Текущий результат
int i;                          // Счетчик месяцев
for (r=0, i=1; i<month; i++)    // Число дней в прошедших месяцах
    r += days[i];
if ((month>2) && (year%4==0)) r++; // Високосный год
r += day;                      // Дней в текущем месяце
return r; }
//----- Преобразование date в long -----
date::operator long(){          // Количество дней " от Рождества Христова"
long r;                         // Текущий результат
r = 365L * (year-1);            // Дней в предыдущих полных годах
r += year / 4;                 // Високосные года
r += (int)(*this);             // Дней в текущем году - предыдущая
return r; }                    // операция (явное преобразование date в int)
//----- Операция вычисления разницы двух дат -----
long date::operator-(date &p){
return((long)(*this) - (long)p); // Преобразовать оба объекта
}                                // к типу long и вычислить разность
```

Обратите внимание еще на один эффект сочетания модульного и объектно-ориентированного программирования. Заявленные элементы-функции (методы) и переопределенные операции можно использовать «друг в друге» без ограничений. Так, для вычисления разности в днях для двух дат нужно просто привести текущий объект и объект – формальный параметр к типу **long**, а соответствующую

шая переопределенная операция для этих объектов возвратит число дней в каждом. Точно так же для определения числа дней «от Рождества Христова» можно получить число дней текущего года в дате, приведя текущий объект к типу **long**.

```
//-----42-05.cpp
//---- Пример использования
void main(){
    date a(28,2,2002);
    int b=a;           // Присваивание неявное преобразование в int
    cout << "От начала года " << b << " дней " << endl;
                        // Явное преобразование к long при выводе
    cout << "От 1.1.0001 " << (long)a << " дней " << endl;
    cout << a++ << endl;
    cout << a << " + 35 = " << a+35 << endl;
}
```

Переопределение операций new и delete. Операции создания и уничтожения объектов в динамической памяти могут быть переопределены следующим образом:

```
void *operator new(size_t size);
void operator delete (void *);
```

где **void *** – указатель на область памяти, выделяемую под объект; **size** – размер объекта в байтах; **size_t** – тип размерности области памяти, **int** или **long**. Переопределение этих операций позволяет написать собственное распределение памяти для объектов класса.

Переопределение операций () и []. Переопределение операции **()** позволяет использовать синтаксис вызова функции применительно к объекту класса (имя объекта с круглыми скобками). Количество операндов в скобках может быть любым. Переопределение операции **[]** позволяет использовать синтаксис элемента массива (имя объекта с квадратными скобками).

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Текстовые строки произвольной длины – достаточно часто используемая структура данных. Рассмотрим, как выглядит набор переопределенных операций, имеющих естественный синтаксис, позволяющий скрыть все проблемы работы с динамическими данными переменной размерности.

Поскольку при переопределении операций необходимо рассматривать всевозможные сочетания операндов, то желательно вынести наиболее часто повторяющиеся действия в набор внутренних (закрытых) элементов-функций (методов).


```
//----- 42-06.cpp
class string {
    char *str;
    void load(char *s) { str=new char[strlen(s)+1]; strcpy(str,s); }
    void add(char *);
    int find(char *);
    int cmp(string &t) { return strcmp(str,t.str); }
public: ...};
//---- Добавление строки в «хвост» строки в объекте
void string::add(char *s){
    char *ss=new char[strlen(str)+strlen(s)+1];
    strcpy(ss,str); strcat(ss,s); delete str; str=ss;
}
//---- Поиск подстроки в строке объекта
int string::find(char *s){
    int i;
    for (int k=0; str[k]!=0; k++){
        for (i=0; str[k+i]!=0 && s[i]!=0 && s[i]==str[k+i]; i++);
        if (s[i]==0) return k;
    }
    return -1;}

```

Конструкторы. Все конструкторы вызывают внутренний метод **load** для загрузки строки в создаваемый в объекте динамический массив. Они поддерживают соглашение о корректности, обеспечивающее единообразие объектов: любой объект должен содержать строку, хотя бы пустую, то есть иметь связанный с ним динамический массив. Определен также конструктор копирования.

```
class string {
    char *str; ...
public:...
    string() { load(""); }; // Пустой - строка нулевой длины
    string(char *s) { load(s); } // Конструктор из текстовой строки
    string(string &t) { load(t.str); } // Конструктор копирования
    ~string() { delete str; }
    ...};

```

Переопределение операции присваивания. Стандартная интерпретация присваивания состоит в освобождении памяти, занятой строкой в текущем объекте, в создании в нем копии строки из объекта – операнда правой части и в возвращении ссылки на текущий объект.

```
class string {
    char *str; // Строка – динамический массив
public: // Переопределение операции присваивания
    string &operator=(string&);
};
string &string::operator=(string& right) {
    delete str; // «Разрушить» текущий объект
    str = new char[strlen(right.str)+1];
    strcpy(src,right.str);
    return *this;} // Возвращает ссылку на левый операнд

```

В принципе, присваивание можно переопределить и более «явно»: с использованием прямого вызова деструктора и метода **load**. Полезно добавить операцию присваивания с левой частью – указателем на строку (присваивание стандартной текстовой строки).

```
class string {
    char *str; ...
public:...
    string &operator=(string &r)
        { this->~string(); load(r.str); return *this; }
    string &operator=(char *s)
        { this->~string(); load(s); return *this; }
    ...};
```

Переопределение операции сложения. Все возможные варианты сложения с операндами – объектами и текстовыми строками и символами – вызывают метод **add** для добавления строки к внутреннему объекту – копии первого операнда, который всегда возвращается по значению. Благодаря конструктору копирования все операции по копированию объектов корректны. Некоторые операции используют уже переопределенную операцию **string+char*** для другого сочетания операндов. Это приводит к появлению лишних промежуточных объектов в процессе трансляции методов, но зато делает программу более «читаемой» и компактной (в исходной записи).

```
class string {
    char *str; ...
public:...
string operator+(char *s)
    { string x(str); // Объект – копия текущего
      x.add(s);      // Добавить в него строку
      return x; }    // Вернуть временный объект
string operator+(char s){
    char c[2];
    c[0]=s; c[1]=0; // Создать строку из символа и использовать
    return *this + c; } // уже переопределенную операцию
string operator+(string &t){
    return *this + t.str; } // Использовать переопределенную операцию
friend string operator+(char *s, string &t){
    string x(s); // Создать объект с текстовой строкой -
    x.add(t.str); // первым операндом и добавить к нему
    return x; }   // строку из второго
    ...};
```

Обратите внимание на синтаксис ***this** для обозначения текущего объекта при вызове переопределенной ранее операции сложения **string+char***.

Переопределение операции []. Операция **string[char*]** используется для вызова внутреннего метода поиска подстроки и

возвращает индекс начала подстроки, заданной операндом в квадратных скобках (текстовой строкой). Операция **string[int]** возвращает символ из объекта-строки с индексом, переданным операндом в квадратных скобках.

```
class string {
    char *str; ...
public:...
int operator[](char *s) { return find(s); }
char operator[](int n) { if (n>=strlen(str)) return 0; else return str[n]; }
    ... };

```

Переопределение операции приведения к int. Приведение к базовому типу данных **int** интерпретируется как получение длины строки в объекте.

```
class string {
    char *str; ...
public:...
    operator int() { return strlen(str); }
};

```

Переопределение операции (). Переопределение этой операции позволяет применить к объекту синтаксис вызова функции с заданным числом параметров. В данном случае **string(int,int)** выделяет из объекта подстроку по заданным начальному и конечному индексам. Выделенная подстрока возвращается в объекте-значении, поэтому внутри самого метода создается временный объект **x**, который инициализируется выделенной частью строки. Для этого в строку текущего объекта ставится фиктивный «конец строки», который потом удаляется.

```
class string {
    char *str; ...
public:...
    string operator()(int,int);
    ...};
string string::operator()(int n1, int n2=-1){
    if (n2==-1) n2=strlen(str); // n2==-1 – до конца строки
    if (n1<0 || n2<0 || n1>strlen(str) || n2>strlen(str) || n1>n2)
        { string x("???"); return x; } // Ошибка
    char c=str[n2]; str[n2]=0; // Поставить временный «конец строки»
    string x(str+n1); // Создать объект из подстроки
    str[n2]=c; // Удалить временный «конец строки»
    return x; // Вернуть временный объект
}

```

Переопределение операций сравнения. Произведено в соответствии со стандартной интерпретацией, когда вызывается внутренний метод **cmp** с возвратом логического значения.

```

class string {
    char *str; ...
public:...
    int operator==(string &t) { return cmp(t)==0; }
    int operator!=(string &t) { return cmp(t)!=0; }
    int operator< (string &t) { return cmp(t)< 0; }
    int operator<=(string &t) { return cmp(t)<=0; }
    int operator> (string &t) { return cmp(t)> 0; }
    int operator>=(string &t) { return cmp(t)>=0; }
};

```

Переопределение операций вычитания. Операция вычитания интерпретируется как «вырезание» строки, заданной вторым операндом, из строки первого операнда, и реализуется с использованием выделения подстроки и сложения, вызываемых через переопределенные операции.

```

class string {
    char *str; ...
public:...
    string operator-(char *s);
    string operator-(string &t)    // Переопределение string-char*
        { return *this-t.str; }    // То же самое для string-string
    ...;
string string::operator-(char *s) {
    int n=find(s);                // n-индекс начала подстроки
    if (n<0) return *this;
    return (*this)(0,n)+(*this)(n+strlen(s)); }

```

Последняя строчка заслуживает отдельного комментария. Выражение **(*this)(0,n)** – выполнение переопределенной операции «**()**» по отношению к текущему объекту, которая возвращает в новом объекте-значении часть строки до начала найденного фрагмента. Выражение **(*this)(n+strlen(s))** – аналогичное действие для части строки от конца выделенного фрагмента до конца строки (второй параметр использован по умолчанию –1, что значит «до конца строки»). Части строк объединяются через переопределенную операцию сложения для полученных объектов-строк.

Переопределение ввода/вывода строк из стандартного потока. Операции ввода/вывода в стандартные потоки переопределяются как операторы-функции вне класса, поскольку первым операндом являются объекты **ostream** или **istream**, недоступные для программирования. Эти операторы должны быть дружественными в классе **string**, поскольку используют данные этого класса. Операция вывода в поток вызывает уже известную переопределенную операцию **ostream<<char*** для строки, содержащейся в объекте – втором операнде, и возвращает ссылку на первый операнд (конвейер ссылок). Переопределение ввода сложнее. Здесь необ-

ходимо не забыть разрушить загружаемый объект (уничтожить старые данные). После чего известная уже операция **istream>>char*** загружает текстовую строку в локальный массив, откуда она загружается в объект методом **load**.

```
// Переопределение ввода-вывода в стандартный поток
class string{
char *str;
void load(char*);
public:
friend ostream &operator<<(ostream& IO, string &t)
{ IO << t.str; return IO; }
friend istream &operator>>(istream &, string &);
};
istream &operator>>(istream& IO, string &t){
char c[80];           // Локальный массив
delete t.str;         // Удалить старые данные
IO.getline (c,80);    // Ввести строку из потока в массив
t.load(c);            // Загрузить объект из текстовой строки
return IO; }          // Вернуть ссылку на объект-поток ввода
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

С использованием переопределения операций разработать стандартную арифметику объектов, включающую арифметические действия над объектами и целыми (вещественными, строками – в зависимости от вида объектов), присваивание, ввод и вывод в стандартные потоки, приведение к базовому типу данных, извлечение и обновление отдельных элементов (например, коэффициентов матрицы или символов строки). По возможности организовать операции в виде конвейера значений, с результатом (новым объектом) и сохранением значений входных операндов. Для выбора варианта заданий использовать перечень классов из раздела 4.1.

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

Определите содержимое объектов после выполнения методов и переопределенных операций. Подсчитайте количество объектов в программе и изобразите схему их взаимодействия (копирование, отображение).

```
//-----42-07.cpp
//-----1.1
class integer1 { int val;
public: integer1 (int v0) { val = v0; }
      friend integer1 INC(integer1);
      integer1 INC(integer1 src) { src.val++; return src; }
};
void main1()
```

```

{ integer1 x(5) ,y =(0), z =(0); y = INC(x); z = INC(INC(x)); }
//-----1.2
class integer2 { int val;
public: integer2 (int v0) { val = v0; }
integer2 INC() { integer2 t = *this; t.val++; return t; }
};
void main2()
{ integer2 x(5) ,y =(0), z =(0); y = x.INC(); z = x.INC().INC(); }
//-----1.3
class integer3 { int val;
public: integer3 (int v0) { val = v0; }
integer3 &INC() { val++; return *this; }
};
void main3 ()
{ integer3 x(5), y(0), z=(0); y = x.INC(); z = x.INC().INC(); }
//-----1.4
class integer4 { int val;
public: integer4 (int v0) { val = v0; }
integer4 *INC() { val++; return this; }
};
void main4 () { integer4 x(5),y (0),z(0); y = *x.INC(); z = *(x.INC()->INC()); }
//-----1.5
class integer5 { int val;
public: integer5 (int v0) { val = v0; }
integer5 &operator+(int s) { val+=s; return *this; }
};
void main() { integer5 a(5),b(0); b=a+5+6; }
//-----1.6
class integer6 { int val;
public: integer6 (int v0) { val = v0; }
integer6 operator+(int s) { integer6 z(0); z=*this; z.val+=s; return z; }
};
void mai6() { integer6 a(5),b(0); b=a+5+6; }
//-----1.7
class integer7 { int val;
public: integer7 (int v0) { val = v0; }
integer7 operator+(integer7 s) { s.val += val; return s; }
};
void main7() { integer7 a(5),b(6),c(0); c=a+b; }
//-----1.8
class integer8 { int val;
public: integer8 (int v0) { val = v0; }
integer8 &operator+(integer8 &s) { val += s.val; return *this; }
};
void main8() { integer8 a(5),b(6),c(0); c=a+b+b; }
//-----1.9
class integer9 { int val;
public: integer9 (int v0) { val = v0; }
integer9 &operator+(integer9 &s) { s.val += val; return *this; }
};
void main9() { integer9 a(5),b(6),c(0); c=a+b+b; }
//-----1.10
class integer10 { int val;
public: integer10 (int v0) { val = v0; }
integer10 &operator+(integer10 &s) { s.val += val; return s; }
};

```

```

void main10() { integer10 a(5),b(6),c(0); c=a+b+b; }
//-----1.11
class integer11 { int val;
public: integer11 (int v0) { val = v0; }
integer11 operator+(integer11 s) { s.val += val; return s; }
};
void main11() { integer11 a(5),b(6),c(0); c=a+b+b; }
//-----1.12
class integer12 { int val;
public: integer12 (int v0) { val = v0; }
integer12 operator+(int two)
{ integer12 res = *this; res.val += two; return res; }
integer12 operator+(integer12 two) { return *this+two.val; }
};
void main12() {integer12 x(5),y(0), z(0); y = x + 5; z = x + y; }
//-----1.13
class integer13 { int val;
public: integer13 (int v0) { val = v0; }
integer13 operator++()
{ integer13 res = *this; val++ ; return res; }
};
void main13() {integer13 x(5),y(0), z(0); y = ++x; z = ++x; }
//-----1.14
class integer14 { int val;
public: integer14 (int v0) { val = v0; }
integer14 operator++() { val ++ ; return *this; }
};
void main14() {integer14 x(5),y(0), z(0); y = ++x; z = ++x; }
//-----1.15
class integer15 { int val;
public: integer15 (int v0) { val = v0; }
integer15 operator-(int two)
{ integer15 res = *this; res.val -= two; return res; }
integer15 operator-(integer15 two) { return *this-two.val; }
};
void main15() {integer15 x(5),y(0), z(0); y = x - 5; z = x - y; }
//-----1.16
class integer16 { int val;
public: integer16 (int v0) { val = v0; }
integer16 operator-(int two)
{ integer16 res(two); res.val -= val; return res; }
integer16 operator-(integer16 two)
{ two.val -= val; return two; }
};
void main16() {integer16 x(5),y(0), z(0); y = x - 5; z = x - y; }
//-----1.17
class integer17 { int val;
public: integer17 (int v0) { val = v0; }
integer17 &operator-(int two)
{ val -= two; return *this; }
integer17 &operator-(integer17 two)
{ val -= two .val; return *this; }
};
void main17() {integer17 x(5),y(0), z(0); y = x - 5; z = x - y; }
//-----1.18
class integer18 { int val;

```

```

public: integer18 (int v0) { val = v0; }
integer18 &operator-(int two)
{ val -= two; return *this; }
integer18 &operator-(integer18 &two)
{ val -= two.val; return two; }
};
void main18() {integer18 x(5),y(0), z(0); y = x - 5; z = x - y; }
//----- 2.1
class string1 {
char *s;
public:string1(char *);
        string1(string1 &);
        string1() ;
char operator[](int n) { if (n >= strlen(s)) return(' '); return(s[n]); }
};
void main21() { char c,t; string1 x("abcd"); c = x[2]; t = x[20]; }
//----- 2.2
class string2 {
char *s;
public:string2(char *);
        string2(string2 &);
        string2() ;
char &operator()(int n) { if (n >= strlen(s)) return(s[0]); else return(s[n]); }
};
void main22() {
string2 x("abcd"); x(2) = 'e'; x(20)='e'; }
//----- 2.3
class string3 {
char *s;
public:string3(char *);
        string3(string3 &);
        string3() ;
string3 operator=(int src){
delete s; s=new char[10];
itoa(src,s,10);          // Перевод во внешнюю форму представления
return *this;}
operator int () { return(strlen(s)); }
};
void main23() {
string3 x("abcd"),y,z; int i0 = 10,i1,i2,i3;
i1 = x; z = i0; i2 = z; i3 = y = (int)x + 10; }
//----- 2.4
class string4 {
char *s;
public:string4(char *);
        string4(string4 &);
        string4() ;
string4 &operator+(string4 &t){
char *p = new char[strlen(s)+strlen(t.s)+1];
strcpy(p, s); strcat(p, t.s); delete s;
s = p; return *this; }
};
void main24() {
string4 x("abcd"),y("11"); x + y + y; x + x; }
//----- 2.5
class string5 {

```



```

char *s;
public:string5(char *);
    string5(string5 &);
    string5();
string5 &operator+(char *t) {
char *p = new char[strlen(s)+strlen(t)+1];
strcpy(p, s); strcat(p, t); delete s;
s = p; return *this ; }
};
void main25()
{ string5 x("abcd"),y("11"),z; x + "444"; y + "22" + "44"; }
//----- 2.6
class string6 {
char *s;
public:string6(char *);
    string6(string6 &);
    string6();
string6 operator+(char *t){
string6 x;
delete x.s;
x.s = new char[strlen(s)+strlen(t)+1];
strcpy(x.s, s); strcat(x.s, t); return x;}
};
void main26()
{ string6 x("abcd"),y,z; y = x + "11"; z = x + "22"; }
//----- 2.7
class string7 {
char *s;
public:string7(char *);
    string7(string7 &);
    string7();
string7 operator+(string7 t){
string7 x; x.s = new char[strlen(s)+strlen(t.s)+1];
strcpy(x.s, s); strcat(x.s, t.s);
return x;}
};
void main27() { string7 x("abcd"),y("111"),z; z = x + y + x; }

```

4.3. КЛАССЫ СТРУКТУР ДАННЫХ. ШАБЛОНЫ

Обеспечение независимости типов хранимых данных от структуры данных. Ранее уже мы обсуждали задачу (раздел 3.3), когда структура данных «не знает», какого типа переменные она хранит. Вне технологии ООП это достигается использованием указателя типа **void*** и итераторов, выполняющих конкретные действия над хранимыми элементами посредством внешних функций, вызываемых через указатели (рис. 4.5). Такое решение основано на механизме **динамического связывания**, поскольку части алгоритма соединяются между собой в процессе выполнения программы, оно может быть перенесено практически без изменений в технологию ООП. Но здесь дополнительно к нему имеется два противоположных по сути решения этой проблемы:

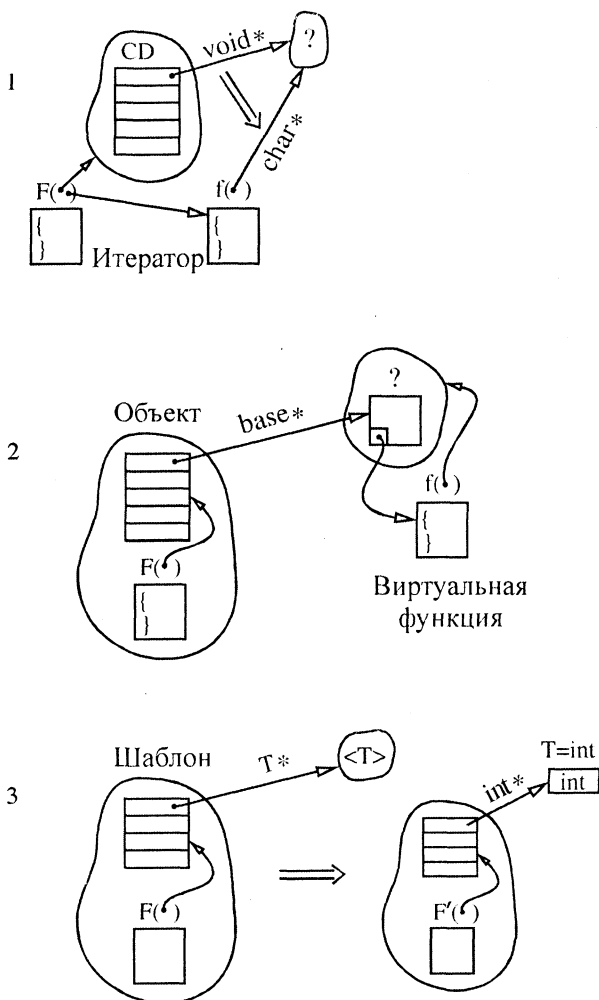


Рис. 4.5

– первый вариант реализует тот же самый механизм динамического связывания в рамках виртуальных функций;

– второй вариант, наоборот, позволяет реализовать необходимое разнообразие статически, то есть во время трансляции программы, а точнее, даже перед началом трансляции. Для этого необходимо иметь заготовку описания структуры данных, в которой хранимый тип обозначен именем-параметром, своего рода «заглушкой». Это и называется **шаблоном**.

Шаблоны. В Си++ имеются средства, позволяющие определить некоторое множество идентичных классов с параметризованным типом внутренних элементов. Они представляют собой заготовку класса (**шаблон**), в которой в виде параметра задан тип (класс) входящих в него внутренних элементов. При создании конкретного объекта необходимо дополнительно указать и конкретный тип внутренних элементов в качестве фактического параметра. Создание объекта сопровождается формальной генерацией соответствующего класса для типа, заданного в виде параметра. Принятый в Си++ способ определения множества классов с параметризованным внутренним типом данных (иначе – макроопределение) называется шаблоном (**template**).

Шаблон – макроопределение (текстовая заготовка) класса с параметром – типом данных.

Следующим примером попробуем «убить двух зайцев». Во-первых, пояснить довольно витиеватый синтаксис шаблона, а во-вторых, выделить особенности реализации структур данных с использованием технологии ООП. Основной принцип шаблона – добавление к имени класса «довеска» в виде имени – параметра (например, **vector<T>**). Это имя обозначает внутренний тип данных, который может использоваться в любом месте класса: как указатель, ссылка, формальный параметр, результат, локальная или статическая переменная. В остальном шаблон не отличается от обычного класса. Само имя шаблона (**vector**) теперь обозначает не один класс, а группу классов, отличающихся только внутренним типом данных.

```
//-----43-01.cpp
//---- Шаблон - Динамический массив указателей
// <class T> - параметр шаблона - класс T, внутренний тип данных
// vector - имя группы шаблонных классов
template <class T> class vector
{
int tsize;           // Общее количество элементов
T **obj;             // Массив указателей на параметризованные
public:              // объекты типа T
T *operator[](int);  // Оператор [int] возвращает указатель на
                    // параметризованный объект класса T
operator int();       // возвращает количество указателей
int insert(T*);       // Включение указателя на объект типа T
int index(T*);
vector(int);
~vector(); };

```

Данный шаблон может использоваться для порождения объектов-векторов, каждый из которых хранит указатели на объекты

определенного типа. Имя класса при этом составляется из имени шаблона **vector** и имени типа данных (класса), который подставляется вместо параметра **T**.

```
vector<int>      a;
vector<double>   b;
extern class     time;
vector<time>     c;
```

При определении каждого вектора с новым типом объектов транслятором генерируется описание нового класса по заданному шаблону (естественно, неявно в процессе трансляции). Например, для типа **int** транслятор получит:

```
class vector<int>{
    int  tsize;
    int  **obj;
public:
    int  *operator[](int);
    operator int();
    void insert(int*);
    int  index(int*); };
```

Далее следует утверждение: элементы – функции шаблона – шаблонные функции (типа «масло масляное»). Это означает, что функции – элементы, составляющие шаблон, также должны быть «заготовками» с тем же самым параметром, то есть генерироваться для каждого нового типа данных. То же самое касается переопределяемых операторов.

```
//-----43-02.cpp
//---- Функции шаблонного класса тоже шаблоны
// Параметр шаблона - класс T, внутренний тип данных
// имя функции-элемента или оператора - параметризовано
template <class T> vector<T>::operator int(){
    for (int n=0; obj[n]!=NULL; n++);
    return n; }
template <class T> T* vector<T>::operator[](int n){
    if (n >=(int) *this) return NULL;    // (int)*this вызов операции
    return obj[n]; }                    // приведения к int

template <class T> int vector<T>::index(T *pobj){
    int sz= *this;                      // Неявный вызов приведения к int
    for ( int n=0; n<sz; n++)
        if (pobj == obj[n]) return n;
    return -1;}
```

При определении каждого вектора с новым типом объектов транслятором генерируется набор элементов – функций по заданным шаблонам (естественно, неявно в процессе трансляции). При этом сами шаблонные функции должны размещаться в том же заголовочном файле, где размещается определение шаблона самого класса. Для типа **int** сгенерированные транслятором функции-элементы будут выглядеть следующим образом:

```

int* vector<int>::operator[](int n){
if (n >=(int)*this) return NULL;    // (int)*this – вызов операции
if (n >=sz) return NULL;           // приведения к int
return obj[n]; }

int vector<int>::index(int *pobj){
int      sz=*this;                  // Неявный вызов приведения к int
for (int n=0; n<sz; n++){
    if (pobj == obj[n]) return n;
return -1;}

```

Не забудьте, что приведенный выше пример – иллюстрация того, что делает транслятор при определении объекта шаблонного класса.

Шаблоны могут иметь и параметры-константы, которые используются для статического определения размерностей внутренних структур данных. Кроме того, шаблон используется для размещения не только указателей на параметризованные объекты, но и самих объектов. В качестве примера рассмотрим шаблон для построения циклической очереди ограниченного размера для параметризованных объектов.

```

//-----43-03.cpp
//----- Шаблон с параметром-константой
template <class T,int size> class FIFO
{
int fst,lst;                        // Индексы начала и конца очереди
T queue[size];                     // Массив объектов класса T размерности size
public:
T from();                          // Функции включения-исключения
void into(T);                      // Конструктор
FIFO();
};
template <class T,int size> FIFO<T,size>::FIFO()
{ fst = lst = 0; }

template <class T,int size> T FIFO<T,size>::from(){
T work=-1;
    if (fst !=lst){
        work = queue[lst++];
        lst = lst % size;
    }
return work;}

template <class T,int size> void FIFO<T,size>::into(T obj) {
queue[fst++] = obj;
fst = fst % size; }

```

Объекты такого шаблонного класса при определении имеют два параметра: тип данных и константу – статическую размерность.

```

struct x {...};
FIFO<double,100>    a;
FIFO<int,20>        b;
FIFO<x,50>          c;

```

Особенности хранимых объектов – параметров шаблона.

Отметим некоторые нюансы взаимоотношений шаблона структуры данных и объектов хранимых классов – параметров шаблона:

- если шаблон хранит указатели на объекты, то он не касается проблем корректного копирования объектов и «не отвечает» за их создание и уничтожение. Деструктор шаблона обязан уничтожить динамические компоненты структуры данных (динамические массивы указателей, элементы списка), но он обычно не уничтожает хранимые объекты;

- если шаблон хранит сами объекты, то он «должен быть уверен» в корректном копировании объектов при их записи и чтении из структуры данных (конструктор копирования для объектов, содержащих динамические данные). При разрушении структуры данных разрушаются и копии хранимых объектов.

Особенности представления классов списков и деревьев.

Для представления списка и дерева необходимы две сущности: элементы списка (вершины дерева), связанные между собой, и заголовков – указатель на первый элемент списка (корневую вершину). В технологии ООП есть два равноправных решения:

- разрабатывается два класса – класс элементов списка и класс заголовка списка. Объекты первого класса пользователем (программой, работающей с классом) не создаются. Они все – динамические, и их порождают методы второго класса – заголовка;

- разрабатывается один класс, объекты которого играют разную роль в процессе работы класса. Первый объект – заголовок, создается программой (статически или динамически), доступен извне и не содержит данных (по крайней мере в момент конструирования). Остальные объекты, содержащие данные, создаются динамически методами, работающими с первым объектом. Этот вариант более прост в реализации, но имеет некоторые тонкости, связанные с «различением» в процессе работы объектов того и другого типа. Все приведенные ниже примеры соответствуют этому варианту.

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Массив указателей произвольной размерности. Объект – массив указателей – должен содержать динамический массив указателей на произвольные элементы данных (типа **void***), его размерность задается при конструировании объекта и должна автоматически увеличиваться при переполнении. Массив указателей со-

держит последовательность указателей, ограниченную **NULL**. Для реализации методов, связанных с упорядоченностью данных, используется технология итераторов (см. раздел 3.3), указатель на внешнюю функцию сравнения для элементов хранимого типа передается при конструировании в объект и там хранится.

```
//-----43-04.cpp
class MU{
    int sz;           // Текущая размерность ДМУ
    void **p;         // Динамический МУ на элементы данных
    int(*cmp)(void*,void*); // Указатель на внешнюю функцию сравнения
    int extend();      // Увеличение размерности ДМУ
public:
    MU(int, int(*)(void*,void*)); // Конструктор - размерность
    ~MU();               // и функция сравнения
    int size();          // Количество указателей в ДМУ
    void *operator[](int); // Извлечение по логическому номеру
    int operator()( void*,int); // Включение по логическому номеру
    void *remove(int);    // Удаление по логическому номеру
    void *min();          // Итератор поиска минимального
};
```

Конструктор создает динамический массив указателей размерности, заданной параметром, и «очищает» его. Деструктор, естественно, разрушает динамический массив указателей. С элементами данных при этом ничего не происходит, поскольку структура данных осуществляет только их хранение и не отвечает за процессы их создания и уничтожения. Метод **extend()** создает новый массив указателей, размерностью в два раза больше, и переписывает в него указатели из «старого».

```
//-----43-05.cpp
MU::MU(int n=20 , int(*pf)(void*,void*)=NULL) // По умолчанию 20, NULL
{ cmp=pf; sz=n; p=new void*[n]; p[0]=NULL; }

MU::~MU() { delete p; }

int MU::extend() {
    void **q=new void*[sz*2];
    if (q==NULL) return 0;
    for (int i=0; i<sz; i++) q[i]=p[i];
    delete p;
    p=q; sz*=2;
    return 1; }
int MU::size() { int i; for (i=0; p[i]!=NULL; i++); return i; }

void *MU::operator[](int n=-1){ // Извлечение по логическому номеру
    int k=size();               // По умолчанию - последний
    if (n== -1) n=k-1;
    if (n<0 || n>=k) return NULL; // Недопустимый номер
    return p[n]; }

int MU::operator()(void *q, int n=-1) { // Включение по номеру
    int k=size();
```

```

if (n== -1) n=k; // По умолчанию - последним
if (n<0 || n>k) return 0; // Недопустимый номер
if (k==sz-1) extend(); // При переполнении -
for (int i=k; i>=n; i--) p[i+1]=p[i]; // увеличить размерность
p[n]=q; // Вставка " с раздвижкой"
return 1; }

void *MU::remove(int n=-1) { // Удалить по логическому номеру
int k=size();
if (n== -1) n=k-1; // По умолчанию - удалить последний
if (n<0 || n>=k) return NULL;
void *q=p[n];
for (;p[n]!=NULL; n++) // "Уплотнить" массив указателей
p[n]=p[n+1];
return q; } // Возвратить указатель на удаленный элемент

void *MU::min() { // Итератор поиска минимального
void *pmin; int i; // элемента
if ( cmp==NULL) return NULL; // Нет функции сравнения
for ( i=0, pmin=p[0]; p[i]!=NULL; i++)
if ( (*cmp)(pmin,p[i]) > 0) pmin=p[i];
return pmin; }

```

Циклический список. Циклический список организован в соответствии с принятым принципом совмещения заголовка списка и его элементов в объектах одного класса. Первый элемент списка – текущий объект, доступный через **this**, является заголовком и не содержит данных. Остальные элементы – динамические, создаются при помещении в список новых данных и удаляются при их исключении (рис. 4.6).

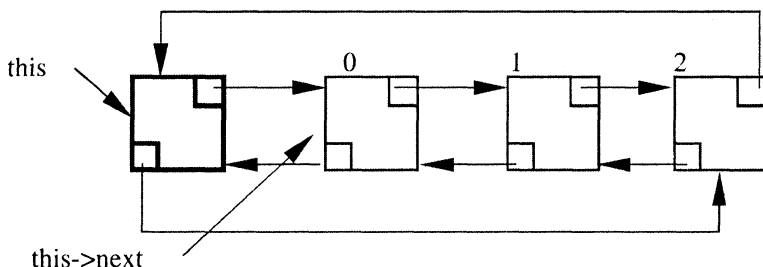


Рис. 4.6

```

//-----43-07.cpp
class zlist {
void *data; // Указатель на элемент данных
zlist *next,*prev;
int (*cmp)(void*,void*); // Внешняя функция сравнения элементов
zlist *find(int); // Вспомогательный метод извлечения
public: // элемента списка по номеру
zlist(int*)(void*,void*)=NULL; // Конструктор пустого списка
~zlist();
int size(); // Количество элементов

```



```

void *operator[](int);           // Извлечение
void operator()(void*,int);     // Включение по номеру
void *remove(int);             // Удаление по номеру
void *remove(void*);           // Удаление по указателю на данные
void *min();                   // Итератор поиска минимального
};

```

Конструктор списка определяет текущий объект как единственный элемент, который в соответствии с правилами построения циклического списка «замкнут сам на себя», а также запоминает указатель на внешнюю функцию сравнения элементов данных.

```

//-----43-08.cpp
zlist::zlist( int(*pf)(void*,void*))
{ prev=next=this; cmp=pf; }

```

Вспомогательный метод извлечения элемента списка по его последовательному номеру демонстрирует все особенности объектно-ориентированной реализации. Первый элемент списка – заголовок – является текущим объектом (**this**), при этом в процессе счета он не учитывается. Цикл просмотра начинается с первого информационного элемента (**this->next** или **next**) и завершается по возвращении на заголовок. В последнем случае логический номер не найден.

```

//-----43-09.cpp
zlist *zlist::find(int n=-1)
{ zlist *p;
  for (p=next; n!=0 && p!=this; n--, p=p->next);
  return p; }

```

Метод подсчета количества элементов в структуре данных стандартным образом обходит циклический список.

```

//-----43-10.cpp
int zlist::size()
{ int n; zlist *p;
  for (n=0, p=next; p!=this; n++, p=p->next);
  return n; }

```

Метод извлечения указателя на элемент данных по логическому номеру (переопределенная операция []) использует для получения указателя на элемент списка внутренний метод **find** и выделяет из элемента списка указатель на данные.

```

//-----43-11.cpp
void *zlist::operator[](int n=-1){
  if (n== -1) n=size()-1;
  zlist *p=find(n);
  return p==this ? NULL : p->data; }

```

Метод исключения элемента данных из списка прежде всего находит элемент списка, извлекает из него указатель на элемент данных. Сам элемент списка при этом удаляется, поскольку он – ди-

намический объект. Указатель на элемент данных возвращается, так как структура данных не несет ответственности за размещение самого элемента данных в памяти и не распределяет память под него.

```
//-----43-12.cpp
void *zlist::remove(int n=-1){
if (n== -1) n=size()-1;           // По умолчанию - удалить последний
zlist *p=find(n);                 // Найти элемент списка по номеру
if (p==this) return NULL;         // Номер не существует - удалить нельзя
void *s=p->data;                   // Сохранить указатель на данные
p->prev->next=p->next;              // "Обойти" элемент двусвязного списка
p->next->prev=p->prev;
p->next=p->prev=p;                 // Перед удалением - сделать его
delete p;                          // "единственным"
return s;                           // Возвратить указатель на данные
```

Метод исключения по указателю на элемент данных используется, когда необходимо удалить уже известный элемент данных. В этом методе ищется заданный указатель и удаляется содержащий его элемент списка.

```
//-----43-13.cpp
void *zlist::remove( void *pd){
zlist *p= next;
for (; p!=this; p=p->next){
if (p->data==pd)
{
p->prev->next=p->next; // "Обойти" элемент списка
p->next->prev=p->prev;
p->next=p->prev=p;    // Перед удалением - сделать его
delete p;             // "единственным"
return pd;            // Возвратить указатель на данные
}}
return NULL; }
```

Два предыдущих метода «зацикливают» удаляемый элемент. Это необходимо для корректной работы деструктора (об этом речь пойдет ниже).

Метод включения элемента данных по логическому номеру, наоборот, создает динамический объект – элемент списка, после чего включает его в список. Таким образом, элементами списка являются динамические объекты, создаваемые методами, работающими с его заголовком.

```
//-----43-14.cpp
void zlist::operator()(void *s,int n=-1)
{
zlist *p=find(n);           // По умолчанию - включить перед заголовком,
zlist *q=new zlist;         // в циклическом списке - последним
q->data=s;                   // Создать новый элемент списка
q->next=p;                   // Включить перед найденным - p
q->prev=p->prev;
p->prev->next=q;
p->prev=q; }
```

Метод поиска минимального элемента использует внешнюю функцию сравнения, указатель на которую хранится в объекте.

```
//-----43-15.cpp
void *zlist::min(){
    if (next==this) return NULL;           // Пустой список
    zlist *pmin=next;
    for (zlist *q=next; q!=this; q=q->next)
        if ( cmp(pmin->data,q->data) > 0) pmin=q;
    return pmin->data;}
```

Отдельного обсуждения заслуживает деструктор. Дело в том, что деструктор может вызываться в двух совершенно различных ситуациях:

- когда удаляется элемент списка (при выполнении операции **remove**). В этой ситуации он всегда – единственный удаляемый;
- когда в программе удаляется сам список. В этом случае деструктор вызывается для объекта-заголовка. Но если список не будет пустым, то деструктор должен предпринять меры к удалению включенных в него элементов списка, которые по своей природе динамические.

Вся проблема заключается в том, что деструктор сам не в состоянии определить, в какой из приведенных ситуаций он находится. Ниже приводится один из способов решения проблемы. Метод **remove** перед удалением динамического объекта-элемента списка делает его «единственным». Деструктор же, наоборот, удаляет при помощи метода **remove** элементы списка, следующие за текущим объектом, пока он не станет единственным. Заметим, что при этом деструктор освобождает только элементы списка, но ничего не делает с указателями на элементы данных (это отдельная проблема).

```
//-----43-16.cpp
zlist::~zlist() { while (remove(0)!= NULL); }
```

В заключение рассмотрим пример использования объектов класса в программе. Хранимые в списке данные-строки являются статическими объектами, поэтому проблемы распределения памяти под них здесь не актуальны. Конструктор передает объекту стандартную функцию сравнения строк, предварительно «поменяв ее прототип» с использованием преобразования типа указателя на функцию.

```
//-----43-17.cpp
//---- Пример работы с объектом – циклическим списком
// Преобразование указателя на функцию int(*)(char*,char*)
// К указателю на функцию int(*)(void*,void*)
void main(){
    zlist A((int*)(void*,void*))stricmp);
```

```

A((void*)"aaaa");
A((void*)"bbbb");
A((void*)"cccc");
for (int i=A.size()-1; i>=0; i--) puts((char*)A[i]);
puts((char*)A.min());
puts((char*)A.remove(1));}

```

Двоичное дерево с указателями на объекты произвольного типа. Для двоичного дерева имеется аналогичная проблема в представлении объекта-заголовка дерева и объектов-вершин. Самый простой вариант состоит в том, что корневая вершина дерева является одновременно и объектом-заголовком (рис. 4.7). Но, поскольку дерево может быть и пустым, следует допустить наличие в вершине дерева **NULL**-указателя на элемент данных. Такая вершина будет считаться «незанятой». Конструктор объекта-вершины дерева должен создавать именно такую «незанятую» вершину.

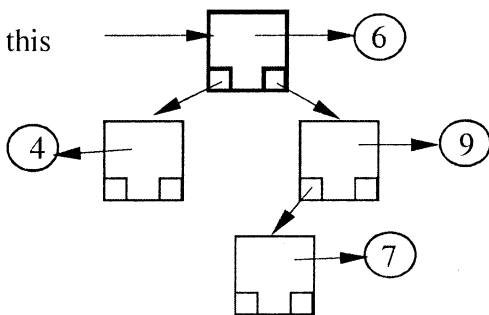


Рис. 4.7

Другая особенность методов для классов деревьев – все они являются рекурсивными. Это значит, что тот же самый метод необходимо вызвать для объекта-потомка через соответствующий указатель в текущем объекте – текущей вершине дерева. И, наконец, двоичное дерево обеспечивает естественную упорядоченность хранимых данных по мере его обхода «левое поддерево – вершина – правое поддерево». Поэтому операция включения производится только с сохранением упорядоченности и является итератором.

//-----43-18.cpp

```

class btree {
void *data;           // Указатель на хранимый элемент данных
int(*cmp)(void*,void*); // Внешняя функция сравнения элементов
btree *l,*r;          // Левое и правое поддерево
public:
btree(int(*) (void*,void*)); // Конструктор - создать " пустую" вершину
~btree();
int size();            // Количество элементов

```

```
void *operator[](int&); // Извлечение по номеру
void *operator[](void*); // Извлечение по ключу (двоичный поиск)
void operator()( void*); // Включение с сохранением упорядоченности
};
```

```
btree::btree(int(*pf)(void*,void*))
{ cmp=pf, r=l=NULL; data=NULL;} // Создает "пустую" вершину
```

Деструктор должен быть рекурсивным, поскольку при разрушении данных в вершине необходимо выполнить операцию уничтожения и освобождения памяти из-под объектов – потомков, которые в свою очередь в своих деструкторах сделают то же самое для своих потомков.

```
//-----43-19.cpp
```

```
btree::~btree(){
if (l!=NULL) delete l;
if (r!=NULL) delete r;}
```

Следующие методы подсчета количества вершин и двоичного поиска по ключу (элемент – образец **key**) используют естественный синтаксис для рекурсивного вызова обычного метода, если известен указатель на объект-потомок. Заметим, что при каждом новом рекурсивном вызове вызываемый объект также становится текущим. В отличие от списка, где **this** обозначал объект-заголовок, а для объектов – элементов списка в том же методе использовался другой указатель, в дереве для всех объектов-вершин используется один (но каждый раз свой) **this**. Причем в отличие от обычных функций, работающих с деревьями, где допускается рекурсивный вызов с **NULL**-указателем, в классе перед обращением к поддереву проверяется его наличие, поскольку **NULL**-указатель на текущий объект недопустим.

```
//-----43-20.cpp
```

```
int btree::size() {
if (data==NULL) return 0;
int n=1;
if (l!=NULL) n+=l->size();
if (r!=NULL) n+=r->size();
return n; }

void *btree::operator[](void *key){
if (data==NULL) return NULL;
int n>(*cmp)(key,data);
if (n==0) return data;
if (n < 0) {
    if (l!=NULL) return (*l)[key];
    else return NULL;
}
else {
    if (r!=NULL) return (*r)[key];
    else return NULL;
}}
```

В методе извлечения по логическому номеру используется обход двоичного дерева, который дает последовательность элементов данных в порядке возрастания. Для подсчета вершин применяется общий для всех вершин формальный параметр – счетчик, который передается по ссылке.

```
//-----43-21.cpp
void * btree::operator[](int &n){
    void *q;
    if (data==NULL) return NULL;
    if (!n) { q=(*l)[n]; if (q!=NULL) return q; }
    if (n-- == 0) return data;
    if (r!=NULL) { q=(*r)[n]; if (q!=NULL) return q; }
    return NULL;}

```

В приведенном фрагменте интересно выглядит рекурсивный вызов переопределенного оператора []. Для этого нужно в синтаксисе вызова указать объект – потомок для текущего – это будет (*l), и для него выполнить указанную операцию, то есть (*l)[n].

Переопределенный оператор () для включения нового элемента данных также имеет свою специфику. Перед переходом в правое или левое поддерево он проверяет соответствующий указатель. Если тот равен NULL, то сначала «подвешивается» новый динамический объект (создаваемый конструктором как «пустой»), а затем в него производится рекурсивный вход, после которого он будет заполнен указателем на данные.

```
//-----43-22.cpp
void btree::operator()(void* pnew){
    if (data==NULL) { data=pnew; return; }
    int n=(*cmp)(pnew,data);
    if (n <= 0) {
        if (!n) l=new btree(cmp);
        (*l)(pnew);
    }
    else {
        if (r==NULL) r=new btree(cmp);
        (*r)(pnew);
    }
}

```

Шаблон класса односвязного списка, содержащего объекты. Одна из особенностей шаблонов для списков и деревьев состоит в том, что элементы списка (вершина дерева) хранят указатели на объекты порождаемого класса и имеют поэтому составное параметризованное имя, например, **LIST<T>**. Такой же тип должен использоваться в операторе **new**.

```
//-----43-23.cpp
template <class T> class LIST
{ LIST<T> *next;           // Указатель на следующий в списке
  T data;                  // Элемент списка хранит сам объект
};

```

```

public:
LIST() { next=NULL; }
~LIST();
void Insert(T,int);          // Включение по логическому номеру
void Insert(T);              // Включение с сохранением порядка
T Min();                     // Поиск минимального элемента
friend ostream &operator<<(ostream&, LIST<T>&);
};                             // Переопределенный вывод в поток

```

Методы класса, являющиеся шаблонными функциями, полностью воспроизводят алгоритмы работы с односвязным списком, учитывая особенность его реализации в классе: первый элемент списка (и текущий объект **this**) – заголовочный, он не содержит данных. Поэтому во всех операциях метода удобно использовать рабочий указатель на предыдущий элемент списка.

```

//-----43-24.cpp
template <class T> void LIST<T>::Insert(T newdata,int n){
LIST<T> *p,*q;
p = new LIST<T>;
p->data = newdata;
for (q = this; q->next !=NULL && n !=0; n--, q = q->next);
p-> next = q-> next;
q-> next = p; }

```

В методах, связанных со сравнением хранимых объектов, играет свою роль тот факт, что шаблон «не знает» свойств этих объектов и предполагает, что любой тип данных, который будет параметром шаблона, имеет стандартным образом определенные (или переопределенные) операции сравнения. Поэтому можно смело ставить знаки операций > или <, не забывая что операндами должны быть объекты класса **T**, а не их указатели.

```

//-----43-25.cpp
template <class T> void LIST<T>::Insert(T newdata){
LIST<T> *p,*q;
p = new LIST<T>;
p->data = newdata;
for (q = this; q->next !=NULL && newdata > q->next->data; q = q->next);
p-> next = q-> next;
q-> next = p;}

template <class T> T LIST<T>::Min(){
LIST<T> *q;
T MinObj;
if (next==NULL) return MinObj;
for (q = next , MinObj = q->data; q !=NULL; q = q->next)
if (q->data < MinObj) MinObj = q->data;
return MinObj; }

```

Отдельное замечание по деструктору. По аналогии с деревом он является рекурсивным. Это связано с тем, что при уничтожении списка деструктор вызывается только для объекта – заголовка, но

он должен также разрушить динамические объекты – элементы списка.

```
//-----43-26.cpp
template <class T> LIST<T>::~~LIST(){
    if (next!=NULL) delete next;
}
```

В переопределении операции вывода в стандартный поток происходит просмотр списка (с этой целью оператор является дружественным в классе списка). Для каждого хранимого объекта вызывается операция << по отношению к параметру – потоку. То есть считается, что хранимый объект тоже имеет определенную (или переопределенную) операцию вывода.

```
//-----43-27.cpp
template <class T> ostream &operator<<(ostream &O, LIST<T> &R){
    LIST<T> *q;
    for (q = R.next; q !=NULL; q = q->next) O << q->data;
    return O; }
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Разработать шаблон класса структуры данных, включая «джентльменский набор» операций: конструктор «пустой» структуры данных, деструктор, операции добавления, включения и исключения по логическому номеру, сортировки и включения с сохранением порядка при наличии стандартным образом переопределенной операции сравнения объектов класса – параметра шаблона. Разработать также методы сохранения и загрузки структуры данных из стандартного текстового (или двоичного) потока при наличии переопределенных операций << и >> для потока и объектов хранимого класса – параметра шаблона. Загрузку структуры данных из стандартного потока производить в создаваемые для этой цели динамические объекты.

1. Стек, представленный динамическим массивом указателей на хранимые объекты. Размерность стека увеличивается в момент его переполнения.

2. Стек, представленный динамическим массивом хранимых объектов. Размерность стека – параметр шаблона.

3. Циклическая очередь, представленная динамическим массивом хранимых объектов. Размерность очереди – параметр шаблона.

4. Циклическая очередь, представленная динамическим массивом указателей на хранимые объекты. Размерность очереди – параметр конструктора.

5. Динамический массив объектов. Размерность – параметр конструктора.

6. Динамический массив указателей на объекты. Размерность массива указателей увеличивается в момент его переполнения. Начальная размерность – параметр конструктора.

7. Односвязный список, содержащий указатели на объекты.

8. Односвязный список, содержащий объекты.

9. Двусвязный список, содержащий указатели на объекты.

10. Двусвязный список, содержащий объекты.

11. Двусвязный циклический список, содержащий указатели на объекты.

12. Двусвязный циклический список, содержащий объекты.

13. Двоичное дерево, содержащее объекты.

14. Двоичное дерево, содержащее указатели на объекты.

15. Дерево, вершина которого содержит статический массив объектов (размерность – параметр шаблона) и указатели на правое и левое поддеревья (работа только с логическими номерами).

4.4. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Элементы данных – объекты класса. Простейшим случаем введения иерархии в систему классов является использование объектов ранее определенных классов в качестве элементов данных нового класса. Взаимодействие классов в этом случае ограничивается тем, что новый класс использует стандартный интерфейс объекта: функции-элементы и переопределенные операции, то есть работает с ним как с любым другим базовым типом данных. Элемент данных – объект класса представляет собой некоторое частное свойство более сложного класса, в котором он определен.

```
class man{
    char   name[20];      // Обычные элементы данных класса
    char   *address;
    date   date1;         // Дата рождения – объект класса date
    date   date2;         // Дата поступления на работу
public: ... };
```

Единственной проблемой является конструирование объектов – элементов данных. Здравый смысл подсказывает, что если в конструкторе нового класса не содержится информации о конструировании объектов – элементов данных, то по умолчанию для них возможен только вызов их собственных конструкторов без параметров, причем перед вызовом конструктора нового класса. Последний может воспользоваться некоторыми свойствами уже инициализированных внутренних объектов.

Если все-таки требуется использовать конструкторы внутренних объектов с параметрами, то в заголовке конструктора нового класса их необходимо явно перечислить. Их параметры могут быть любыми выражениями, включающими формальные параметры конструктора нового класса.

```
class man {
    char  name[20];    // Другие элементы класса
    dat   dat1;        // Дата рождения
    dat   dat2;        // Дата поступления на работу
public:
    man(char *,char *,char *); // Конструкторы
    man(char *);
};
//----- Конструктор класса man с неявным вызовом -----
//      конструкторов для dat1 и dat2 без параметров
man::man(char *p) { ... }
//----- Конструктор класса man с явным вызовом -----
//      конструкторов для dat1 и dat2 с параметрами
man::man(char *p,char *p1, char *p2) : dat1(p1), dat2(p2)
{ ... }
void main() { man JOHN("John","8-9-1958","15-1-1987"); }
```

Конструктор с единственным параметром вызывает по умолчанию для внутренних объектов конструкторы без параметров. Конструктор с тремя параметрами передает второй и третий из них конструкторам внутренних объектов.

Производные классы. Наследование. Основной способ создания иерархии классов заключается в том, что новый класс автоматически включает в себя все свойства старого класса, а затем развивает их. С абстрактной точки зрения старый класс определяет только общие свойства, а новый – конкретизирует более частные. Сохранение с новым классе свойств старого называется **наследованием**: элементы данных старого класса автоматически становятся элементами данных нового класса, а все функции-элементы старого класса применимы к объекту нового класса, точнее, к его старой составляющей. Старый класс при этом называется **базовым классом (БК)**, новый – **производным классом (ПК)**.

Основные свойства базового и производного классов:

- объект базового класса определяется в производном классе как неименованный. Это значит, что он не может быть использован в явном виде как обычный элемент данных;

- элементы данных базового класса включаются в объект производного класса (как правило, транслятор размещает их в начале объекта производного класса). Однако личная часть базового класса закрыта для прямого использования в производном классе;

– функции-элементы базового класса наследуются в производном классе, то есть вызов функции, определенной в базовом классе, возможен для объекта производного класса и понимается как вызов ее для входящего в него объекта базового класса;

– в производном классе можно переопределить (перегрузить) наследуемую функцию, которая будет вызываться вместо нее. При этом для выполнения соответствующих действий над объектом базового класса она может включать явный вызов переопределенной функции по полному имени.

Предполагается, что при вызове в производном классе функций, наследуемых из базового, транслятор преобразует указатель **this** объекта производного класса в указатель на входящий в него объект базового класса, учитывая размещение второго в первом.

Взаимоотношение конструкторов и деструкторов базового и производного классов аналогичны описанным выше:

– если конструктор производного класса определен обычным образом, то сначала вызывается конструктор базового класса без параметров, а затем конструктор производного класса. Деструкторы вызываются в обратном порядке: сначала для производного, затем для базового;

– в заголовке конструктора производного класса может быть явно указан вызов конструктора базового класса с параметрами. Он может быть без имени или с именем базового класса. Если базовых классов несколько, то вызовы конструкторов базовых классов должны быть перечислены через запятую и поименованы.

Синтаксис и права доступа. Синтаксис наследования устанавливает правила переноса личной и общей частей базового класса в производный. Для этого личная часть класса разбивается на собственно личную часть (без метки или с меткой **privat**) и защищенную личную часть (метка **protected**). Разница между ними состоит в том, что обычная личная часть при наследовании становится вообще недоступной, а защищенная остается доступной для общей части производного класса. Кроме того, имеется обычное наследование, которое «закрывает» внешний доступ к общей части базового класса, и публичное (со словом **public** в заголовке производного класса), которое сохраняет этот доступ (рис. 4.8). Приводимая ниже условная схема (внимание: не программа) показывает внутреннее содержимое производного класса и все упомянутые правила переноса.

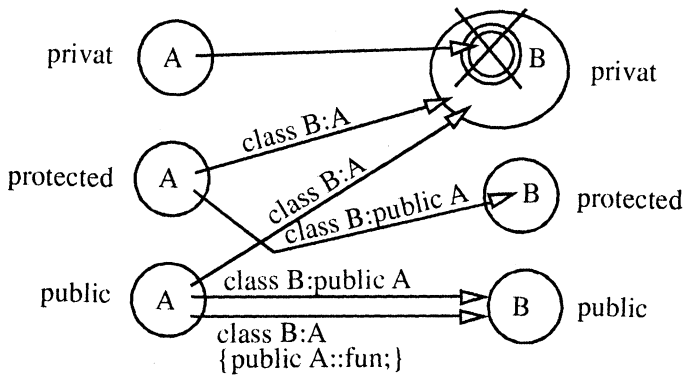


Рис. 4.8

```
// Обычное и публичное наследование
class A      {      // Базовый класс
    int n;      // Личная часть
    void f();
protected:   int m;      // Защищенная часть
    void q();
public:      int k;      // Общая часть
    void t(); };
//----- Обычное наследование
class B : A {      // Заголовок класса
    ...
public A::t;      // Явное объявление элемента общей части
};
//----- Содержимое производного класса
class B      {
    (-)int n;      // Личная часть класса A недоступна
    (-)void f();
privat:      int m;      // Защищенная часть перенесена в личную
    void q();      // часть
    int k;      // Общая часть перенесена в личную
public:      ...      // часть
    void t();      // Явно перенесенный элемент общей части
                // в общую часть
//----- Публичное наследование
class B : public A { // Заголовок класса
    };
//----- Содержимое производного класса
class B      {
    (-)int n;      // Личная часть класса A недоступна
    (-)void f();
protected:   int m;      // Защищенная часть перенесена
    void q();      // в защищенную часть
public:      int k;      // Общая часть перенесена в общую часть
    void t(); };

```

Наследование как способ изменения свойств класса. Принцип наследования следует воспринимать прежде всего в рамках

программирования «от класса к классу». При проектировании производного класса определяется потенциальное множество объектов с новыми свойствами, отличными от свойств объектов базового класса. Внешне наблюдаемые свойства объекта – это его методы. Перечисленные варианты наследования методов базового класса в производном нужно воспринимать в широком смысле – как способы изменения свойств объекта.

1. **«Новое свойство»**. Имя определяемого в производном классе метода не совпадает ни с одним из известных в базовом классе. В этом случае это – «новое свойство» объекта, которое он приобретает в производном классе.

```
class a {
public: void f() {}
};
class b : public a {
public: void newb() {}    // newb() - новое свойство (метод)
};
```

2. **«Полное неявное наследование»**. Если в производном классе метод не переопределяется, то по умолчанию он наследуется из базового класса. Это значит, что он может быть применен к объекту производного класса, при этом будет вызван метод для базового класса, причем именно для объекта базового класса, включенного в производный. Определенное в базовом классе свойство не меняется.

```
class a {
public: void f() {}
};
class b : public a{
public:                // f() - унаследованное свойство (метод)
                    // эквивалентно void f() { a::f(); }
};
```

3. **«Полное перекрытие»**. Если в производном классе определяется метод, совпадающий с именем метода базового класса, причем в теле метода отсутствует вызов одноименного метода в базовом классе, то мы имеем дело с полностью переопределенным свойством. В этом случае свойство объекта базового класса в производном классе отрицается, а метод производного класса «перекрывает» метод базового.

```
class a {
public: void f() {}
};
class b : public a
{
public:
void f() {...}    // Переопределенное свойство (метод)
};
```

4. **«Условное наследование».** Наиболее точно отражает сущность наследования последний вариант, в котором в производном классе переопределяется метод, перекрывающий одноименный метод базового класса. Но в методе базового класса обязательно имеется вызов перекрытого метода базового класса – условный или безусловный. Этот прием полнее всего соответствует принципу развития свойств объекта, поскольку свойство в производном классе является усложненным вариантом аналогичного свойства объекта базового класса.

```
class a {
public: void f() {}
};
class b : public a
{
public:
    void f()
    { ... a::f(); .... }
};
```

Переопределенное свойство развивает соответствующее свойство объекта базового класса. Переопределенный метод в явном виде вызывает метод в базовом классе по его полному имени.

Указатели на объекты базового и производного классов. Преобразуя указатель на объект производного класса в указатель на объект базового класса, мы получаем доступ к вложенному объекту базового класса. Но при такой трактовке преобразования типа указателя транслятору необходимо учитывать размещение объекта базового класса в производном, что он и делает. В результате значение указателя (адрес памяти) на объект базового класса может оказаться не равным исходному значению указателя на объект производного. Ввиду «особости» такого преобразования оно может быть выполнено в Си++ неявно (остальные преобразования типов указателей должны быть явными). Побочный эффект такого преобразования состоит в том, что транслятор «забывает» об объекте производного класса и вместо переопределенных в нем функций вызывает функции базового:

```
class A {
    public: void    f1();
};
class B : A {
public:    void    f1(); // Переопределена в классе B
          void    f2();
};
void main(){
    B    x;
    A *pa = &x;    // Прямое преобразование - неявное
```

```

pa->f1();           // Вызов A::f1(), хотя внутри объекта класса B
pb = (B*) pa;       // Обратное преобразование - явное
pb ->f2(); }         // Корректно, если под pa был объект класса B

```

После преобразования указателя на объект класса **B** в указатель на объект класса **A** происходит вызов функции из вложенного объекта базового класса **A::f1()**, хотя реально под указателем находится объект класса **B**.

Обратное преобразование от указателя на базовый класс к указателю на производный делается только явно. Преобразование корректно, если данный объект базового класса действительно входит в объект того производного класса, к типу указателя которого он приводится.

Содержательное определение полиморфизма. Наиболее содержательным синонимом к термину «полиморфный» является слово **многоликий**. Полиморфная функция – это метод (функция), определенный в группе родственных классов и выполняющий одно и то же по своей сути действие, но в особой интерпретации применительно к каждому из классов. Свойство полиморфности заключается в том, что функция в состоянии идентифицировать класс объекта и корректно выполниться в нем, даже если отсутствует полная информация о том, к какому из классов относится объект. Этим самым создается иллюзия функции «единой во многих лицах» – в каждом из родственных классов.

Полиморфность базируется на наследовании. В классах имеется естественный способ «установления родства» – общее происхождение, то есть общий базовый класс, который служит формальным основанием для объединения разнородных объектов.

Важнейшее следствие полиморфности – возможность организовать регулярный процесс обработки объектов группы производных классов.

Виртуальная функция. В Си++ свойство полиморфности реализуется виртуальной функцией. Пусть имеется базовый класс **A** и производные классы **B**, **C**. В классе **A** определена функция-элемент **f()**, в классах **B**, **C** – унаследована и, возможно, переопределена. Пусть имеется массив указателей на объекты базового класса – **p**. Он инициализирован указателями как на объекты класса **A**, так и на объекты производных классов **B**, **C** (точнее, на вложенные в них объекты базового класса **A**) (рис. 4.9):

```

class a { ... void f(); };
class b : public a { ... void f(); };
class c : public a { ... void f(); };
a A1; b B1; c C1;
a *p[3] = { &B1, &C1, &A1 };

```

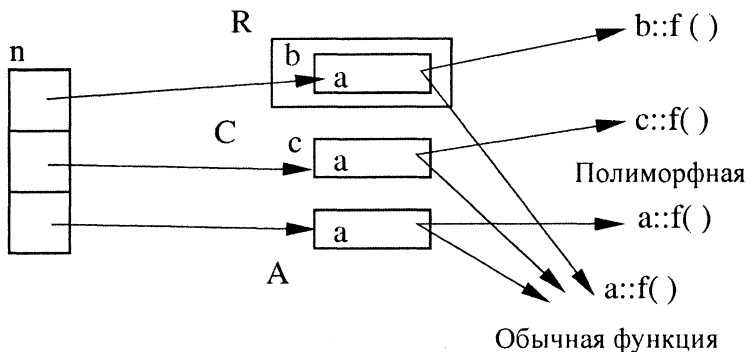


Рис. 4.9

Как будет происходить вызов обычной непалиморфной функции при использовании указателей из этого массива? Транслятор, располагая исключительно информацией о том, что указуемыми переменными являются объекты базового класса А (это следует из определения массива), вызовет во всех случаях функцию **a::f()**. То же самое произойдет, если обрабатывать массив указателей в цикле:

```
p[0]->f();    // Вызов a::f() во всех трех случаях
p[1]->f();    // по указателю на объект базового класса
p[2]->f();
for (i=0; i<=2; i++) p[i]->f();
```

Наличие указателя на объект базового класса А свидетельствует: в данной точке программы транслятор не располагает информацией о том, объект какого из производных классов расположен под указателем. Тем не менее, если функция полиморфна, то при вызове ее по указателю на объект базового класса она должна идентифицировать его производный класс и вызвать переопределенную функцию именно для этого класса:

```
p[0]->f();          // Вызов b::f() для B1
p[1]->f();          // Вызов c::f() для C1
p[2]->f();          // Вызов a::f() для A1
for (i=0; i<=2; i++) // Вызов b::f(), c::f(), a::f()
p[i]->f();          // В зависимости от типа объекта
```

Виртуальная функция – функция, определенная в базовом и переопределенная (унаследованная) в группе производных классов. При вызове виртуальной функции через указатель на объект базового класса происходит вызов функции в производном классе, соответствующем объекту производного класса, его окружающему.

Таким образом, при преобразовании типа «указатель на производный класс» в «указатель на базовый класс» происходит потеря информации о типе объекта производного класса, а при вызове виртуальной функции – обратный процесс неявного восстановления типа объекта. Объект базового класса должен быть доступен через указатель только по той причине, что это единственный в Си механизм, позволяющий ссылаться на объекты неопределенного вида.

Виртуальная функция и динамическое связывание. Нетрудно заметить, что виртуальная функция и динамическое связывание имеют много общего. В обоих случаях контексты программы (указатель на функцию, указатель на объект базового класса) «ничего не говорят» о том, какая функция в действительности будет вызвана. Механизм виртуальных функций реализуется через указатели на функции, которые связываются с объектом базового класса в момент его создания (то есть динамически, во время работы программы) (рис. 4.10):

- для каждой пары производный класс – базовый класс транслятором генерируется свой массив указателей, каждой виртуальной функции соответствует в нем свое значение индекса (смещение);

- указатель на массив (начальный адрес) записывается в объект базового класса в момент конструирования объекта производного класса;

- если объект базового класса расположен не в начале объекта производного класса, то перед вызовом виртуальной функции транслятор должен предусмотреть преобразование указателя на объект базового класса в указатель на объект производного (например, использовать дополнительную таблицу смещений);

- вызов виртуальной функции транслируется в вызов функции по указателю, извлеченному по фиксированному смещению из таблицы, связанной с объектом базового класса.

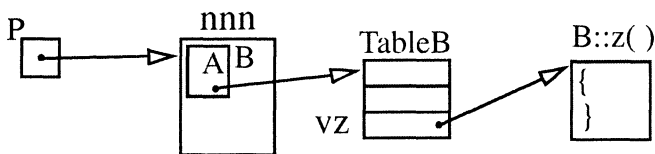


Рис. 4.10

```
// Иллюстрация механизма виртуальных функций «классическим» Си
// Выделены компоненты, создаваемые транслятором
class A {
void (**ftable)(); // Указатель на массив указателей
public:             // виртуальных функций (таблицу функций)
```

```

virtual void x();
virtual void y();
virtual void z();
    A();
    ~A(); };
#define vx      0      // Индексы в массиве
#define vy      1      // указателей на виртуальные функции
#define vz      2      //

// Массив указателей функций класса A
void (*TableA[])() = { A::x, A::y, A::z };

A::A()
{ ftable = TableA; ...} // Назначение таблицы для класса A

class B : public A {
public:
    void x();
    void z();
    B();
    ~B(); };

// Массив адресов функций класса A в B
// A::y - наследуется из A, B::x - переопределяется в B
void (*TableB[])() = { B::x, A::y, B::z };
B::B()
{ A::ftable = TableB; ...} // Назначение таблицы для класса B
void main(){
A* p;
B nnn; // Ссылается на объект производного класса B
A *p = &nnn; // Указатель p базового класса A
p->z(); } // Реализация - (*(p->ftable[vz]))();

```

Абстрактные классы. Если базовый класс используется только для порождения производных классов, то виртуальные функции в базовом классе могут быть «пустыми», поскольку никогда не будут вызваны для объекта базового класса. Определять тела этих функций не требуется. Базовый класс, в котором есть хотя бы одна такая функция, называется **абстрактным**.

```

class base {
public:
virtual print()=0;
virtual get() =0; };

```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Класс двоичного файла произвольного доступа, производный от fstream. Для работы с двоичным файлом произвольного доступа можно развить базовый класс **fstream**, включив в него методы, открывающие файл в нужном режиме, а также поддерживающие часто используемый объектами формат записи переменной длины. Обратите внимание, что все методы базового класса

применимы к производному как в самих его методах, так и к объекту **BinFile** извне.

```
//-----44-01.cpp
#include <fstream.h>
#define FNULL -1L
typedef int BOOL;
typedef long FPTR;
typedef unsigned char *BUF;
class BinFile : public fstream{
public:
    BOOL Create(char *);
    BOOL Open(char *);
    FPTR Size()
    { seekg(0L,ios::end);
      return(tellg()); }
    void *VSZLoad(int&);
    FPTR VSZAppend(void*,int);
    FPTR VSZUpdate(void*,int,int);
    BinFile() { fstream(); }
    ~BinFile() { close(); }
};
//----- Создать "пустой" файл и закрыть его
BOOL BinFile::Create(char *s){
    int a=1;
    open(s, ios::trunc | ios::out | ios::binary );
    if (good()) { close(); return 1; }
    return 0; }
//----- Открыть существующий файл в режиме двоичного чтения /записи
BOOL BinFile::Open(char * s){
    open(s,ios::in | ios::out | ios::binary);
    return good(); }
```

До сих пор функции-элементы класса представляли собой практически чистые вызовы библиотечных функций. Но поскольку значительное число объектов имеет переменную размерность, то класс **BinFile** полезно дополнить функциями, работающими с записями переменной длины. Напомним, что запись переменной длины в файле представлена целой переменной – счетчиком и следующими за ним байтами данных, число которых определяется счетчиком.

```
//-----44-02.cpp
void *BinFile::VSZLoad(int &sz){
    char *pdata;
    read((BUF)&sz, sizeof(int));
    if (!good()) return NULL;
    if ((pdata = new char[sz])==NULL ) return NULL;
    read((unsigned char *)pdata,sz);
    if (good()) return (void *)pdata;
    delete pdata;
    return NULL; }
```

При обновлении записи переменной длины проверяется размерность уже существующей старой записи. Если она недостаточ-

на, новая запись добавляется в конец файла. Значение параметра **mode=1** устанавливает режим проверки. Методы обновления и добавления возвращают указатель в файле (типа **FPTR**) на размещенную запись.

```
//-----44-03.cpp
FPTR BinFile::VSZUpdate(void *buf ,int sz, int mode) {
    int oldsz;
    FPTR pos;
    pos = tellg();
    if (mode){
        read((BUF)&oldsz,sizeof(int));
        if (!good()) return(FNULL);
        if (oldsz < sz) return VSZAppend(buf,sz);
        seekg(pos);
        if (!good()) return FNULL;
    }
    write((BUF)&sz,sizeof(int));
    write((BUF)buf,sz);
    if (!good()) return FNULL;
    return pos; }

FPTR BinFile::VSZAppend(void *buf ,int sz){
    FPTR pos;
    if ((pos = Size()) ==FNULL) return FNULL;
    write((BUF)&sz,sizeof(int));
    write((BUF)buf,sz);
    if (!good()) return FNULL;
    return pos; }
```

Виртуальные функции – как элемент «отложенного» проектирования. Свойство виртуальной функции – выполнять особые действия в каждом производном классе – можно рассмотреть еще и с другой стороны. Пусть имеется некоторый базовый класс, в котором отдельные частные действия необходимо отнести «на потом», то есть определить уже в процессе использования и адаптации этого класса к конкретным применениям. Тогда «пользователь класса» должен разработать на основе заданного класса свой производный класс, а «вынесенные» за пределы базового класса действия реализовать в виде виртуальной функции.

Рассмотрим в качестве примера фрагмент класса двоичного файла, в котором обработка ошибок открытия файла вынесена за пределы класса – в производный класс.

```
//---Класс двоичных файлов с «отложенной» функцией обработки ошибок
#include <fstream.h>
typedef int      BOOL;
typedef long FPTR;           // Тип - указатель в файле

class BinFile : public fstream {
public:
    BOOL Open(char *);           // Открыть существующий
    virtual int OnError(char *s) // Обработка ошибок открытия по умолчанию –
    {return 0; }
```

```

}; // отказ от дальнейших попыток открыть файл
BOOL BinFile::Open(char * s){
char ss[80];
strcpy(ss,s);
while (1){
    open(ss,ios::in | ios::out | ios::binary);
    if (good()) return 1;
    if (!OnError(ss)) // Виртуальная функция в производном классе
        return 0; // ищет другое подходящее имя файла
}
return 1;}

```

Виртуальная функция в производном классе должна выполнить конкретный диалог, содержание которого в базовом классе не раскрывается. В качестве результата она должна загрузить строку – имя нового файла. Если «пользователь» производного класса предполагает продолжать диалог, он может это сделать, например, так:

```

//-----
class MyFile : public BinFile {
public:
virtual int OnError(char *s) {
    cout << "не могу открыть файл" << s << endl;
    cout << "введите еще (CR-отказ):";
    cin.getline(s,80);
    return s[0]==0;
}};

```

Виртуальные функции – как элемент объединения классов и создания интерфейсов. Один из наиболее распространенных приемов использования виртуальных функций – создание базовых классов, объединяющих в группу различные классы на основе некоторого общего свойства. Базовый класс при этом включает в себе общие свойства этой группы, а весь набор действий, которые одинаково применимы к объектам из любого класса, реализуется через виртуальные функции. Таким образом, базовый класс создает **интерфейс**, позволяющий единообразно работать с разнородными объектами, например, хранить их в общей структуре данных или выполнять одну и ту же операцию над всеми ими в одном цикле.

В качестве примера рассмотрим группу классов – пользовательских типов данных. Допустим, проектируется структура данных, предназначенная для хранения произвольных объектов (пользовательских типов данных). Прежде всего, определяется ряд общих действий, которые обязательно должны быть выполнимы в объекте любого класса, чтобы он мог включаться в структуру данных, и выделяются в абстрактный (пустой) базовый класс.

```
//-----44-05.cpp
// Абстрактный базовый класс для пользовательских типов данных
class ADT {
public:
virtual int Get(char *)=0;      // Загрузка объекта из строки
virtual char *Put()=0;         // Выгрузка объекта в строку
virtual long Append(BinFile&)=0; // Добавить объект в двоичный файл
virtual int Load(BinFile&)=0;
virtual int Type()=0;          // Возвращает идентификатор
                                // типа объекта
virtual char *Name()=0;         // Возвращает имя типа объекта
virtual int Cmp(ADT *)=0;       // Сравнивает значения объектов
virtual ADT *Copy()=0;          // Создает динамический объект -
                                // копию с себя (клонирование)
virtual ~ADT(){};              // Виртуальный деструктор
};
```

Как видим, базовый класс получился абстрактным, то есть его объект не содержит данных, а функции «пустые». Это значит, что объекты базового класса не могут создаваться в программе, а сам класс – это исключительно «объединяющая идея». В принципе, базовый класс может содержать данные и непустые функции, если в самой группе классов выделяется некоторая общая часть.

Естественно, что при проектировании любого производного класса в первую очередь в нем должны быть реализованы виртуальные функции, которые поддерживают в нем перечисленные действия. Остальная часть класса может быть какой угодно, естественно, что она уже не может использоваться в общих функциях работы со структурой данных.

```
//-----44-06.cpp
#define STR_TYPE 1           // Внутренний тип объекта - строка
class string : public ADT {
char *str;
void load(char *s) { str=new char[strlen(s)+1]; strcpy(str,s); }
public:
    string(char *);
    string();
virtual int Get(char *);
virtual char *Put();
virtual FPTR Append(BinFile&);
virtual int Load(BinFile &);
virtual int Type();
virtual char *Name();
virtual int Cmp(ADT *);
virtual ADT *Copy();
};

int string::Get(char *s) { delete str; load(s); }
char *string::Put()
{ char *p =new char[strlen(str)+1];
strcpy(p,str); return p; }
```

```

FPTR string::Append(BinFile &F)
{ return F.VSZAppend(str,strlen(str)+1); }
int string::Load(BinFile &F)
{ int n; delete str;
if ((str=(char*)F.VSZLoad(n))==NULL)
    { load("File Error"); return 0; }
return 1; }
int string::Type() { return 1; }
char *string::Name() { return "String"; }
int string::Cmp(ADT *s) { return strcmp(str,((string*)s)->str); }
ADT *string::Copy() { string *p=new string(str); return (ADT*)p; }

```

Все эти методы конкретизируют действия, которые должны быть произведены в классе строк при доступе к его объектам через интерфейс **ADT**. Например, метод добавления объекта к заданному двоичному файлу вызывает в этом файле метод добавления строки, содержащейся в объекте, в виде записи переменной длины.

Базовый класс и набор виртуальных функций используются как общий интерфейс доступа к объектам – типам данных при проектировании структуры данных. Любое множество объектов, для которых осуществляются основные алгоритмы (хранение, включение, исключение, сортировка, поиск и т.д.), будет представлено как множество указателей на объекты базового класса **ADT**, а за ними способны «скрываться» объекты любых производных классов. Все действия, выполняемые над объектами, осуществляются уже в производных классах через перечисленные виртуальные функции. В качестве примера рассмотрим фрагмент класса – массив указателей.

```

//-----44-07.cpp
//----- Класс - массив указателей на объекты произвольного типа
// #include "ADT.h"
class MU {
int sz; // Текущая размерность ДМУ
ADT **p; // Динамический массив указателей ADT*
public:
Append(ADT*); // Добавление указателя на объект
ADT *min(); // Поиск минимального
void sort(); // Сортировка
int test(); // Проверка на идентичность типов объектов
int save(char *);
int load(char *);
MU(int);
~MU();
};
int MU::test() { // Вызов виртуальных функций выделен
for (int i=1; p[i]!=NULL; i++)
if ( p[i]->Type() != p[i-1]->Type()) return 0;
return 1; }

ADT *MU::min()
{ ADT *pmin; int i;
if (p[0]==NULL || !test()) return NULL;

```

```

for (i=0, pmin=p[0]; p[i]!=NULL; i++)
    if ( pmin->Cmp(p[i]) > 0) pmin=p[i];
return pmin; }
void MU::sort()
{ int d,i; ADT *q;
if (p[0]==NULL || !test()) return;
do {
    for (d=0, i=1; p[i]!=NULL; i++)
        if ( p[i-1]->Cmp(p[i]) > 0)
            {d++; q=p[i-1]; p[i-1]=p[i]; p[i]=q; }
    } while (d!=0);}

```

Отдельного обсуждения заслуживают проблемы уничтожения объектов из группы производных классов, указатели на которые хранятся в структуре данных. Если предположить, что хранимые объекты динамические и при ее разрушении возможно разрушение этих объектов, то деструктор, вызываемый для объекта базового класса **ADT**, должен быть виртуальным.

```

//-----44-08.cpp
//--- Разрушение структуры данных совместно с хранимыми объектами
MU::~MU(){
for (int i=0; p[i]!=NULL; i++)
    delete p[i]; // Разрушить объект через указатель на базовый
delete p; } // Разрушить массив указателей

```

Еще одна тема для размышления – работа с двоичным файлом, в который «вперемешку» записываются объекты, хранимые в структуре данных. Очевидно, что для идентификации объектов в файле перед каждым из них потребуется сохранять его тип (результат виртуальной функции **Type**), после чего вызывать виртуальную функцию сохранения объекта в заданном двоичном файле **Append**.

```

//-----44-09.cpp
int MU::save(char *s){
    BinFile F;
    if (!F.Create(s)) return 0; // Создать "пустой" двоичный файл
    if (!F.Open(s)) return 0; // Открыть его
    for ( int i=0; p[i]!=NULL; i++){ // Пройтись по структуре данных
        int t= p[i]->Type(); // Получить тип объекта
        F.write((BUF)&t,sizeof(int)); // Записать тип объекта в файл
        if (!F.good()) { F.close(); return 0; }
        if ( p[i]->Append(F)==FNULL) // Записать в файл сам объект
            { F.close(); return 0; }
    }
    int t=0; // Записать в файл ограничитель 0
    F.write((BUF)&t,sizeof(int));
    return 1; }

```

При загрузке структуры данных из двоичного файла необходимо создавать динамические объекты различных производных клас-

сов, в которые «подгружать» содержимое при помощи виртуальной функции **Load**. Здесь уже динамические возможности Си++ достигли своего потолка: создать произвольный объект по его идентификатору, полученному из файла, можно только явно, с использованием переключателя.

```
//-----44-10.cpp
int MU::load(char *s){
    BinFile F; ADT *q;
    if (!F.Open(s)) return 0;
    while (1) {
        int t;
        F.read((BUF)&t,sizeof(int));
        if (!F.good()) { F.close(); return 0; }
        switch (t) { // Виртуальный конструктор !!!!!!!!!!!!!!!!!!!!!!!
                    // Создать объект производного класса по его типу
        case 0:      F.close(); return 1;
        case STR_TYPE: q =(ADT*) new string; break;
        case DATE_TYPE: q =(ADT*) new date; break;
        default:      F.close(); return 0;
                    }
        q->Load(F); // Загрузить содержимое объекта
        Append(q); // Добавить к структуре данных
    }
    return 1; }
```

Взаимодействие объектов в программе. Из рассмотренных нами элементов технологии ООП – классов, объектов, наследования, программирования «от класса к классу» пока не выстраивается полная картина. Вопрос: «А как объекты разных классов связываются между собой и образуют единое целое, называемое объектной программой?» – выпал из поля зрения. Действительно, структурное программирование дает нам общий вид программы – «пирамиды» из функций, на вершине которой находится **main**.

Объектно-ориентированный подход заключается в первичности данных (объектов) по отношению к алгоритмам (методам) их обработки. Причем любая сущность, с которой программист сталкивается при написании программы, должна являться объектом (меню, файл, таблица, строка, вплоть до самой **main**). Цепочка «функция–функция» в такой программе заменяется на цепочку «объект–метод–объект», которая уже не всегда строго древовидна. Сравним два варианта.

Объект **aa** класса **A** вызывает метод **F**, в котором создается локальный объект **bb** класса **B**, для которого вызывается метод **G**. Здесь мы имеем некоторый эквивалент цепочки вызовов функций: объект во время работы порождает объект, во время работы с ним порождается объект и т.д. (рис. 4.11).

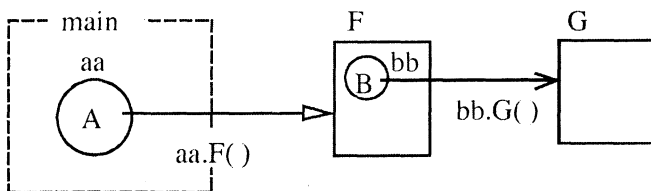


Рис. 4.11

```

class B{
public:    void G(){ ... }
};
class A{
public:    void F(){ B bb; bb.G(); ... }
};
void main(){ A aa; aa.F(); }

```

В следующем примере структура данных, хранящая указатели на объекты различных типов с общим базовым классом, начинает сохранение содержимого этих объектов с определения локального объекта класса **BinFile** (двоичный файл). Затем метод просматривает структуру данных, выбирая из нее указатели на объекты, и вызывает для них виртуальную функцию **Append**, которая в качестве параметра получает ссылку на двоичный файл. В результате выполняется метод **Append** в одном из производных классов, который в свою очередь использует объект **BinFile** для выполнения собственных действий, определяемых форматом представления этого объекта в двоичном файле (рис. 4.12).

Логическим завершением этого подхода является взгляд на программу как на систему взаимодействующих объектов. Но тогда теряется само представление о программе как о едином потоке управления. Объекты в процессе вызова и выполнения своих методов получают доступ к другим объектам, для которых они вызывают соответствующие методы, и т.д. В такой схеме очень важен вопрос – сколько объектов имеется в программе, как они получают информацию друг о друге, кто и когда их создает и уничтожает. В зависимости от принятых решений и выбранной стратегии взаимодействия объектов находятся такие свойства программы, как гибкость и универсальность. Применительно к связям между объектами можно употребить термины «статический» и «динамический»:

- объекту известно имя другого объекта – в этом случае связь устанавливается программистом при написании программы и никогда не меняется;

– объект получает указатель на другой объект. В этом случае связи между объектами устанавливаются динамически во время работы программы.

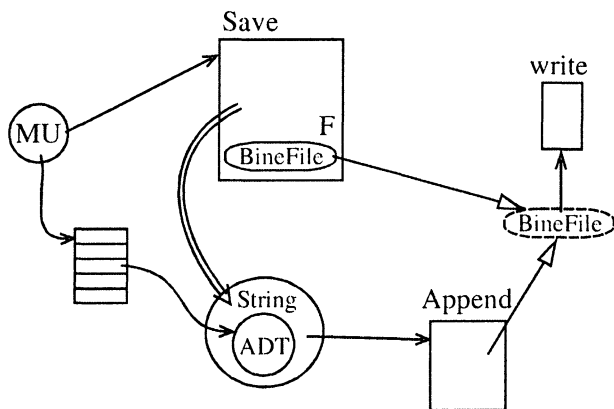


Рис. 4.12

Динамические объекты и связи между ними. Прежде всего, необходимо провести четкую грань между динамическими и обыкновенными именованными (в этом смысле – статическими) переменными и объектами. Обычные объекты, имеющие имя (локальные и глобальные), привязаны к управляющей структуре программы (функциям), а поэтому так или иначе связаны с потоком управления, то есть последовательностью вызова функций или методов. Локальные объекты, определенные внутри метода, по своей природе связаны с выполнением действий, для которых они предназначены, поэтому не имеют самостоятельного значения в программе. У глобальных объектов тоже есть недостаток: они возникают в момент начала работы программы, когда еще нет оснований для определения их свойств. К тому же количество их постоянно.

Динамические объекты могут создаваться программой когда угодно, их создание и уничтожение не связано с управляющей структурой программы. Но при их использовании возникает другая проблема: как объекты «будут знать» о существовании друг друга? В любом случае проблема «знания» упирается в вопрос: кто будет хранить указатели на динамические объекты? Известно несколько вариантов решения.

Порождение объектами объектов. Объект класса, создающий динамический объект, несет полную ответственность за работу с

ним и доступ к нему. Обычно это осуществляется в форме сеанса: объект создает динамический объект и запоминает указатель к нему, после чего может работать с этим объектом сам или передавать указатель другим объектам. По окончании работы он должен разрушить созданный им объект. Если динамических объектов несколько, то это не меняет дела: объект-прародитель должен интегрировать указатели на них в собственную структуру данных.

Интегрирование динамических объектов в структуру данных. Объекты нескольких родственных классов (например, графические элементы изображения) могут интегрироваться в общую структуру данных, через которую любой желающий получает доступ к ним. Для этого им достаточно иметь базовый класс, который при создании объекта (конструктор) включает указатель на него в структуру данных, доступную через статический объект.

Система объектов, управляемых сообщениями. До сих пор считалось, что для доступа к объекту нужно знать его имя либо иметь указатель на него. Но можно обойтись и без этого, если построить взаимодействие объектов по принципу широкоэвещательной локальной сети: «каждый со всеми». Объект имеет право послать сообщение, которое будет получено всеми объектами программы. В этом случае «правила игры» устанавливаются на основе реакции различных объектов на сообщения различных типов.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Сделать разработанный в разделах 4.1 и 4.2 тип данных производным от класса **ADT**, переопределив в нем соответствующие методы. Выполнить аналогичную процедуру еще над каким-либо простым классом (например, даты или целого числа). Переделать шаблон структуры данных из перечня, приведенного в разделе 4.3, в класс, хранящий указатели на объекты класса **ADT**. Разработать программу, демонстрирующую возможность хранения в одной структуре данных объектов различного типа.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Определить значения переменных после выполнения действий с учетом наследования.

```
//-----44-11.cpp
//----- 1
class a1{
int x;
public: a1() { x = 0; }
```

```

    a1(int n) { x = n; }
    int get() { return(x); };
class b1 : public a1 {
public:
int get() { return (a1::get()+1); }
b1(int n) : a1(n+1) {}
};
void main1()
{ a1 a(10); b1 b(12);
int x = a.get(); int y = b.get(); }
//----- 2
class a2 {
int x;
public: a2() { x = 0; }
    a2(int n) { x = n; }
    int inc() { return ++x; };
class b2: public a2 {
public: int inc() { int n = a2::inc(); return n-1; }
b2(int n) : a2(n+1) {} };
void main2()
{ a2 a(10); b2 b(12);
int x = a.inc(); int y = b.inc() + a.inc(); }
//----- 3
class a3 {
int x;
public: a3() { x = 0; }
    a3(int n) { x = n; }
    int inc() { return ++x; };
class b3 : public a3 {
public: int inc() { int n = a3::inc(); return n-1; }
b3(int n) : a3(n+1) {} };
void main3()
{ a3 a(10); b3 b(12);
a3 *pa = &b; b3 *pb = &b;
int x = a.inc();
int y = b.inc() + pa->inc();
int z = pb->inc(); }
//----- 4
class a4
{ int x;
public:
    virtual int out() { return x; }
    a4(int n) { x = n; };
class b4 : public a4 {
public: int out() { return (a4::out()+1); }
b4(int n) : a4(n) {} };
class c4 : public a4 {
public: c4(int n) : a4(n) {} };
void main4()
{ a4 A1(5); b4 B1(6); c4 C1(10);
    a4 *p[] = { &A1, &B1, &C1 };
int r1 = p[0]->out() + p[1]->out() + p[2]->out();
int r2 = A1.out() + B1.out() + C1.out(); }
//----- 5
class a5 {
public: virtual int put()=0;
    a5() {} };

```

```

class integer : public a5
{ int val;
public: int put() { return val; }
      integer(int n) { val=n; }
};
class string : public a5{
char *str;
public: int put() { return strlen(str); }
      string(char *s) { str = s; }
};
void main5(){
integer a1(12),a2(24); string a3("aaaa");
a5 *p[4]= { &a1, &a2, &a3, &a1 };
for ( int x = 0, i = 0; i < 4; i++) x += p[i]->put(); }
//----- 6
class mem {
protected:
void *addr;
public: mem() {}
      virtual int put() { return 0; } };
class integer1 : public mem {
public: int put() { return *(int*)addr; }
      integer1(int &d) { addr = (void *)&d; } };
class string1 : public mem {
public: int put() { return strlen((char*)addr); }
      string1(char *p) { addr = (void*)p; } };
void main6() {
int x=12; integer1 i0(x),i1(x); string1 s0("aaaa"),s1("bb");
mem *p[4] = { &i0, &i1, &s0, &s1 };
int n1 = i0.put() + s0.put();
for ( int i=0,n2=0; i<4; i++) n2 += p[i]->put(); }

```

ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы и структуры данных. – СПб.: Невский диалект, 2001. – 351 с.
2. Подбельский В.В., Фомин С.С. Программирование на языке Си. – М.: Финансы и статистика, 1999. – 600 с.
3. Подбельский В.В. Язык Си++: Учеб. пособие. – М.: Финансы и статистика, 1995. – 560 с.
4. Дейтел Х.М., Дейтел П.Дж. Как программировать на Си++. – М.: БИНОМ, 1999. – 1000 с.
5. Топп У., Форд У. Структуры данных в Си++. – М.: БИНОМ, 1999. – 800 с.
6. Климова Л.М. Основы практического программирования на языке Си. – М.: ПРИОР, 1999. – 464 с.
7. Керниган Б., Ритчи Д. Язык программирования Си. – М.: Финансы и статистика, 1992. – 271 с.
8. Кнут Д. Искусство программирования для ЭВМ. – Т. 3. Сортировка и поиск. – М.: Изд. дом «Вильямс», 2000. – 832 с.
9. Страуструп Б. Язык программирования С++. – Киев: Диасофт, 2001. – 900 с.
10. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск. – Киев: Диасофт, 2001. – 688 с.

СОДЕРЖАНИЕ

Предисловие.....	5
1. Анализ и проектирование программ.....	7
1.1. Прежде чем начать	7
1.2. Как работает программа.....	8
1.3. Стандартные программные контексты	15
1.4. Процесс проектирования программы	42
1.5. Структурное программирование	54
1.6. Модульное программирование.....	68
1.7. Логическое и «историческое» в программировании	77
2. Программист «начинающий»	83
2.1. Арифметические задачи.....	84
2.2. Итерационные циклы и приближенные вычисления.....	94
2.3. Структуры данных. Последовательность. Стек. Очередь	100
2.4. Символы. Строки. Текст	106
2.5. Сортировка и поиск.....	120
2.6. Указатели.....	137
2.7. Структурированные типы	159
2.8. Типы данных, переменные, функции.....	170
3. Программист «системный»	184
3.1. Указатели и управление памятью.....	185
3.2. Динамические переменные и массивы.....	201
3.3. Динамическое связывание	209
3.4. Рекурсия	218
3.5. Структуры данных. Массивы указателей	241

3.6. Структуры данных. Линейные списки	256
3.7. Структуры данных. Деревья	269
3.8. Иерархические структуры данных	287
3.9. Биты, байты, машинные слова.....	298
3.10. Двоичные файлы произвольного доступа	325
4. Программист «объектно-ориентированный»	354
4.1. Программирование объектов. Конструкторы	355
4.2. Программирование методов. Переопределение операций.....	371
4.3. Классы структур данных. Шаблоны.....	388
4.4. Наследование и полиморфизм.....	404
Литература.....	425